Metal Shading Language Specification

Version 4.1

Developer

Contents

Tables and Figures

1 Introduction

1.1 Purpose of This Document

Metal enables you to develop apps that take advantage of the graphics and compute processing power of the GPU. This document describes the Metal Shading Language (MSL), which you will use to write a *shader program*, which is graphics and data-parallel compute code that runs on the GPU. Shader programs run on different programmable units of the GPU. MSL is a single, unified language that allows tighter integration between the graphics and compute programs. Since MSL is C++-based, you will find it familiar and easy to use.

MSL works with the Metal framework, which manages the execution and optionally the compilation of the Metal programs. Metal uses clang and LLVM so you get a compiler that delivers optimized performance on the GPU.

1.2 Organization of This Specification

This document is organized into the following chapters:

- This chapter, “Introduction,” is an introduction to this document that covers the similarities and differences between Metal and C++17. It also details the options for the Metal compiler, including preprocessor directives, options for math intrinsics, and options for controlling optimization.
- “Data Types” lists the Metal data types, including types that represent vectors, matrices, buffers, textures, and samplers. It also discusses type alignment and type conversion.
- “Operators” lists the Metal operators.
- “Address Spaces” describes disjoint address spaces for allocating memory objects with access restrictions.
- “Function and Variable Declarations” details how to declare functions and variables, with optional attributes that specify restrictions.
- “Metal Standard Library” defines a collection of built-in Metal functions.
- “Numerical Compliance” describes requirements for representing floating-point numbers, including accuracy in mathematical operations.

iOS and macOS support for features (functions, enumerations, types, attributes, or operators) described in this document is available since Metal 1, unless otherwise indicated.

For the rest of this document, the abbreviation X.Y stands for “Metal version X.Y”; for example, 2.1 indicates Metal 2.1. Please note that though a feature is supported in MSL shading language, it may not be supported on all GPUs. Please refer to the Metal Feature Set Tables at developer.apple.com.

1.3 New in Metal 4.1

Metal 4.1 introduces the following new features:

- Support for placement `new` (see sections 1.5.4 and 6.2)
- An option to set default rounding mode for float-to-float conversions to round toward zero (section 1.6.3)
- Adds `function_id` to intersection result type (sections 2.17.4 and 2.17.5)
- Support for packed numeric types for block-scaling formats (section 2.21)
- Support for multiplane tensors and `tensor_blockwise` with block-scaling data formats (section 2.22)
- Support for `deinterleave` and `interleave` (section 6.4)
- Support for memory order (including acquire and release) to barriers and atomics (sections 6.10, 6.16.1.2, and 6.16.4)
- Support for clamp-to-edge texture reads, integer-coordinate texture reads with offsets, and multi-pixel texture reads (section 6.13)
- Updates the supported data types for Metal Performance Primitives matrix multiply (section 17.2.1)

1.4 References

Metal

Here is a link to the Metal documentation on apple.com:

https://developer.apple.com/documentation/metal

1.5 Metal and C++17

In Metal 4 and later, the Metal programming language is a C++17-based specification with extensions and restrictions. Refer to the C++17 specification (also known as ISO/IEC 14882:2017) for a detailed description of the language grammar. Prior language versions of Metal are a C++14-based specification with extensions and restrictions.

This section and its subsections describe the modifications and restrictions to the C++17 and C++14 language supported in Metal.

For more about Metal preprocessing directives and compiler options, see section 1.6 of this document.

1.5.1 Overloading

Metal supports overloading, as defined by section 13 of the C++17 and C++14 specification. Metal extends the function overloading rules to include the address space attribute of an

argument. You cannot overload Metal graphics and kernel functions. (For a definition of graphics and kernel functions, see section 5.1 of this document.)

1.5.2 Templates

Metal supports templates, as defined by section 14 of the C++17 and C++14 specification.

1.5.3 Preprocessing Directives

Metal supports the preprocessing directives, as defined by section 16 of the C++17 and C++14 Specification.

1.5.4 Restrictions

All OS: Metal 3.2 and later support lambda expressions.

All OS: Metal 4.1 and later support placement `new`.

The following C++17 features are not available in Metal (section numbers in this list refer to the C++17 Specification):

- lambda expressions (section 5.1.2) prior to Metal 3.2
- `dynamic_cast` operator (section 5.2.7)
- type identification (section 5.2.8)
- `new` and `delete` operators (sections 5.3.4 and 5.3.5). Metal 4.1 and later supports placement `new`.
- `noexcept` operator (section 5.3.7)
- `goto` statement (section 6.6)
- `register`, `thread_local` storage attributes (section 7.1.1)
- `virtual` function attribute (section 7.1.2)
- derived classes (section 10, section 11)
- exception handling (section 15)

Do not use the C++ standard library in Metal code. Instead, Metal has its own standard library, as discussed in section 5 of this document.

Metal restricts the use of pointers:

- You must declare arguments to Metal graphics and kernel functions that are pointers with the Metal `device`, `constant`, `threadgroup`, `threadgroup_imageblock`, `object_data`, or `ray_data` address space attribute. (For more about Metal address space attributes, see section 4 of this document.)
- Metal 2.3 and later support function pointers.

Metal supports recursive function calls (C++ section 5.2.2, item 9) in compute (kernel) context starting with Metal 2.4.

You can't call a Metal function `main`.

1.6 Compiler and Preprocessor

You can use the Metal compiler online (with the appropriate APIs to compile Metal sources) or offline. You can load Metal sources that are compiled offline as binaries, using the appropriate Metal APIs.

This section explains the compiler options supported by the Metal compiler and categorizes them as preprocessor options, options for math intrinsics, options that control optimization, miscellaneous compilation options, and linking.

1.6.1 Preprocessor Compiler Options

The following options control the Metal preprocessor that runs on each program source before actual compilation:

`-D name`

Predefine `name` as a macro, with definition 1.

`-D name=definition`

Metal tokenizes and processes the contents of `definition` as if they appear in a `#define` directive. This option allows you to compile Metal code to enable or disable features. You may use this option multiple times, and the preprocessor processes the definitions in the order in which they appear.

`-I dir`

Add the directory `dir` to the search path of directories for header files. This option is only available for the offline compiler.

1.6.2 Preprocessor Definitions

The Metal compiler sets a number of preprocessor definitions by default, including:

```
__METAL_VERSION__ // Set to the Metal language revision
__METAL_MACOS__ // Set if compiled with the macOS Metal language
__METAL_IOS__   // Set if compiled with the iOS Metal language
__METAL__       // Set if compiled with the unified Metal language
                // Set with -std=metal3.0 or above
```

You can use definitions to conditionally apply shading language features that are only available on later language version (see section 1.6.10 Compiler Options Controlling the Language Version).

The version number is MajorMinorPatch. For example, for Metal 1.2, patch 0, `__METAL_VERSION__` is 120; for Metal 2.1, patch 1, `__METAL_VERSION__` is 211.

To conditionally include code that uses features introduced in Metal 2, you can use the preprocessor definition in code, as follows:

```
#if __METAL_VERSION__ >= 200
// Code that requires features introduced in Metal 2.
#endif
```

1.6.3 Math Intrinsics Compiler Options

The following section describes options to control compiler behavior regarding floating-point arithmetic, trading off between speed and correctness.

For more about math functions, see section 6.6. For more about the relative errors of ordinary and fast math functions, see section 8.4.

The options enable or disable the optimizations for floating-point arithmetic that may violate the IEEE 754 standard. They also enable or disable the high precision variant of math functions for single precision floating-point scalar and vector types.

The fast math optimizations for floating-point arithmetic include:

- **No NaNs**: Allow optimizations to assume the arguments and result are not NaN (not a number).
- **No INFs**: Allow optimizations to assume the arguments and result are not positive or negative infinity.
- **No Signed Zeroes**: Allow optimizations to treat the sign of a zero argument or result as insignificant.
- **Allow Reciprocal**: Allow optimizations to use the reciprocal of an argument rather than perform a division.
- **Allow Reassociation**: Allow algebraically equivalent transformations, such as reassociating floating-point operations that may dramatically change the floating-point results.
- **Allow Contract**: Allow floating-point contraction across statements. For example, allow fusing a multiple followed by an additional into a single fused-multiply-add.

In Xcode 16 and later and Metal Developer Tools, Metal supports the following options for Windows 5 (SDK supporting iOS 18 or macOS 15):

`-fmetal-math-fp32-functions=<fast|precise>`

This option sets the single-precision floating-point math functions described in section 6.6 to call either the `fast` or `precise` version. The default is `fast`. For Apple silicon, starting with Apple GPU Family 4, the math functions honor INF and NaN.

`-fmetal-math-mode=<fast, relaxed, safe>`

This option sets how aggressive the compiler can be with floating-point optimizations. The default is `fast`.

If you set the option to `fast`, it lets the compiler make aggressive, potentially lossy assumptions about floating-point math. These include no NaNs, no INFs, no signed zeros, allow reciprocal, allow reassociation, and FP contract to be fast.

If you set the option to `relaxed`, it lets the compiler make aggressive, potentially lossy assumptions about floating-point math, but honors INFs and NaNs. These include no signed zeros, allow reciprocal, allow reassociation, and FP contract to be fast. This supports Apple silicon.

If you set the option to `safe`, it disables unsafe floating-point optimizations by preventing the compiler from making any transformations that might affect the results. This sets the FP contract to on.

Metal supports the following legacy options:

`-ffast-math`

Equivalent to `-fmetal-math-fp32-functions=fast` and `-fmetal-math-mode=fast`.

`-fno-fast-math`

Equivalent to `-fmetal-math-fp32-functions=precise` and `-fmetal-math-mode=safe`.

When utilizing fast math in your program, it is important to understand that the compiler can assume certain properties and make optimizations accordingly. For example, the use of fast math asserts that the shader will never generate `INF` or `NaN`. If the program has an expression X/Y, the compiler can assume Y is never zero as this could potentially result in positive/negative infinite or `NaN`, depending on the value of X. If Y can be zero, you would have an undefined program if compiled with fast math.

The `#pragma metal fp` pragmas allow you to specify floating-point options for a source code section.

The following pragma has the same semantics to allow you to specify precise floating-point semantics and floating-point exception behavior for a source code section. It can only appear in file or namespace scope, within a language linkage specification, or at the start of a compound statement (excluding comments). When using it within a compound statement, the pragma is active within the scope of the compound statement:

```
#pragma METAL fp math_mode([relaxed | safe | fast])
```

By default, the compiler allows floating-point contractions. For example, `a*b+c` may be converted to a single fused-multiply-add. These contractions could lead to computation differences if other expressions are not contracted. To disable allowing the compiler to contractions, pass the following option:

```
-ffp-contract=off
```

The compiler also supports controlling contractions with the following pragma:

```
#pragma METAL fp contract([off | on | fast])
```

Using `off` disables contractions, `on` allows contractions with statement, and `fast` allows contractions across statements. You can also use:

```
#pragma STDC FP_CONTRACT OFF
```

In Metal 4.1 and later, use the following option to change the default rounding mode for float-to-float conversions from RTNE (round to nearest, ties to even) to RTZ (round toward zero) for compatibility with non-Metal APIs:

```
-fmetal-rtz-fp-conversion
```

1.6.4 Invariance Compiler Options

If you are building with an SDK that supports iOS 14 or macOS 11, you need to pass the following option to support vertex invariance:

`-fpreserve-invariance`

Preserve invariant for computations marked with `[[invariant]]` in vertex shaders. If not set, `[[invariant]]` is ignored.

In previous versions of Metal, `[[invariant]]` was a best-effort analysis to mark which operations need to be invariant and may fail in certain cases. This is replaced with a conservative invariant model where the compiler marks operations that doesn't go into an invariant calculation. This will guarantee anything that is invariant calculation remains invariant. This option may reduce performance as it may prevent certain optimizations to preserve invariance.

1.6.5 Optimization Compiler Options

These options control the optimization level of the compiler:

`-O2`

Optimize for performance (default).

`-Os`

Like `-O2` with extra optimizations to reduce code size.

1.6.6 Maximum Total Threadgroup Size Option

All OS: Metal 3 and later support maximum total threadgroup size option.

This option specifies the number of threads (value) in a threadgroup for every function in the translation unit:

```
-fmax-total-threads-per-threadgroup=<value>
```

The attribute `[[max_total_threads_per_threadgroup]]` function attribute described in section 5.1.3, section 5.1.7, and section 5.1.8 takes precedence over the compile option. The value must fit within 32 bits.

This option is useful for setting the option to enable functions compiled for a dynamic library to be compatible with a PSO.

1.6.7 Texture Write Rounding Mode

Configure the rounding mode for texture writes to floating-point pixel types by setting the `-ftexture-write-rounding-mode` compiler flag to one of the options in Table 1.1.

Table 1.1. Rounding mode

Rounding mode	Description
`native` (default)	Texture writes use the hardware's native rounding strategy.
`rte` All OS: Metal 2.3 and later	Texture writes round to the nearest even number.
`rtz` All OS: Metal 2.3 and later	Texture writes round toward zero.

The `-ftexture-write-rounding-mode` flag is available for these SDKs:

- macOS 11 and later
- iOS 14 and later

For more information about which GPU families support rounding modes other than `native` and what native defaults to, see the Metal Feature Set Tables.

1.6.8 Compiler Options to Enable Modules

The compiler supports multiple options to control the use of modules. These options are only available for the offline compiler:

`-fmodules`

Enable the modules feature.

`-fimplicit-module-maps`

Enable the implicit search for module map files named `module.modulemap` or a similar name. By default, `-fmodules` enables this option. (The compiler option `-fno-implicit-module-maps` disables this option.)

`-fno-implicit-module-maps`

Disable the implicit search for module map files named `module.modulemap`. module map files are only loaded if they are explicitly specified with `-fmodule-map-file` or transitively used by another module map file.

`-fmodules-cache-path=<directory>`

Specify the path to the modules cache. If not provided, the compiler selects a system-appropriate default.

`-fmodule-map-file=<file>`

Load the specified module map file, if a header from its directory or one of its subdirectories is loaded.

If you are building with an SDK that supports iOS 16 or macOS 13, `-fmodules` has the following additional options:

`-fmodules=[mode]`

Supported values for modes are:

`stdlib`: Enable the modules feature but restrict the search for module maps to the Metal standard library. Enabled by default with an SDK that supports iOS 16 or macOS 13.

`all`: Enable the modules feature (equivalent to `-fmodules`).

`none`: Disable the modules feature.

1.6.9 Compiler Options to Enable Logging

All OS: Metal 3.2 and later support logging for Apple silicon.

You need to provide the following compiler option to enable logging (see section 6.20) during compilation:

`-fmetal-enable-logging`

1.6.10 Compiler Options Controlling the Language Version

The following option controls the version of the unified graphics and computing language accepted by the compiler:

`-std=`

Determine the language revision to use. A value for this option must be provided, which must be one of:

- `ios-metal1.0`: Supports the unified graphics and computing language revision 1 programs for iOS 8. [[Deprecated]]

- `ios-metal1.1`: Supports the unified graphics and computing language revision 1.1 programs for iOS 9.
- `ios-metal1.2`: Supports the unified graphics and computing language revision 1.2 programs for iOS 10.
- `ios-metal2.0`: Supports the unified graphics and computing language revision 2 programs for iOS 11.
- `ios-metal2.1`: Supports the unified graphics and computing language revision 2.1 programs for iOS 12.
- `ios-metal2.2`: Supports the unified graphics and computing language revision 2.2 programs for iOS 13 and iPadOS 13.1.
- `ios-metal2.3`: Supports the unified graphics and computing language revision 2.3 programs for iOS 14 and iPadOS 14.
- `ios-metal2.4`: Supports the unified graphics and computing language revision 2.4 programs for iOS 15 and iPadOS 15.
- `macos-metal1,1` or `osx-metal1.1`: Supports the unified graphics and computing language revision 1.1 programs for macOS 10.11.
- `macos-metal1.2` or `osx-metal1.2`: Supports the unified graphics and computing language revision 1.2 programs for macOS 10.12.
- `macos-metal2.0` or `osx-metal2.0`: Supports the unified graphics and computing language revision 2 programs for macOS 10.13.
- `macos-metal2.1`: Supports the unified graphics and computing language revision 2.1 programs for macOS 10.14.
- `macos-metal2.2`: Supports the unified graphics and computing language revision 2.2 programs for macOS 10.15.
- `macos-metal2.3`: Supports the unified graphics and computing language revision 2.3 programs for macOS 11.
- `macos-metal2.4`: Supports the unified graphics and computing language revision 2.4 programs for macOS 12.

Note that `macos-*` is available in macOS 10.13 SDK and later.

In iOS 16, macOS 13, and tvOS 16 and later, Metal has unified the shading language between the platforms:

- `metal3.0`: Supports the unified graphics and computing language revision 3 programs for iOS 16, iPadOS 16, macOS 13, and tvOS 16.
- `metal3.1`: Supports the unified graphics and computing language revision 3.1 programs for iOS 17, iPadOS 17, macOS 14, tvOS 17, and visionOS 1.

Only Apple silicon supports new features in language standard 3.2 and above:

- `metal3.2`: Supports the unified graphics and computing language revision 3.2 programs for iOS 18, iPadOS 18, macOS 15, tvOS 18, and visionOS 2.
- `metal4.0`: Supports the unified graphics and computing language revision 4 programs for iOS 26, iPadOS 26, macOS 26, tvOS 26, and visionOS 26.
- `metal4.1`: Supports the unified graphics and computing language revision 4.1 programs for iOS 27, iPadOS 27, macOS 27, tvOS 27, and visionOS 27.

1.6.11 Compiler Option to Warn About Missing Metal Address Spaces

You can use the following option to warn about missing address space qualifiers on member functions.

```
-Wmetal-addr-spaces
```

The compiler can update the code using its fix-it capabilities. For example, you can pass the following option to fix missing thread address space qualifiers:

```
-Xclang -fixit -Xclang -fix-what-you-can -Wmetal-addr-spaces
```

1.6.12 Compiler Options to Request or Suppress Warnings

The following options are available:

`-Werror`

Make all warnings into errors.

`-w`

Inhibit all warning messages.

1.6.13 Target Conditionals

Metal defines several macros which one can use to determine what platform the shader is running on. The following macros are defined in `<TargetConditionals.h>`:

```
TARGET_OS_MAC          : Generated code runs under Mac OS X variant
TARGET_OS_OSX          : Generated code runs under OS X devices
TARGET_OS_IPHONE       : Generated code for firmware, devices or simulator
TARGET_OS_IOS          : Generated code runs under iOS
TARGET_OS_TV           : Generated code runs under tvOS
TARGET_OS_MACCATALYST  : Generated code runs under macOS
TARGET_OS_SIMULATOR    : Generated code runs under a simulator
TARGET_OS_VISION       : Generated code runs under visionOS
                         (Available in SDKs in late 2023)
```

Note that this header is not part of `<metal_stdlib>`.

1.6.14 Dynamic Library Linker Options

The Metal compiler driver can pass options to the linker. Here is a brief description of some of these options. See the Metal linker for more information:

`-dynamiclib`

Specify that the output is a dynamic library.

`-install_name`

Used with `-dynamiclib` to specify the location of where the dynamic library is expected be installed and found by the loader. Use with `@executable_path` and `@loader_path`.

1.6.15 Options for Compiling to GPU Binaries

The following options are available for compiling to a GPU binary if you are building with an SDK that supports iOS 16 or macOS 13:

`-arch [architecture]`

Specify the architecture to build for.

`-gpu-family [gpu family name]`

Specify the architectures associated with the `MTLGPUFamily` to build for. See `MTLGPUFamily` in Metal API for the list of available families.

`-N [descriptor.mtlp-json]`

Specify the pipeline descriptors in Metal script format. The descriptor files must end in `.mtlp-json`.

1.6.16 Options for Generating Metal Library Symbol Files

If you are building with an SDK that supports iOS 15 or macOS 12, the following option is available to generate a Metal library symbol file:

`-frecord-sources`

Enable the compiler to store source information into the AIR or Metal library file (`.metallib`).

`-frecord-sources=flat`

Enable the compiler to store source information if generating an AIR file. Enable the compiler to store the source information in a symbol companion file (`.metallibsym`) if generating a Metal Library file.

See Generating and loading a Metal library symbol file at developer.apple.com for more information.

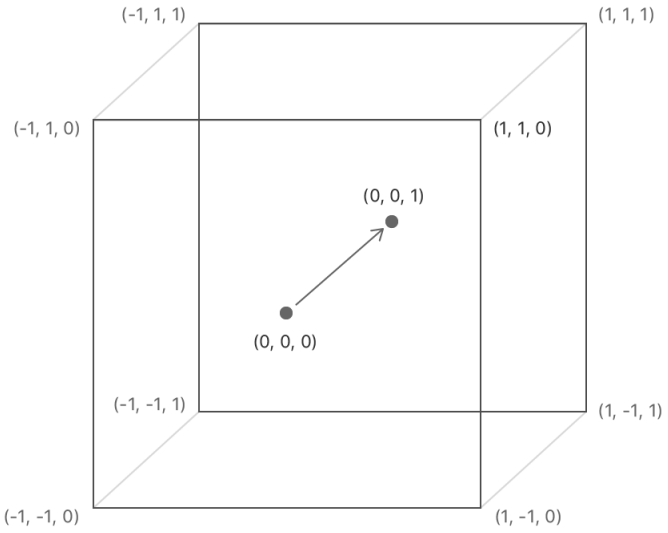
1.7 Metal Coordinate Systems

Metal defines several standard coordinate systems to represent transformed graphics data at different stages along the rendering pipeline.

A four-dimensional homogenous vector `(x,y,z,w)` specifies a three-dimensional point in *clip-space coordinates*. A vertex shader generates positions in clip-space coordinates. Metal divides the `x`, `y`, and `z` values by `w` to convert clip-space coordinates into *normalized device coordinates*.

Normalized device coordinates use a *left-handed coordinate system* (see Figure 1) and map to positions in the viewport. These coordinates are independent of viewport size. The lower-left corner of the viewport is at an `(x,y)` coordinate of `(-1.0,-1.0)` and the upper corner is at `(1.0,1.0)`. Positive-`z` values point away from the camera ("into the screen"). The visible portion of the `z` coordinate is between `0.0` and `1.0`. The Metal rendering pipeline clips primitives to this box.

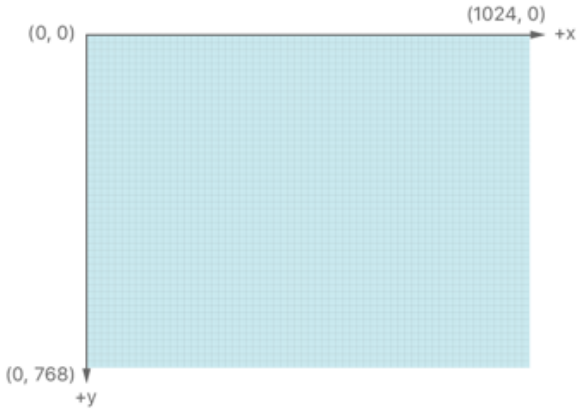
Figure 1. Normalized device coordinate system



The rasterizer stage transforms normalized-device coordinates (NDC) into *viewport coordinates* (see Figure 2). The `(x,y)` coordinates in this space are measured in pixels, with the origin in the top-left corner of the viewport and positive values going to the right and down. You specify viewports in this coordinate space, and the Metal maps NDC coordinates to the extents of the viewport.

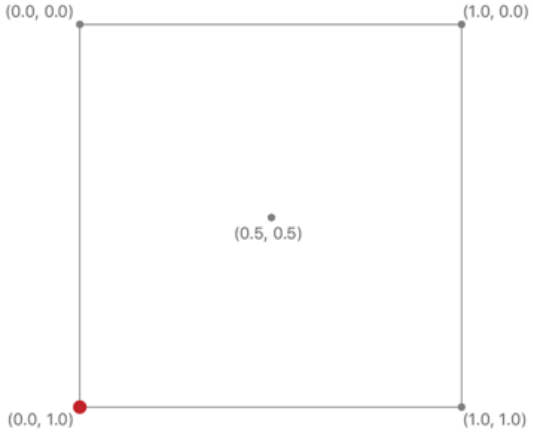
If you are using variable rasterization rate (see Section 6.15), then the viewport coordinate system is a logical coordinate system independent of the render target's physical layout. A rate map determines the relationship between coordinates in this logical coordinate system (sometimes called *screen space*) and pixels in the render targets (physical coordinates).

Figure 2. Viewport coordinate system



Texture coordinates use a similar coordinate system to viewport coordinates. Texture coordinates can also be specified using *normalized texture coordinates*. For 2D textures, normalized texture coordinates are values from `0.0` to `1.0` in both `x` and `y` directions, as seen in Figure 3. A value of (`0.0`, `0.0`) specifies the pixel at the first byte of the image data (the top-left corner of the image). A value of (`1.0`, `1.0`) specifies the pixel at the last byte of the image data (the bottom-right corner of the image).

Figure 3. Normalized 2D texture coordinate system



2 Data Types

This chapter details the Metal data types, including types that represent vectors and matrices. The chapter also discusses atomic data types, buffers, textures, samplers, arrays, user-defined structures, type alignment, and type conversion.

2.1 Scalar Data Types

Metal supports the scalar types listed in Table 2.1. Metal does **not** support the `double`, `long long`, `unsigned long long`, and `long double` data types.

Table 2.1. Metal scalar data types

Type	Description
`bool`	A conditional data type that has the value of either `true` or `false`. The value `true` expands to the integer constant 1, and the value `false` expands to the integer constant 0.
`char` `int8_t`	A signed two's complement 8-bit integer.

`unsigned char`
`uchar`
`uint8_t`
	An unsigned 8-bit integer.

`short` `int16_t`	A signed two's complement 16-bit integer.

`unsigned short`
`ushort`
`uint16_t`
	An unsigned 16-bit integer.

`int` `int32_t`	A signed two's complement 32-bit integer.

`unsigned int`
`uint`
`uint32_t`
	An unsigned 32-bit integer.

`long`
`int64_t`
All OS: Metal 2.2 and later
	A signed two's complement 64-bit integer.

Type	Description

`unsigned long`
`ulong`
`uint64_t`
All OS: Metal 2.2 and later
	An unsigned 64-bit integer.

`half`	A 16-bit floating-point. The `half` data type must conform to the IEEE 754 binary16 storage format.
`bfloat` All OS: Metal 3.1 and later	A 16-bit brain floating-point. The `bfloat` data type is a truncated version of `float` for machine learning applications, using an 8-bit (7 explicitly stored) rather than 24-bit mantissa).
`float`	A 32-bit floating-point. The `float` data type must conform to the IEEE 754 single precision storage format.
`size_t`	An unsigned integer type of the result of the `sizeof` operator. This is a 64-bit unsigned integer.
`ptrdiff_t`	A signed integer type that is the result of subtracting two pointers. This is a 64-bit signed integer.
`void`	The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed.

Metal supports:

- the `f` or `F` suffix to specify a single precision floating-point literal value (such as `0.5f` or `0.5F`).
- the `h` or `H` suffix to specify a half precision floating-point literal value (such as `0.5h` or `0.5H`).
- the `bf` or suffix to specify a brain precision floating-point literal value (such as `0.5bf` or `0.5BF`).
- the `u` or `U` suffix for unsigned integer literals.
- the `l` or `L` suffix for signed long integer literals.

Table 2.2 lists the size and alignment of most of the scalar data types.

Table 2.2. Size and alignment of scalar data types

Type	Size (in bytes)	Alignment (in bytes)
`bool`	1	1

Type	Size (in bytes)	Alignment (in bytes)

`char`
`int8_t`
`unsigned char`
`uchar`
`uint8_t`
	1	1

`short`
`int16_t`
`unsigned short`
`ushort`
`uint16_t`
	2	2

`int`
`int32_t`
`unsigned int`
`uint`
`uint32_t`
	4	4

`long`
`int64_t`
`unsigned long`
`uint64_t`
	8	8

`size_t`	8	8
`half`	2	2
`bfloat`	2	2
`float`	4	4

2.2 Vector Data Types

Metal supports a subset of the vector data types implemented by the system vector math library. Metal supported these vector type names, where `n` is `2`, `3`, or `4`, representing a 2-, 3-, or 4-component vector type, respectively:

- `booln`
- `charn`
- `shortn`
- `intn`
- `longn`
- `ucharn`
- `ushortn`
- `uintn`

- `ulongn`
- `halfn`
- `bfloatn` (Metal 3.1 and later)
- `floatn`

Metal also supports `vec<T,n>` where `T` is a valid scalar type and `n` is `2`, `3`, or `4`, representing a 2-, 3-, or 4- component vector type.

Table 2.3 lists the size and alignment of the vector data types.

Table 2.3. Size and alignment of vector data types

Type	Size (in bytes)	Alignment (in bytes)
`bool2`	2	2
`bool3`	4	4
`bool4`	4	4
`char2` `uchar2`	2	2
`char3` `uchar3`	4	4
`char4` `uchar4`	4	4
`short2` `ushort2`	4	4
`short3` `ushort3`	8	8
`short4` `ushort4`	8	8
`int2` `uint2`	8	8
`int3` `uint3`	16	16
`int4` `uint4`	16	16
`long2` `ulong2`	16	16

Type	Size (in bytes)	Alignment (in bytes)
`long3` `ulong3`	32	32
`long4` `ulong4`	32	32
`half2`	4	4
`half3`	8	8
`half4`	8	8
`bfloat2`	4	4
`bfloat3`	8	8
`bfloat4`	8	8
`float2`	8	8
`float3`	16	16
`float4`	16	16

2.2.1 Accessing Vector Components

You can use an array index to access vector components. Array index `0` refers to the first component of the vector, index `1` to the second component, and so on. The following examples show various ways to access array components:

```
pos = float4(1.0f, 2.0f, 3.0f, 4.0f);

float x = pos[0]; // x = 1.0
float z = pos[2]; // z = 3.0

float4 vA = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 vB;

for (int i=0; i<4; i++)
     vB[i] = vA[i] * 2.0f // vB = (2.0, 4.0, 6.0, 8.0);
```

Metal supports using a period (`.`) as a selection operator to access vector components, using letters that may indicate coordinate or color data:

```
<vector_data_type>.xyzw
<vector_data_type>.rgba
```

The following code initializes a vector test and then uses the `.xyzw` or `.rgba` selection syntax to access individual components:

```
int4 test = int4(0, 1, 2, 3);
int a = test.x; //  a = 0
int b = test.y; //  b = 1
int c = test.z; //  c = 2
int d = test.w; //  d = 3
int e = test.r; //  e = 0
int f = test.g; //  f = 1
int g = test.b; //  g = 2
int h = test.a; //  h = 3
```

The component selection syntax allows the selection of multiple components:

```
float4 c;
c.xyzw = float4(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = float2(3.0f, 4.0f);
c.xyz = float3(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows the permutation or replication of components:

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz = pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left-hand side (lvalue) of an expression. To form the lvalue, you may apply swizzling. The resulting lvalue may be either the scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type. The resulting lvalue of vector type must not contain duplicate components.

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
// pos = (5.0, 2.0, 3.0, 6.0)
pos.xw = float2(5.0f, 6.0f);
```

```
// pos = (8.0, 2.0, 3.0, 7.0)
pos.wx = float2(7.0f, 8.0f);

// pos = (3.0, 5.0, 9.0, 7.0)
pos.xyz = float3(3.0f, 5.0f, 9.0f);
```

When assigning a swizzled value to a variable, the GPU may need to read the existing value, modify it, and write the result back. The assignment to `pos.xw` in the example above, causes the GPU to load the `float4` value, shuffle values `5.0f` and `6.0f` into it, and the write back the result back into pos. If two threads write to different components of the vector at the same time, the result is undefined.

The following methods of vector component access are not permitted and result in a compile-time error:

- Accessing components beyond those declared for the vector type is an error. 2-component vector data types can only access `.xy` or `.rg` elements. 3-component vector data types can only access `.xyz` or `.rgb` elements.

  ```
  float2 pos; // This is a 2-component vector.
  pos.x = 1.0f; // x is legal and so is y.
  pos.z = 1.0f; // z is illegal and so is w. z is the 3rd component.
  float3 pos; // This is a 3-component vector.
  pos.z = 1.0f; // z is legal for a 3-component vector.
  pos.w = 1.0f; // This is illegal. w is the 4th component.
  ```

- Accessing the same component twice on the left-hand side is ambiguous and is an error:

  ```
  // This is illegal because 'x' is used twice.
  pos.xx = float2(3.0f, 4.0f);
  ```

- Accessing a different number of components is an error:

  ```
  // This is illegal due to a mismatch between float2 and float4.
  pos.xy = float4(1.0f, 2.0f, 3.0f, 4.0f);
  ```

- Intermixing the `.rgba` and `.xyzw` syntax in a single access is an error:

  ```
  float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
  pos.x = 1.0f; // OK
  pos.g = 2.0f; // OK

  // These are illegal due to mixing rgba and xyzw attributes.
  ```

```
pos.xg = float2(3.0f, 4.0f);
float3 coord = pos.ryz;
```

- A pointer or reference to a vector with swizzles is an error:

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
my_func(&pos.xy); // This is an illegal pointer to a swizzle.
```

The `sizeof` operator on a vector type returns the size of the vector. This is typically the number of components * size of each component, except for 3-component vectors whose size is the same as the 4-component vector (see Table 2.3) . For example, `sizeof(float4)` returns 16 and `sizeof(half4)` returns 8.

2.2.2 Vector Constructors

You can use constructors to create vectors from a set of scalars or vectors. The parameter signature determines how to construct and initialize a vector. For instance, if the vector is initialized with only a single scalar parameter, all components of the constructed vector are set to that scalar value.

If you construct a vector from multiple scalars, one or more vectors, or a mixture of scalars and vectors, Metal consumes the vector's components in order from the components of the arguments. Metal consumes the arguments from left to right. Metal consumes all of an argument's components, in order, before any components from the following argument.

This is a list of constructors for `float4`:

```
float4(float x);
float4(float x, float y, float z, float w);
float4(float2 a, float2 b);
float4(float2 a, float b, float c);
float4(float a, float b, float2 c);
float4(float a, float2 b, float c);
float4(float3 a, float b);
float4(float a, float3 b);
float4(float4 x);
```

This is a list of constructors for `float3`:

```
float3(float x);
float3(float x, float y, float z);
float3(float a, float2 b);
float3(float2 a, float b);
```

```
float3(float3 x);
```

This is a list of constructors for `float2`:

```
float2(float x);
float2(float x, float y);
float2(float2 x);
```

The following examples illustrate uses of the constructors:

```
float x = 1.0f, y = 2.0f, z = 3.0f, w = 4.0f;
float4 a = float4(0.0f);
float4 b = float4(x, y, z, w);
float2 c = float2(5.0f, 6.0f);

float2 a = float2(x, y);
float2 b = float2(z, w);
float4 x = float4(a.xy, b.xy);
```

Under-initializing a vector constructor results in a compile-time error.

2.2.3 Packed Vector Types

You must align the vector data types described in section 2.2 to the size of the vector. You can also require their vector data to be tightly packed; for example, a vertex structure that may contain position, normal, tangent vectors and texture coordinates tightly packed and passed as a buffer to a vertex function.

The supported packed vector type names are:

- `packed_charn`
- `packed_shortn`
- `packed_intn`
- `packed_ucharn`
- `packed_ushortn`
- `packed_uintn`
- `packed_halfn`
- `packed_bfloatn` (Metal 3.1 and later)
- `packed_floatn`
- `packed_longn` (Metal 2.3 and later)

Where `n` is `2`, `3`, or `4`, representing a 2-, 3-, or 4-component vector type, respectively. (The `packed_booln` vector type names are reserved.)

Metal also supports `packed_vec<T,n>` where `T` is a valid scalar type and `n` is `2`, `3`, or `4`, representing a 2-, 3-, or 4-component packed vector type.

Table 2.4 lists the size and alignment of the packed vector data types.

Table 2.4. Size and alignment of packed vector data types

Type	Size (in bytes)	Alignment (in bytes)
`packed_char2`, `packed_uchar2`	2	1
`packed_char3`, `packed_uchar3`	3	1
`packed_char4`, `packed_uchar4`	4	1
`packed_short2`, `packed_ushort2`	4	2
`packed_short3`, `packed_ushort3`	6	2
`packed_short4`, `packed_ushort4`	8	2
`packed_int2`, `packed_uint2`	8	4
`packed_int3`, `packed_uint3`	12	4
`packed_int4`, `packed_uint4`	16	4
`packed_half2`	4	2
`packed_half3`	6	2
`packed_half4`	8	2
`packed_bfloat2`	4	2
`packed_bfloat3`	6	2
`packed_bfloat4`	8	2
`packed_float2`	8	4
`packed_float3`	12	4

Type	Size (in bytes)	Alignment (in bytes)
`packed_float4`	16	4
`packed_long2`	16	8
`packed_long3`	24	8
`packed_long4`	32	8

Packed vector data types are typically used as a data storage format. Metal supports the assignment, arithmetic, logical, relational, and copy constructor operators for packed vector data types. Metal also supports loads and stores from a packed vector data type to an aligned vector data type and vice-versa.

Examples:

```
device float4 *buffer;
device packed_float4 *packed_buffer;
int i;
packed_float4 f ( buffer[i] );
pack_buffer[i] = buffer[i];

// An operator used to convert from packed_float4 to float4.
buffer[i] = float4( packed_buffer[i] );
```

You can use an array index to access components of a packed vector data type. In Metal 2.1 and later, you can use `.xyzw` or `.rgba` selection syntax to access components of a packed vector data type. The semantics and restrictions when swizzling for packed vector data type are the same as for vector types.

Example:

```
packed_float4 f;
f[0] = 1.0f; // OK
f.x = 1.0f; // OK, Metal 2.1 and later.
```

2.3 Matrix Data Types

Metal supports a subset of the matrix data types implemented by the system math library.

The supported matrix type names are:

- `halfnxm`
- `floatnxm`

Where `n` and `m` are numbers of columns and rows. `n` and `m` must be `2`, `3`, or `4`. A matrix of type `floatnxm` consists of `n floatm` vectors. Similarly, a matrix of type `halfnxm` consists of `n halfm` vectors.

Metal also supports `matrix<T,c, r>`, where `T` is a valid floating-point type, `c` is `2`, `3`, or `4`, and `r` is `2`, `3`, or `4`.

Table 2.5 lists the size and alignment of the matrix data types.

Table 2.5. Size and alignment of matrix data types

Type	Size (in bytes)	Alignment (in bytes)
`half2x2`	8	4
`half2x3`	16	8
`half2x4`	16	8
`half3x2`	12	4
`half3x3`	24	8
`half3x4`	24	8
`half4x2`	16	4
`half4x3`	32	8
`half4x4`	32	8
`float2x2`	16	8
`float2x3`	32	16
`float2x4`	32	16
`float3x2`	24	8
`float3x3`	48	16
`float3x4`	48	16
`float4x2`	32	8
`float4x3`	64	16
`float4x4`	64	16

2.3.1 Accessing Matrix Components

You can use the array subscripting syntax to access the components of a matrix. Applying a single subscript to a matrix treats the matrix as an array of column vectors. Two subscripts select a column and then a row. The top column is column 0. A second subscript then operates on the resulting vector, as defined earlier for vectors.

```
float4x4 m;


// This sets the 2nd column to all 2.0.
m[1] = float4(2.0f);
// This sets the 1st element of the 1st column to 1.0.
m[0][0] = 1.0f;
// This sets the 4th element of the 3rd column to 3.0.
m[2][3] = 3.0f;
```

Access `floatnxm` and `halfnxm` matrices as an array of `n floatm` or `n halfm` entries.

Accessing a component outside the bounds of a matrix with a nonconstant expression results in undefined behavior. Accessing a matrix component that is outside the bounds of the matrix with a constant expression generates a compile-time error.

2.3.2 Matrix Constructors

Use constructors to create matrices from a set of scalars, vectors, or matrices. The parameter signature determines how to construct and initialize a matrix. For example, if you initialize a matrix with only a single scalar parameter, the result is a matrix that contains that scalar for all components of the matrix's diagonal, with the remaining components initialized to `0.0`. For example, a call to:

```
float4x4(fval);
```

Where `fval` is a scalar floating-point value constructs a matrix with these initial contents:

```
fval  0.0   0.0   0.0
0.0   fval  0.0   0.0
0.0   0.0   fval  0.0
0.0   0.0   0.0   fval
```

You can also construct a matrix from another matrix that has the same number of rows and columns. For example:

```
float3x4(float3x4);
float3x4(half3x4);
```

Metal constructs and consumes matrix components in column-major order. The matrix constructor needs to have just enough specified values in its arguments to initialize every component in the constructed matrix object. Providing more arguments than necessary results in an error. Under-initializing a matrix constructor results in a compile-time error.

You can also construct a matrix of type `T` with `n` columns and `m` rows from `n` vectors of type `T` with `m` components. The following examples are legal constructors:

```
float2x2(float2, float2);
float3x3(float3, float3, float3);
float3x2(float2, float2, float2);
```

In Metal 2 and later, a matrix of type `T` with `n` columns and `m` rows can also be constructed from `n` * `m` scalars of type `T`. The following examples are legal constructors:

```
float2x2(float, float, float, float);
float3x2(float, float, float, float, float, float);
```

The following are examples of matrix constructors that Metal doesn't support. You can't construct a matrix from combinations of vectors and scalars.

```
// Not supported.
float2x3(float2 a, float b, float2 c, float d);
```

2.4 SIMD-group Matrix Data Types

All OS: Metal 2.3 and later support SIMD-group matrix types.

Metal supports a matrix type `simdgroup_matrix<T,Cols,Rows>` defined in `<metal_simdgroup_matrix>`. Operations on SIMD-group matrices are executed cooperatively by threads in the SIMD-group. Therefore, all operations must be executed only under uniform control-flow within the SIMD-group or the behavior is undefined.

Metal supports the following SIMD-group matrix type names, where `T` is `half`, `bfloat` (in Metal 3.1 and later) or `float` and `Cols` and `Rows` are `8`:

- `simdgroup_half8x8`
- `simdgroup_bfloat8x8` (Metal 3.1 and later)
- `simdgroup_float8x8`

The mapping of matrix elements to threads in the SIMD-group is unspecified. For a description of which functions Metal supports on SIMD-group matrices, see section 6.8

2.5 Alignment of Data Types

You can use the `alignas` alignment specifier to specify the alignment requirement of a type or an object. You may also apply the `alignas` specifier to the declaration of a variable or a data member of a structure or class. You may also apply it to the declaration of a structure, class, or enumeration type.

The Metal compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a graphics or kernel function declared to be a pointer to a data type, the Metal compiler assumes that the object referenced by the pointer is always appropriately aligned as required by the data type.

2.6 Atomic Data Types

Objects of atomic types are free from data races. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.

Metal supports `atomic<T>,` where `T` can be `int`, `int32_t`, `uint`, `uint32_t`, `bool`, or `ulong` for all OSes that support Metal 2.4 and later, or `T` can be `float` for all OSes that support Metal 3 and later.

Metal provides these type aliases for atomic types:

`atomic_int`	A type of alias of `atomic<int>` for OSes that support Metal 1 and later.
`atomic_uint`	A type of alias of `atomic<uint>` for OSes that support Metal 1 and later.
`atomic_bool`	A type of alias of `atomic<bool>` for OSes that support Metal 2.4 and later.
`atomic_ulong`	A type of alias of `atomic<ulong>` for OSes that support Metal 2.4 and later.
`atomic_float`	A type of alias of `atomic<float>` for OSes that support Metal 3 and later.

Metal atomic functions (as described in section 6.16) can only use Metal atomic data types. These atomic functions are a subset of the C++17 atomic and synchronization functions.

2.7 Pixel Data Types

iOS: Metal 2 and later support pixel data types.
macOS: Metal 2.3 and later support pixel data types.
iPadOS and visionOS: Metal always support pixel data types.

The Metal pixel data type is a templated type that describes the pixel format type and its corresponding ALU type. The header `<metal_pixel>` defines Metal pixel data. The ALU type represents the type returned by a load operation and the input type specified for a store operation. Pixel data types are generally available in all address spaces. (For more about address spaces, see section 4.)

Table 2.6 lists supported pixel data types in MSL, as well as their size and alignment.

Table 2.6. Metal pixel data types

Pixel data type	Supported values of T	Size (in bytes)	Alignment (in bytes)
`r8unorm<T>`	`half` or `float`	1	1
`r8snorm<T>`	`half` or `float`	1	1
`r16unorm<T>`	`float`	2	2
`r16snorm<T>`	`float`	2	2
`rg8unorm<T>`	`half2` or `float2`	2	1
`rg8snorm<T>`	`half2` or `float2`	2	1
`rg16unorm<T>`	`float2`	4	2
`rg16snorm<T>`	`float2`	4	2
`rgba8unorm<T>`	`half4` or `float4`	4	1
`srgba8unorm<T>`	`half4` or `float4`	4	1
`rgba8snorm<T>`	`half4` or `float4`	4	1
`rgba16unorm<T>`	`float4`	8	2
`rgba16snorm<T>`	`float4`	8	2
`rgb10a2<T>`	`half4` or `float4`	4	4
`rg11b10f<T>`	`half3` or `float3`	4	4
`rgb9e5<T>`	`half3` or `float3`	4	4

Only assignments and equality/inequality comparisons between the pixel data types and their corresponding ALU types are allowed. (The following examples show the `buffer(n)` attribute, which is explained in section 5.2.1.)

Example:

```
kernel void
my_kernel(device rgba8unorm<half4> *p [[buffer(0)]],
          uint gid [[thread_position_in_grid]], …)
{
     rgba8unorm<half4> x = p[index]; half4 val = p[gid];
     …
     p[gid] = val;
     p[index] = x;
}
```

Example:

```
struct Foo {
     rgba8unorm<half4> a;
};

kernel void
my_kernel(device Foo *p [[buffer(0)]],
          uint gid [[thread_position_in_grid]], …)
{
     half4 a = p[gid].a;
     …
     p[gid].a = a;
}
```

2.8 Buffers

MSL implements a buffer as a pointer to a built-in or user defined data type described in the `device`, `constant`, or `threadgroup` address space. (For more about these address space attributes, see sections 4.1, 4.2, and 4.4, respectively.)

Ordinary Metal buffers may contain:

- Basic types such as `float` and `int`
- Vector and matrix types
- Arrays of buffer types
- Structures of buffer types
- Unions of buffer types

Note: In Metal 2.3 and later, Metal supports buffers that contain `long` or `ulong` data types.

The example below shows buffers as arguments to a function. The first two arguments are buffers in the `device` address space. The third argument is a buffer in the `constant` address space.

```
vertex ColorInOut
phong_vertex(const device packed_float3* vertices [[buffer(0)]],
             const device packed_float3* normals [[buffer(1)]],
             constant AAPL::uniforms_t& uniforms [[buffer(2)]],
             unsigned int vid [[vertex_id]])
{
   ...
}
```

For more about the `buffer(n)` attribute used in the example, see section 5.2.1.

For details about argument buffers, see section 2.13.

2.9 Textures

All OS: Metal 3.2 and later support `memory_coherence` for Apple silicon.

The texture data type is a handle to one-, two-, or three-dimensional texture data that corresponds to all or a portion of a single mipmap level of a texture.

```
enum class access { sample, read, write, read_write };
```

In Metal 3.2 and later, texture supports the optional memory coherence parameter (see section 4.8).

```
enum memory_coherence {
  memory_coherence_threadgroup,
  memory_coherence_device
};
```

The description below uses the Metal 3.2 template definition with the additional optional coherence parameter. Metal 3.1 and earlier drop that parameter. For example,

```
// Prior to Metal 3.2
texture1d<T, access a = access::sample>
```

versus:

```
// Metal 3.2 and later
texture1d<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>
```

The following templates define specific texture data types:

```
texture1d<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texture1d_array<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texture2d<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texture2d_array<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texture3d<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texturecube<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texturecube_array<T, access a = access::sample,
             memory_coherence c = memory_coherence_threadgroup>

texture2d_ms<T, access a = access::read,
```

```
            memory_coherence c = memory_coherence_threadgroup>
texture2d_ms_array<T, access a = access::read,
            memory_coherence c = memory_coherence_threadgroup>
```

To use `sample_compare` with a depth format, you need to declare one of the following texture types:

```
depth2d<T, access a = access::sample,
          memory_coherence c = memory_coherence_threadgroup>


depth2d_array<T, access a = access::sample,
            memory_coherence c = memory_coherence_threadgroup>


depthcube<T, access a = access::sample,
            memory_coherence c = memory_coherence_threadgroup>
depthcube_array<T, access a = access::sample,
            memory_coherence c = memory_coherence_threadgroup>
```

macOS supports `texture2d_ms_array` and `depth2d_ms_array` in Metal 2 and later. All other types supported in Metal 1 and later.

iOS supports all types except `texture2d_ms_array` and `depth2d_ms_array` in Metal 1 and later.

`T` specifies the color type of one of the components returned when reading from a texture or the color type of one of the components specified when writing to the texture. For texture types (except depth texture types), `T` can be `half`, `float`, `short`, `ushort`, `int`, or `uint`. For depth texture types, `T` must be `float`.

If `T` is `int` or `short`, the data associated with the texture must use a signed integer format. If `T` is `uint` or `ushort`, the data associated with the texture must use an unsigned integer format. If `T` is `half`, the data associated with the texture must either be a normalized (signed or unsigned integer) or half-precision format. If `T` is `float`, the data associated with the texture must either be a normalized (signed or unsigned integer), half or single-precision format.

These `access` attributes describe support for accessing a texture:

- `sample` — A graphics or kernel function can sample the texture object. `sample` implies the ability to read from a texture with and without a sampler.
- `read` — Without a sampler, a graphics or kernel function can only read the texture object.
- `write` — A graphics or kernel function can write to the texture object.
- `read_write` — A graphics or kernel function can read and write to the texture object.

All OS: Metal 1.2 and later support `read_write` access. Metal 1 and later support other access qualifiers.

Multisampled textures only support the `read` attribute. Depth textures only support the `sample` and `read` attributes. Sparse textures do not support `write` or `read_write` attributes.

The following example uses access qualifiers with texture object arguments:

```
void foo (texture2d<float> imgA [[texture(0)]],
          texture2d<float, access::read> imgB [[texture(1)]],
          texture2d<float, access::write> imgC [[texture(2)]])
{…}
```

(For a description of the `texture` attribute, see section 5.2.1.)

You can use a texture type as the variable type for any variables declared inside a function. The `access` attribute for variables of texture type declared inside a function must be `access::read` or `access:sample`. Declaring variables inside a function to be a texture type without using `access::read` or `access:sample` qualifiers causes a compilation error.

Examples:

```
void foo (texture2d<float> imgA [[texture(0)]],
          texture2d<float, access::read> imgB [[texture(1)]],
          texture2d<float, access::write> imgC [[texture(2)]])
{
    texture2d<float> x = imgA; // OK
    texture2d<float, access::read> y = imgB; // OK
    texture2d<float, access::write> z; // This is illegal.
    …
}
```

In Metal 3.2 and later, you can indicate whether texture operations are coherent across the device, meaning that texture operations are visible to other threads across thread groups if you synchronize them properly; for example:

```
constant texture2d<float, access::sample,
                   memory_coherence_device> gtex [[ texture(2)]];

constant texture2d<int, access::write,
                   memory_coherence::memory_coherence_device>
         gtex2 [[ texture(8)]];
```

See section 4.8 for more information about coherence.

2.9.1 Texture Buffers

All OS: Metal 2.1 and later support texture buffers.

A texture buffer is a texture type that can access a large 1D array of pixel data and perform dynamic type conversion between pixel formats on that data with optimized performance. Texture buffers handle type conversion more efficiently than other techniques, allowing access

to a larger element count, and handling out-of-bounds read access. Similar type conversion can be achieved without texture buffers by either:

- Reading the pixel data (just like any other array) from a texture object and performing the pixel transformation to the desired format.
- Wrapping a texture object around the data of a buffer object, then accessing the shared buffer data via the texture. This wrapping technique provides the pixel conversion, but requires an extra processing step, and the size of the texture is limited.

The following template defines the opaque type `texture_buffer`, which you can use like any texture:

```
texture_buffer<T, access a = access::read>
```

`access` can be one of `read`, `write`, or `read_write`.

`T` specifies the type of a component returned when reading from a texture buffer or the type of component specified when writing to a texture buffer. For a texture buffer, `T` can be one of `half`, `float`, `short`, `ushort`, `int`, or `uint`.

For a format without an alpha channel (such as R, RG, or RGB), an out-of-bounds read returns (`0, 0, 0, 1`). For a format with alpha (such as RGBA), an out-of-bounds read returns (`0, 0, 0, 0`). For some devices, an out-of-bounds read might have a performance penalty.

Metal ignores an out-of-bounds write.

A texture buffer can support more texture data than a generic 1D texture, which has is a maximum width of 16384. However, you cannot sample a texture buffer.

A texture buffer also converts data, delivering it in the requested texture format, regardless of the source's format. When creating a texture buffer, you can specify the format of the data in the buffer (for example, `RGBA8Unorm`), and later the shader function can read it as a converted type (such as `float4`). As a result, a single pipeline state object can access data stored in different pixel formats without recompilation.

A texture buffer, like a texture type, can be declared as the type of a local variable to a shader function. For information about arrays of texture buffers, see section 2.12.1. For more about texture buffer, see section 6.13.16.

2.10 Samplers

The `sampler` type identifies how to sample a texture. The Metal API allows you to create a sampler object and pass it in an argument to a graphics or kernel function. You can describe a sampler object in the program source instead of in the API. For these cases, you can only specify a subset of the sampler state: the addressing mode, filter mode, normalized coordinates, and comparison function.

Table 2.7 lists the supported sampler state enumerations and their associated values (and defaults). You can specify these states when initializing a sampler in Metal program source.

Table 2.7. Sampler state enumeration values

Enumeration	Valid values	Description
`coord`	`normalized` (default) `pixel`	When sampling from a texture, specifies whether the texture coordinates are normalized values.

`address`	`repeat`
	`mirrored_repeat`
	`clamp_to_edge` (default)
	`clamp_to_zero`
	`clamp_to_border`
		Sets the addressing mode for all texture coordinates.

`s_address`
`t_address`
`r_address`
	`repeat`
	`mirrored_repeat`
	`clamp_to_edge` (default)
	`clamp_to_zero`
	`clamp_to_border`
		Sets the addressing mode for individual texture coordinates.

`border_color`
macOS: Metal 1.2
iOS: Metal 2.3
iPadOS: Metal 2.3
visionOS: Always
	`transparent_black` (default)
	`opaque_black`
	`opaque_white`
		Specifies the border color to use with the `clamp_to_border` addressing mode.

`filter`	`nearest` (default) `linear`	Sets the magnification and minification filtering modes for texture sampling.
`mag_filter`	`nearest` (default) `linear`	Sets the magnification filtering mode for texture sampling.
`min_filter`	`nearest` (default) `linear`	Sets the minification filtering mode for texture sampling.

`mip_filter`	`none` (default)
	`nearest`
	`linear`
		Sets the mipmap filtering mode for texture sampling. If `none`, the texture is sampled as if it has a single mip level. All samples are read from level 0.

`compare_func`	`never` (default)
	`less`
	`less_equal`
	`greater`
	`greater_equal`
	`equal`
	`not_equal`
	`always`
		Sets the comparison test used by the `sample_compare` and `gather_compare` texture functions.

Enumeration	Valid values	Description

`reduction`
All OS: Metal 2.3
	`weighted_average`
	`minimum`
	`maximum`
		Sets how to compute the filtered pixel value by computing the component-wise to be `weighted_average` (default), `minimum` or `maximum`.

`bias` All OS: Metal 4.0	`float value`	The level-of-detail (LOD) bias to apply before sampling. See the Metal Feature Set Tables for more information about which GPU families support sampler bias.

macOS: Metal 1.2 and later support `clamp_to_border` address mode and `border_color`.

iOS and iPadOS: Metal 2.3 and later support `clamp_to_border` address mode or `border_color`.

visionOS: Metal supports `clamp_to_border` address mode or `border_color`.

With `clamp_to_border`, sampling outside a texture only uses the border color for the texture coordinate (and does not use any colors at the edge of the texture). If the address mode is `clamp_to_border`, then `border_color` is valid.

`clamp_to_zero` is equivalent to `clamp_to_border` with a border color of `transparent_black` (`0.0, 0.0, 0.0`) with the alpha component value from the texture. If `clamp_to_zero` is the address mode for one or more texture coordinates, the other texture coordinates can use an address mode of `clamp_to_border` if the border color is `transparent_black`. Otherwise, Metal doesn't define the behavior.

If `coord` is set to `pixel`, the `min_filter` and `mag_filter` values must be the same, the `mip_filter` value must be `none`, and the `address` modes must be either `clamp_to_zero`, `clamp_to_border`, or `clamp_to_edge`.

In addition to the enumeration types, you can also specify the maximum anisotropic filtering and an level-of-detail (LOD) range for a sampler:

```
max_anisotropy(int value)
lod_clamp(float min, float max)
```

The following Metal program source illustrates several ways to declare samplers. (The `sampler(n)` attribute that appears in the code below is explained in section 5.2.1.) Note that

samplers or constant buffers declared in program source do not need these attribute qualifiers. You must use `constexpr` to declare samplers that you initialize in MSL source:

```
constexpr sampler s(coord::pixel,
                    address::clamp_to_zero,
                    filter::linear);

constexpr sampler a(coord::normalized);

constexpr sampler b(address::repeat);

constexpr sampler s(address::clamp_to_zero,
                    filter::linear,
                    compare_func::less);

constexpr sampler s(address::clamp_to_zero,
                    filter::linear,
                    compare_func::less,
                    max_anisotropy(10),
                    lod_clamp(0.0f, MAXFLOAT));

kernel void
my_kernel(device float4 *p [[buffer(0)]],
          texture2d<float> img [[texture(0)]],
          sampler smp [[sampler(3)]],
          …)
{
      …
}
```

2.11 Imageblocks

iOS: Metal 2 and later support imageblocks.
macOS: Metal 2.3 and later support imageblocks.
iPadOS and visionOS: Metal supports imageblocks.

An imageblock is a 2D data structure (represented by width, height, and number of samples) allocated in threadgroup memory that is an efficient mechanism for processing 2D image data. Each element of the structure can be a scalar or vector integer or floating-point data type, pixel data types (specified in Table 2.6 in section 2.7), an array of these types, or structures built using these types. The data layout of the imageblock is opaque. You can use an (x, y) coordinate and optionally the sample index to access the elements in the imageblock. The elements in the imageblock associated with a specific (x, y) are the per-thread imageblock data or just the imageblock data.

Section 5.6 details imageblock attributes, including the `[[imageblock_data(type)]]` attribute. Section 6.14 lists the built-in functions for imageblocks.

Imageblocks are only used with fragment and kernel functions. Sections 5.6.3 and 5.6.4 describe how to access an imageblock in a fragment or kernel function, respectively.

For fragment functions, you can access only the fragment's imageblock data (identified by the fragment's pixel position in the tile). Use the tile size to derive the imageblock dimensions.

For kernel functions, all threads in the threadgroup can access the imageblock. You typically derive the imageblock dimensions from the threadgroup size, before you specify the imageblock dimensions.

An imageblock *slice* refers to a region in the imageblock that describes the values of a given element in the imageblock data structure for all pixel locations or threads in the imageblock. The storage type of the imageblock slice must be compatible with the texture format of the target texture, as listed in Table 2.8.

Table 2.8. Imageblock slices and compatible target texture formats

Pixel storage type	Compatible texture formats
`float`, `half`	`R32Float`, `R16Float`, `R8Unorm`, `R8Snorm`, `R16Unorm`, `R16Snorm`
`float2`, `half2`	`RG32Float`, `RG16Float`, `RG8Unorm`, `RG8Snorm`, `RG16Unorm`, `RG16Snorm`
`float4`, `half4`	`RGBA32Float`, `RGBA16Float`, `RGBA8Unorm`, `RGBA8Snorm`, `RGBA16Unorm`, `RGBA16Snorm`, `RGB10A2Unorm`, `RG11B10Float`, `RGB9E5Float`
`int`, `short`	`R32Sint`, `R16Sint`, `R8Sint`
`int2`, `short2`	`RG32Sint`, `RG16Sint`, `RG8Sint`
`int4`, `short4`	`RGBA32Sint`, `RGBA16Sint`, `RGBA8Sint`
`uint`, `ushort`	`R32Uint`, `R16Uint`, `R8Uint`
`uint2`, `ushort2`	`RG32Uint`, `RG16Uint`, `RG8Uint`
`uint4`, `ushort4`	`RGBA32Uint`, `RGBA16Uint`, `RGBA8Uint`
`r8unorm<T>`	`A8Unorm`, `R8Unorm`
`r8snorm<T>`	`R8Snorm`
`r16unorm<T>`	`R16Unorm`
`r16snorm<T>`	`R16Snorm`
`rg8unorm<T>`	`RG8Unorm`
`rg8snorm<T>`	`RG8Snorm`
`rg16unorm<T>`	`RG16Unorm`

Pixel storage type	Compatible texture formats
`rg16snorm<T>`	`RG16Snorm`
`rgba8unorm<T>`	`RGBA8Unorm`, `BGRA8Unorm`
`srgba8unorm<T>`	`RGBA8Unorm_sRGB`, `BGRA8Unorm_sRGB`
`rgba8snorm<T>`	`RGBA8Snorm`, `BGRA8Unorm`
`rgba16unorm<T>`	`RGBA16Unorm`
`rgba16snorm<T>`	`RGBA16Snorm`
`rgb10a2<T>`	`RGB10A2Unorm`
`rg11b10f<T>`	`RG11B10Float`
`rgb9e5<T>`	`RGB9E5Float`

2.12 Aggregate Types

Metal supports several aggregate types: arrays, structures, classes, and unions.

Do not specify a structure member with an address space attribute, unless the member is a pointer type. All members of an aggregate type must belong to the same address space. (For more about address spaces, see section 4.)

2.12.1 Arrays of Textures, Texture Buffers, and Samplers

iOS: Metal 1.2 and later support arrays of textures. Metal 2 and later support arrays of samplers. Metal 2.1 and later support arrays of texture buffers.
macOS: Metal 2 and later support arrays of textures and arrays of samplers. Metal 2.1 and later support arrays of texture buffers.
iPadOS and visionOS: Metal supports arrays of textures, samplers, and texture buffers.

Declare an array of textures as either:

```
array<typename T, size_t N>
```

```
const array<typename T, size_t N>
```

`typename` is a texture type you declare with the `access::read` or `access::sample` attribute. Metal 2 and later support an array of writeable textures (`access::write`) in macOS. Metal 2.2 and later, with Apple GPU Family 5 and later, support it in iOS. (For more about texture types, see section 2.9.)

Construct an array of texture buffers (see section 2.9.1) with the `access::read` qualifier using:

```
array<texture_buffer<T>, size t N>
```

Declare an array of samplers as either:

```
array<sampler, size_t N>

const array<sampler, size_t N>
```

You can pass an array of textures or an array of samplers as an argument to a function (graphics, kernel, or user function) or declare an array of textures or samples as a local variable inside a function. You can also declare an array of samplers in program scope. Unless used in an argument buffer (see section 2.13), you cannot declare an `array<T, N>` type (an array of textures, texture buffers, or samplers) in a structure.

MSL also adds support for `array_ref<T>`. An `array_ref<T>` represents an immutable array of `size()` elements of type `T`. `T` must be a sampler type or a supported texture type, including texture buffers. The storage for the array is not owned by the `array_ref<T>` object. Implicit conversions are provided from types with contiguous iterators like `metal::array`. A common use for `array_ref<T>` is to pass an array of textures as an argument to functions so they can accept a variety of array types.

The `array_ref<T>` type cannot be passed as an argument to graphics and kernel functions. However, the `array_ref<T>` type can be passed as an argument to user functions. The `array_ref<T>` type cannot be declared as local variables inside functions.

The member functions listed in sections 2.12.1.1 to 2.12.1.3 are available for the array of textures, array of samplers, and the `array_ref<T>` types.

2.12.1.1 Array Element Access with its Operator

Elements of an array of textures, texture buffers, or samplers can be accessed using the `[]` operator:

```
reference operator[] (size_t pos);
```

Elements of an array of textures, texture buffers, or samplers, or a templated type `array_ref<T>` can be accessed using the following variant of the `[]` operator:

```
constexpr const_reference operator[] (size_t pos) const;
```

2.12.1.2 Array Capacity

`size()` returns the number of elements in an array of textures, texture buffers, or samplers:

```
constexpr size_t size();

constexpr size_t size() const;
```

Example:

```
kernel void
my_kernel(const array<texture2d<float>, 10> src [[texture(0)]],
          texture2d<float, access::write> dst [[texture(10)]],
          …)
```

```
{
    for (int i=0; i<src.size(); i++)
    {
        if (is_null_texture(src[i]))
            break;
        process_image(src[i], dst);
    }
}
```

2.12.1.3 Constructors for Templated Arrays

```
constexpr array_ref();
constexpr array_ref(const array_ref &);
array_ref & operator=(const array_ref &);
constexpr array_ref(const T * array, size_t length);

template<size_t N>
constexpr array_ref(const T(&a)[N]);

template<typename T>
constexpr array_ref<T> make_array_ref(const T * array, size_t
length)

template<typename T, size_t N>
constexpr array_ref<T> make_array_ref(const T(&a)[N])
```

Examples of constructing arrays:

```
float4 foo(array_ref<texture2d<float>> src)
{
    float4 clr(0.0f);
    for (int i=0; i<src.size; i++)
    {
        clr += process_texture(src[i]);
    }
    return clr;
}

kernel void
my_kernel_A(const array<texture2d<float>, 10> src [[texture(0)]],
          texture2d<float, access::write> dst [[texture(10)]],
          …)
{
    float4 clr = foo(src);
    …
}

kernel void
my_kernel_B(const array<texture2d<float>, 20> src [[texture(0)]],
          texture2d<float, access::write> dst [[texture(10)]],
```

```
            …)
{
    float4 clr = foo(src);
    …
}
```

Below is an example of an array of samplers declared in program scope:

```
constexpr array<sampler, 2> = { sampler(address::clamp_to_zero),
                                sampler(coord::pixel) };
```

2.12.2 Structures of Buffers, Textures, and Samplers

Arguments to a graphics, kernel, visible, or user function can be a structure or a nested structure with members that are buffers, textures, or samplers only. You must pass such a structure by value. Each member of such a structure passed as the argument type to a graphics or kernel function can have an attribute to specify its location (as described in section 5.2.1).

Example of a structure passed as an argument:

```
struct Foo {
     texture2d<float>  a [[texture(0)]];
     depth2d<float>    b [[texture(1)]];
};

[[kernel]] void
my_kernel(Foo f)
{…}
```

You can also nest structures, as shown in the following example:

```
struct Foo {
     texture2d<float>  a [[texture(0)]];
     depth2d<float>    b [[texture(1)]];
};

struct Bar {
     Foo f;
     sampler s [[sampler(0)]];
};


[[kernel]] void
my_kernel(Bar b)
{…}
```

Below is an example of invalid use-cases that shall result in a compilation error:

```
struct MyResources {
     texture2d<float>  a [[texture(0)]];
     depth2d<float>    b [[texture(1)]];
     int c;
};

[[kernel]] void
my_kernel(MyResources r) // This is an illegal use.
{…}
```

2.13 Argument Buffers

All OS: Metal 2 and later support argument buffers.

Argument buffers extend the basic buffer types to include pointers (buffers), textures, texture buffers, and samplers. However, argument buffers cannot contain unions. The following example specifies an argument buffer structure called `Foo` for a function:

```
struct Foo {
     texture2d<float, access::write> a;
     depth2d<float> b;
     sampler c;
     texture2d<float> d;
     device float4* e;
     texture2d<float> f;
     texture_buffer<float> g;
     int h;
};
kernel void
my_kernel(constant Foo & f [[buffer(0)]])
{…}
```

Arrays of textures and samplers can be declared using the existing `array<T, N>` templated type. Arrays of all other legal buffer types can also be declared using C-style array syntax.

Members of argument buffers can be assigned a generic `[[id(n)]]` attribute, where `n` is a 32-bit unsigned integer that can be used to identify the buffer element from the Metal API. Argument buffers can be distinguished from regular buffers if they contain buffers, textures, samplers, or any element with the `[[id]]` attribute.

The same index may not be assigned to more than one member of an argument buffer. Manually assigned indices do not need to be contiguous, but they must be monotonically increasing. In the following example, index `0` is automatically assigned to `foo1`. The `[[id(n)]]` attribute specifies the index offsets for the `t1` and `t2` structure members. Since `foo2` has no specified index, it is automatically assigned the next index, `4`, which is determined by adding 1 to the maximum ID used by the previous structure member.

```
struct Foo {
```

```
    texture2d<float> t1 [[id(1)]];
    texture2d<float> t2 [[id(3)]];
};
struct Bar {
    Foo foo1; // foo1 assigned idx 0, t1 and t2 assigned idx 1 and 3
    Foo foo2; // foo2 assigned idx 4, t1 and t2 assigned idx 5 and 7
};
```

If you omit the `[[id]]` attribute, Metal automatically assigns an ID according to the following rules:

1. Metal assigns IDs to structure members in order, by adding 1 to the maximum ID of the previous structure member. In the example below, the indices are not provided, so indices `0` and `1` are automatically assigned.

   ```
   struct MaterialTexture {
       texture2d<float> tex; // Assigned index 0
       float4 uvScaleOffset; // Assigned index 1
   };
   ```

2. Metal assigns IDs to array elements in order, by adding 1 to the maximum ID of the previous array element. In the example below, indices 1-3 are automatically assigned to the three array elements of `texs1`. Indices 4-5 are automatically assigned to the fields in `materials[0]`, indices 6-7 to `materials[1]`, and indices 8-9 to `materials[2]`. The `[[id(20)]]` attribute starts by assigning index 20 to constants.

   ```
   struct Material {
       float4 diffuse; // Assigned index 0
       array<texture2d<float>, 3> texs1; // Assigned indices 1–3
       MaterialTexture materials[3]; // Assigned indices 4–9
       int constants [[id(20)]] [4]; // Assigned indices 20–23
   };
   ```

3. If a structure member or array element `E` is itself a structure or array, Metal assigns indices to its structure members or array elements according to rules `1` and `2` recursively, starting from the ID assigned to `E`. In the following example, index `4` is explicitly provided for the nested structure called `normal`, so its elements (previously defined as `tex` and `uvScaleOffset`) are assigned IDs `4` and `5`, respectively. The elements of the nested structure called `specular` are assigned IDs `6` and `7` by adding one to the maximum ID (`5`) used by the previous member.

   ```
   struct Material {
       MaterialTexture diffuse; // Assigned indices 0, 1
       MaterialTexture normal [[id(4)]]; // Assigned indices 4, 5
       MaterialTexture specular; // Assigned indices 6, 7
   }
   ```

4. Metal assigns IDs to top-level argument buffer arguments starting from `0`, according to the previous three rules.

2.13.1 Tier 2 Hardware Support for Argument Buffers

With Tier 2 hardware, argument buffers have the following additional capabilities that are not available with Tier 1 hardware.

You can access argument buffers through pointer indexing. This syntax shown below refers to an array of consecutive, independently encoded argument buffers:

```
kernel void
kern(constant Resources *resArray [[buffer(0)]])
{
     constant Resources &resources = resArray[3];
}

struct TStruct {
  texture2d<float> tex;
};
kernel void
kern(constant TStruct *textures [[buffer(0)]]);
```

To support GPU driven pipelines and indirect draw calls and dispatches, you can copy resources between structures and arrays within a function, as shown below:

```
kernel void
copy(constant Foo & src [[buffer(0)]],
     device Foo & dst [[buffer(1)]])
{
     dst.a = src.d;
     …
}
```

Samplers cannot be copied from the thread address space to the device address space. As a result, samplers can only be copied into an argument buffer directly from another argument buffer. The example below shows both legal and illegal copying:

```
struct Resources {
     sampler sam;
};
kernel void
copy(device Resources *src,
     device Resources *dst,
     sampler sam1)
{
     constexpr sampler sam2;
     dst->sam = src->sam; // Legal: device -> device
     dst->sam = sam1; // Illegal: thread -> device
     dst->sam = sam2; // Illegal: thread -> device
}
```

Argument buffers can contain pointers to other argument buffers:

```
struct Textures {
     texture2d<float> diffuse;
     texture2d<float> specular;
};
struct Material {
     device Textures *textures;
};
fragment float4
fragFunc(device Material & material);
```

2.14 Uniform Type

All OS: Metal 2 and later support uniform types.

2.14.1 The Need for a Uniform Type

In the following function example, the variable `i` is used to index into an array of textures given by `texInput`. The variable `i` is nonuniform; that is, it can have a different value for threads executing the graphics or kernel function for a draw or dispatch call, as shown in the example below. Therefore, the texture sampling hardware must handle a sample request that can refer to different textures for threads executing the graphics or kernel function for a draw or dispatch call.

```
kernel void
my_kernel(array<texture2d<float>, 10> texInput,
          array<texture2d<float>, 10> texOutput,
          sampler s,
          …,
          uint2 gid [[thread_position_in_grid]])
{
     int i = …;
     float4 color = texInput[i].sample(s, float2(gid));
     …;
     texOutput[i].write(color, float2(gid));
}
```

If the variable `i` has the same value for all threads (is uniform) executing the graphics or kernel function of a draw or dispatch call and if this information was communicated to the hardware, then the texture sampling hardware can apply appropriate optimizations. A similar argument can be made for texture writes, where a variable computed at runtime is used as an index into an array of textures or to index into one or more buffers.

To indicate that this variable is uniform for all threads executing the graphics or kernel function of a draw or dispatch call, MSL adds a new template class called uniform (available in the header `metal_uniform`) that can be for declaring variables inside a graphics or kernel

function. This template class can only be instantiated with arithmetic types (such as Boolean, integer, and floating-point) and vector types.

The code below is a modified version of the previous example, where the variable `i` is declared as a `uniform` type:

```
kernel void
my_kernel(array<texture2d<float>, 10> texInput,
          array<texture2d<float>, 10> texOutput,
          sampler s,
          …,
          uint2 gid [[thread_position_in_grid]])
{
     uniform<int> i = …;
     float4 color = texInput[i].sample(s, float2(gid));
     …;
     texOutput[i].write(color, float2(gid));
}
```

2.14.2 Behavior of the Uniform Type

If a variable is of the `uniform` type, and the variable does not have the same value for all threads executing the kernel or graphics function, then the behavior is undefined.

Uniform variables implicitly type convert to nonuniform types. Assigning the result of an expression computed using uniform variables to a uniform variable is legal but assigning a nonuniform variable to a uniform variable results in a compile-time error. In the following example, the multiplication legally converts the uniform variable `x` into nonuniform product `z`. However, assigning the nonuniform variable `z` to the uniform variable `b` results in a compile-time error.

```
uniform<int> x = …;
int y = …;
int z = x*y; // Here, x converts to a nonuniform for a multiply.
uniform<int> b = z; // Illegal; causes a compile-time error.
```

To declare an array of uniform elements:

```
uniform<float> bar[10]; // Elements stored in bar array are uniform.
```

The `uniform` type is legal for both parameters and the return type of a function. For example:

```
uniform<int> foo(…); // foo returns a uniform integer value.
int bar(uniform<int> a, …);
```

It is legal to declare a pointer to a uniform type, but not legal to declare a uniform pointer. For example:

```
device uniform<int> *ptr; // Values pointed to by ptr are uniform.
uniform<device int *> ptr; // Illegal; causes a compile-time error.
```

The results of expressions that combine uniform with nonuniform variables are nonuniform. If the nonuniform result is assigned to a uniform variable, as in the example below, the behavior is undefined. (The front-end might generate a compile-time error, but it is not guaranteed to do so.)

```
uniform<int> i = …;
int j = …;
if (i < j) { // Nonuniform result for expression (i < j).
     …
     i++;  // Causes a compile-time error, undefined behavior.
}
```

The following example is similar:

```
bool p = … // Nonuniform condition.
uniform<int> a = …, b = …;
uniform<int> c = p ? a : b; // Causes a compile-time error,
                            // undefined behavior.
```

2.14.3 Uniform Control Flow

When a control flow conditional test is based on a uniform quantity, all program instances follow the same path at that conditional test in a function. Code for control flow based on uniform quantities should be more efficient than code for control flow based on nonuniform quantities.

2.15 Visible Function Table

All OS: Metal 2.3 and later support visible function table.

Defined in the header `<metal_visible_function_table>`, you use the `visible_function_table` type to represent a table of function pointers to visible functions (see section 5.1.4) that the system stores in device memory. In Metal 2.3 and later, you can use it in a compute (`kernel`) function. In Metal 2.4 and later, you can use it in fragment, vertex, and tile functions. It is an opaque type, and you can't modify the content of the table from the GPU. You can use a `visible_function_table` type in an argument buffer or directly pass it to a qualified function using a `buffer` binding point.

To declare a `visible_function_table` type with a template parameter `T` where

`T` is the signature of the function stored in the table, use the following template function.

```
visible_function_table<typename T>
```

The following example shows how to declare a table that is compatible with a function whose definition is "[[visible]] int func(float f)":

```
visible_function_table<int(float)> functions;
```

To get a visible function pointer from the table, use the [] operator:

```
using fnptr = T (*)(…) [[visible]]
fnptr operator[](uint index) const;
```

size() returns the number of function pointer entries in the table:

```
uint size() const
```

empty() returns true if the table is empty:

```
bool empty() const
```

The following function can be used to determine if a table is a null visible_function_table. A null visible_function_table is a table that is not pointing to anything.

```
bool is_null_visible_function_table(visible_function_table<T>);
```

The following example shows how the table can be passed in a buffer:

```
using TFuncSig = void(float, int);
kernel void F(uint tid [[thread_position_in_grid]],
              device float* buf [[buffer(0)]],
              visible_function_table<TFuncSig> table [[buffer(1)]])
{
    uint tsize = table.size();
    table[tid % tsize](buf[tid], tid);
}
```

2.16 Function Groups Attribute

All OS: Metal 2.3 and later support [[function_groups]].

The optional [[function_groups]] attribute can be used to indicate the possible groups of functions being called from an indirect call through a function pointer or visible_function_table. This is a compiler hint to enable the compiler to optimize the call site. The groups of functions are specified as string literal arguments of the attribute. This attribute can be applied in three different contexts:

- Variable declarations with an initializer expression — It affects all indirect call expressions in the initializer expressions.
- Expression statements — It affects all the indirect call expressions of the given expression.

- Return statements — It affects all the indirect call expressions of the return value expression.

The following examples show how `[[function_groups]]` can be used:

```
float h(visible_function_table<float(float)> table,
        float (*fnptr[3])(float))
{
  // An indirect call to table[0] is restricted to “group1”.
  [[function_groups("group1")]] float x = table[0](1.0f);

  // An indirect call to `fnptr[0]` can call any function.
  x += fnptr[0](2.0f);

  // An indirect call to `fnptr[1]` is restricted to
  // "group2"+"group3".
  [[function_groups("group2", "group3")]] return x + fnptr[1](3.0f);
}
```

2.17 Ray-Tracing Types

All OS: Metal 2.3.and later support ray-tracing types.

The header `<metal_raytracing>` defines these types in the namespace `metal::raytracing`. In Metal 2.3 and later, these types are only supported in a compute function (kernel functions) except where noted below. In Metal 2.4 and later, they are also supported in vertex, fragment, and tile functions. In Metal 3.1 and later, ray tracing supports curves and multilevel instancing.

2.17.1 Ray-Tracing Intersection Tags

All OS: Metal 2.3.and later support ray-tracing intersection tags.

The header `<metal_raytracing>` defines `intersection_tags` in the namespace `metal::raytracing`. They are listed in Table 2.9 and are used in ray tracing when defining:

- intersection functions ([[`intersection`]] section 5.1.6)
- intersection function tables (`intersection_function_table` section 2.17.3)
- intersection results (`intersection_result` section 2.17.4)
- intersector types and associated functions (`intersector` section 6.19.2)
- acceleration structure types (`acceleration_structure` section 2.17.7 and 6.19.1)
- intersection queries (`intersection_query` section 6.19.5).

The tags are used to configure the ray tracing process and control the behavior and semantics of the different types and tables. The tags identify the type of accelerator structure being intersected, the built-in parameters available for intersection functions, the type of intersection

function in an intersection function table, the methods available on intersector type or intersection query object, and the data returned in an intersection result type.

The `intersection_tags` must match in tag type and order between related uses of `intersection_function_table`, `intersection_result` , `intersector`, and `intersection_query`, or the compiler will generate an error. The acceleration structure type being intersected must match the ordering of instancing, `primitive_motion`, and `instance_motion` tags if they are present on the other ray tracing types used to intersect the acceleration structure. When calling `intersection` functions in an `intersection` function table, you need to ensure they use the same ordered set of tags, or else the result is undefined.

Table 2.9. Intersection tags

Intersection tag	Description
`instancing`	This tag indicates intersection functions can read the built-in `instance_id` and/or `user_instance_id` as described in section 5.2.3.7, and the acceleration structure is an instance acceleration structure. The `intersector<intersection_tags...>::intersect()` function and `intersection_query<intersection_tags...>` assume that the acceleration structure needs to be an `instance_acceleration_structure` and it returns the `instance_id` value.
`triangle_data`	This tag indicates triangle intersection functions can read input parameters with `barycentric_coord` or `front_facing` attribute as described in section 5.2.3.7. This tag cannot be used in defining an acceleration structure. The `intersector<intersection_tags...>::intersect()` function and `intersection_query<intersection_tags...>` returns the `triangle_barycentric_coord` and `triangle_front_facing` values.

Intersection tag	Description
`world_space_data`	This tag indicates intersection functions declared with this tag can query `world_space_origin`, `world_space_direction`, `object_to_world_transform`, and `world_to_object_transform` as described in section 5.2.3.7. This tag cannot be used in defining an acceleration structure or `intersection_query`. It enables support for world space data in `intersector` and `intersection_function_table`.
`primitive_motion` All OS: Metal 2.4 and later	This tag enables support for primitive level motion in `intersector`, `intersection_function_table`, and acceleration structures.
`instance_motion` All OS: Metal 2.4 and later	This tag enables support for instance level motion in `intersector`, `intersection_function_table`, and acceleration structure.
`extended_limits` All OS: Metal 2.4 and later	This tag indicates acceleration structures passed to intersection functions are built with extended limits for the number of primitives, number of geometries, number of instances, and increases the number of bits used for visibility masks. This tag cannot be used in defining an acceleration structure.
`curve_data` All OS: Metal 3.1and later	This tag makes the `curve_parameter` of the curve intersection point available as a field of `intersection_result` object from methods of the `intersection_query` objects, and as input parameter to intersection functions as described in section 5.2.3.7.
`max_levels<Count>` All OS: Metal 3.1 and later	This tag enables support for multilevel instancing in `intersector`, `intersection_query` and `intersection_function_table`. It cannot be used in acceleration structures. `Count` is a template parameter that determines the maximum number of acceleration structure levels that can be traversed. It

Intersection tag	Description
	must be between [2, 16] for `intersection_query`. It must be [2, 32] for `intersector`. For `intersection_function_table`, it needs to match it use with `intersection_query` or `intersector`.
`intersection_function_buffer` All OS: Metal 4 and later	This tag signals that this intersection function is available for use in an intersection function buffer.
`user_data` All OS: Metal 4 and later	This tag makes the "user data" pointer available as a parameter marked by `user_data_buffer` to the function, which is available to pass resources (or any other data) to the intersection function intended for use in an intersection function buffer.

In Metal 2.3 and later, the following are valid combinations of intersection tags:

- no `tags`
- `triangle_data`
- `instancing`
- `instancing, triangle_data`
- `instancing, world_space_data`
- `instancing, triangle_data, world_space_data`

Metal 2.4 and later add the following additional valid combinations:

- `primitive_motion`
- `triangle_data, primitive_motion`
- `instancing, primitive_motion`
- `instancing, triangle_data, primitive_motion`
- `instancing, world_space_data, primitive_motion`
- `instancing, triangle_data, world_space_data, primitive_motion`
- `instance_motion`
- `instancing, instance_motion`
- `instancing, triangle_data, instance_motion`
- `instancing, world_space_data, instance_motion`
- `instancing, triangle_data, world_space_data, instance_motion`

- `instancing, primitive_motion, instance_motion`
- `instancing, triangle_data, primitive_motion, instance_motion`
- `instancing, world_space_data, primitive_motion, instance_motion`
- `instancing, triangle_data, world_space_data, primitive_motion, instance_motion`

The `extended_limits` tag may be added to all combinations listed above.

In Metal 3.1 and later, `curve_data` may be added to all combinations listed above. The intersection tag `max_levels<Count>` may be added to any combination above containing `instancing`.

In Metal 4 and later, `intersection_function_buffer` may be added to all combinations listed above. The tag `user_data` may only be used in combination with `intersection_function_buffer`.

2.17.2 Ray Type

The ray structure is a container for the properties of the ray required for an intersection.

```
struct ray
{
  ray(float3 origin = 0.0f, float3 direction = 0.0f,
      float min_distance = 0.0f, float max_distance = INFINITY);
  float3 origin;
  float3 direction;
  float min_distance;
  float max_distance;
};
```

The ray's `origin` and `direction` field are in world space. When a ray object is passed into a custom intersection or triangle intersection function, the `min_distance` and `max_distance` fields will be based on the current search interval: As candidate intersections are discovered, `max_distance` will decrease to match the newly narrowed search interval. Within intersection functions, the `origin` and `direction` are in object space.

A `ray` can be invalid. Examples of invalid rays include:

- INFs or NaNs in `origin` or `direction`
- `min_distance` == NaN or `max_distance` == NaN
- `min_distance` == INF (Note that `max_distance` may be positive INF).
- length(`ray.direction`) == 0.0
- `min_distance` > `max_distance`
- `min_distance` < 0.0 or `max_distance` < 0.0

The ray direction does not need to be normalized, although it does need to be nonzero.

2.17.3 Intersection Function Table

The `intersection_function_table<intersection_tags...>` structure type describes a table of custom intersection functions passed into the shader as defined from section 5.1.6. The intersection `tags` are defined from Table 2.9. The intersection tags on `intersection_function_table` type and the intersection functions must match. An example of such a declaration is:

```
  intersection_function_table<triangle_data, instancing>
intersectionFuncs;
```

Call the following function to check if the `intersection_function_table` is `null`:

```
bool
is_null_intersection_function_table(
   intersection_function_table< intersection_tags...>)
```

Call the following member function to check if the `intersection_function_table` is empty:

```
bool empty() const
```

Call the following member function to return the number of entries in `intersection_function_table`:

```
uint size() const
```

Metal 3 supports the following function: `get_buffer` and `get_visible_function_table`.

Call the following member function to return the buffer at index from the `intersection_function_table`, where `T` is a pointer or reference in the `device` or `constant` address space:

```
template<typename T>
  T get_buffer(uint index) const
```

Call the following member function to return the `visible_function_table<T>` at index from the `intersection_function_table`. `T` is the signature of the function stored in the table.

```
template <typename T> visible_function_table<T>
  get_visible_function_table(uint index) const;
```

Metal 3.1 supports the following functions: `set_buffer` and `set_visible_function_table`.

Call the following member functions to set the `device` or `constant` buffer object at the `index` position in the `intersection_function_table` entry.

```
void set_buffer(const device void *buf, uint index)
void set_buffer(constant void *buf, uint index)
```

Call the following member function to set the `visible_function_table` at the `index` position in the `intersection_function_table`, where `T` is the signature of the function stored in the table.

```
template<typename T>
  void set_visible_function_table(visible_function_table<T> vft,
                                  uint index)
```

2.17.4 Intersection Result Type

The results of an intersection return in an `intersection_result<intersection_tags...>` structure where `intersection_tags` are defined in Table 2.9. The return structure is defined as:

```
class intersection_type {
  none,
  triangle,
  bounding_box,
  curve // Available in Metal 3.1 and later.
};

template <typename...intersection_tags>
struct intersection_result
{
  intersection_type type;
  float             distance;
  uint              primitive_id;
  uint              geometry_id;

  const device void *primitive_data; // Available in Metal 3 and
                                     // later.

  // Available only if intersection_tags include instancing without
  // max_levels<Count>.
  uint              instance_id;
  uint              user_instance_id; // Available in Metal 2.4 and
                                      // later.

  // In Metal 3.1 and later, replace instance_id and
  // user_instance_id with an array if intersection_tags
  // include instancing and max_levels<Count>.
  uint instance_count; // The number of instances
                       // intersected by the ray.
  uint instance_id[Count - 1]; // The instance IDs of instances
                               // intersected by the ray.
```

```
  uint user_instance_id[Count - 1]; // The user instance IDs of
                                    // instances intersected by
                                    // the ray.

  // Available only if intersection_tags include triangle_data.
  float2            triangle_barycentric_coord;
  bool              triangle_front_facing;

  // In Metal 2.4 and later, the following is available only if
  // intersection_tags include world_space_data and instancing.
  float4x3          world_to_object_transform;
  float4x3          object_to_world_transform;

  // In Metal 3.1 and later, the following is available only if
  // intersection_tags include curve_data.
  float curve_parameter;

  // In Metal 4.1 and later, the following is available only if
  // intersection_tags include intersection_function_buffer.
  uint function_id;
};
```

If a `ray` is invalid, an `intersection::none` is returned.

The `distance` returned is in world space.

For vertex attributes `v0`, `v1`, and `v2`, the attribute value at the specified triangle barycentric point is:

```
  v1 * triangle_barycentric_coord.x +
  v2 * triangle_barycentric_coord.y +
  v0 * (1.0f - (triangle_barycentric_coord.x +
                triangle_barycentric_coord.y))
```

2.17.5 Intersection Result Reference Type

All OS: Metal 3.2 and later support `intersection_result_ref<intersection_tags…>` for Apple silicon. The Metal Feature Set Table lists the supported hardware.

In some use cases, it's possible to avoid a copy of `intersection_result` by using `intersection_result_ref<intersection_tags…>` whose lifetime is the duration of the lambda function that passes to the `intersector` intersect function (see section 6.19.2). The `intersection_result_ref<intersection_tags…>` structure where `intersection_tags` are defined in Table 2.9.

```
template <typename...intersection_tags>
struct intersection_result_ref {
public:
  intersection_type get_type() const;
  float get_distance() const;
  uint get_primitive_id() const;
  uint get_geometry_id() const;
  const device void *get_primitive_data() const;
```

```
  float3 get_ray_origin() const;
  float3 get_ray_direction() const;
  float get_ray_min_distance() const;

  // Available only if intersection_tags include instancing without.
  // max_levels<Count>.
  uint get_instance_id() const;
  uint get_user_instance_id() const;

  // Available only if intersection_tags include instancing with
  // max_levels<Count>.
  uint get_instance_count() const;
  uint get_instance_id(uint depth) const;
  uint get_user_instance_id(uint depth) const;

 // Available only if intersection_tags include triangle_data.
  float2 get_triangle_barycentric_coord() const;
  bool is_triangle_front_facing() const;

  // Available only if intersection_tags include curve_data.
  float get_curve_parameter() const;

  // Available only if intersection_tags include world_space_data
  // and instancing.
  float4x3 get_object_to_world_transform() const;
  float4x3 get_world_to_object_transform() const;

  // In Metal 4.1 and later, the following is available only if
  // intersection_tags include intersection_function_buffer.
  uint get_function_id() const;
};
```

2.17.6 Intersector Type

The `intersector<intersection_tags...>` structure type defines an object that controls the acceleration structure traversal and defines functions to intersect rays like `intersect()`. Use the `intersection_tags` (described in Table 2.9) when creating the intersector to specialize on which types of acceleration structure it operates on and which functions are available (see section 6.19.2). Intersection tags on the intersector type must match their associated intersection function (section 5.1.6), or the behavior is undefined.

```
// Create a default intersector.
intersector<> primitiveIntersector;

// Create a specialized intersector to support triangle and
// world space data.
```

```
intersector<triangle_data, instancing, world_space_data>
instanceInter;
```

The `intersector<intersection_tags...>` struct type provides a convenience type for the intersection result type defined in section 2.17.6:

```
intersector<intersection_tags...>::result
```

2.17.7 Acceleration Structure Type

All OS: Metal 2.3 and later support acceleration structure types.

All OS: Metal 2.4 and later support acceleration structure templatized types.

Metal 2.3 and later support two types of acceleration structure:

- `primitive_acceleration_structure`
- `instance_acceleration_structure`.

These are opaque objects that can be bound directly using buffer binding points or via argument buffers:

```
struct AccelerationStructs {
  primitive_acceleration_structure prim_accel;
  instance_acceleration_structure inst_accel;
  array<primitive_acceleration_structure, 2> prim_accel_array;
  array<instance_acceleration_structure, 2>  inst_accel_array;
};

[[kernel]]
void
intersectInstancesKernel(
   primitive_acceleration_structure prim_accel [[buffer(0)]],
   instance_acceleration_structure inst_accel [[buffer(1)]],
   device AccelerationStructs *accels [[buffer(3)]]) {…}
```

It is possible to create default initialized variables of such types, and the default value is the `null` value for the acceleration structures.

In Metal 2.4 and later, the acceleration structure is replaced with a templatized type `acceleration_structure<intersection_tags…>`. The template parameter `intersection_tags` can be empty or a combination of `instancing`, `primitive_motion`, or `instance_motion` as defined in Table 2.9. Intersection tags. For example, the following defines an instance acceleration structure that supports primitive motion:

```
acceleration_structure<instancing, primitive_motion> accel_struct;
```

The following combinations of tags can be used to declare a primitive acceleration structure:

- no `tags`
- `primitive_motion`

The following combinations of tags can be used to declare an instance acceleration structure:

- `instancing`
- `instancing`, `primitive_motion`
- `instancing`, `instance_motion`
- `instancing`, `primitive_motion`, `instance_motion`

To maintain backward compatibility, `primitive_acceleration_structure` is aliased to `acceleration_structure<>` and `instance_acceleration_structure` is aliased to `acceleration_structure<instancing>`.

As before, these are opaque objects that can be bound directly using buffer binding points or via argument buffers:

```
struct AccelerationMotionStructs {
  acceleration_structure<primitive_motion> prim_motion_accel;
  acceleration_structure<instancing,
                         instance_motion> inst_motion_accel;
  array<acceleration_structure<>, 2> prim_accel_array;
  array<acceleration_structure<instancing>, 2> inst_accel_array;
};

[[kernel]]
void
intersectMotionKernel(
  acceleration_structure<primitive_motion> prim   [[buffer(15)]],
  acceleration_structure<instancing,
                         primitive_motion, instance_motion>
                                           inst   [[buffer(16)]],
  device AccelerationMotionStructs        *accels [[buffer(17)]])
{…}
```

When binding these acceleration structures from the Metal API to the compute or graphic functions, the acceleration structure's type must match what is defined in the shader. For instance acceleration structures, you can bind instance acceleration structures without support for `primitive_motion` to a shader that expects instance acceleration structures with

`primitive_motion`. For example, a Metal buffer with an instance acceleration structure that can be passed to a shader with `acceleration_structure<instancing>` can also be given to a shader with `acceleration_structure<instancing, primitive_motion>`. This capability allows you to write one shader function that can handle either an acceleration structure with or without `primitive_motion` at the cost of the ray tracing runtime checking for primitive motion. To avoid this cost, write two functions where one uses an acceleration structure with `primitive_motion` and one without.

See section 6.19.1 for the functions to call if the acceleration structure is `null`.

2.17.8 Intersection Query Type

All OS: Metal 2.4 and later support intersection query types.

The `intersection_query<intersection_tags...>` type defines an object that enables users to fully control the ray tracing process and when to call custom intersection code. The intersection query object provides a set of functions to advance the query through an acceleration structure and query traversal information. Use the `intersection_tags` (defined in Table 2.9) when creating the `intersection_query<intersection_tags...>` type to specialize the type of acceleration structure and what functions are available (see section 6.19.5). It supports the following combinations of intersection tags:

- `no tags`
- `triangle_data`
- `instancing`
- `instancing, triangle_data`

Metal 3.1 supports the following additional combinations:

- `instancing, max_levels<Count>`
- `instancing, triangle_data, max_levels<Count>`

In Metal 3.1 and later, `curve_data` may be added to all combinations listed above.

The `intersection_query<intersection_tags...>` type has the following restrictions:

- it cannot be used for members of a structure/union
- it cannot be returned from a function
- it cannot be assigned to

These restrictions prevent the intersection query object from being copied.

2.18 Interpolant Type

All OS: Metal 2.3 and later support interpolant types.

The interpolant type `interpolant<T,P>` defined in `<metal_interpolate>` is a templatized type that encapsulates a fragment shader input for pull-model interpolation (section 6.12). Type parameters `T` and `P` represent the input's data type and perspective-correctness, respectively. Supported values for `T` are the scalar and vector floating-point types. Supported values of `P` are the types `interpolation::perspective` and `interpolation::no_perspective`.

You can declare a variable with the `interpolant<T,P>` type only in the following contexts:

- As a fragment shader input argument with `[[stage_in]]`. Such a declaration must match a corresponding vertex shader output argument of type `T` with the same name or `[[user(name)]]` attribute. The declaration can't have a sampling-and-interpolation attribute (section 5.4).
- As a local or temporary variable, which needs to be initialized as a copy of the above.

An `interpolant<T,P>` variable is not automatically convertible to a value of type `T`. Instead, retrieve a value by calling one of several interpolation methods (see section 6.12). The interpolation shall be perspective-correct if the value of `P` is `interpolation::perspective`.

2.19 Per-Vertex Values

All OS: Metal 4 and later support per -vertex values.

The vertex value type `vertex_value<T>` defined in `<metal_vertex_value>` is a templatized type to provide access to the per-vertex value (preraster per-vertex triangle attributes) in the fragment shader. You can declare a variable with `vertex_value<T>` as a fragment shader input argument where type `T` must match the corresponding type in the vertex output. See the Metal Feature Set Tables to determine which GPUs support this feature.

Call the following function to return the per-vertex value (non-interpolated value) at index `i`:

```
enum class vertex_index { first, second, third };
T get(vertex_index i);
```

The following example shows a shader that computes the interpolated value as a dot product between the non-interpolated values and the barycentric weights:

```
struct vertex_in {
  float3 position [[attribute(0)]];
  float4 color [[attribute(1)]];
};

struct vertex_out {
  float4 position [[position]];
```

```
  float4 color;
};

[[vertex]] vertex_out vert(vertex_in vert_in [[stage_in]]) { … }

struct fragment_in {
  float4 position [[position]];
  float3 barycentric_coords [[barycentric_coord,
                              center_no_perspective]];
  vertex_value<float4> color;
};

struct fragment_out {
  float4 color;
};


[[fragment]] fragment_out frag(fragment_in frag_in [[stage_in]]) {
  fragment_out frag_out;
  auto bc = frag_in.barycentric_coords;
  auto c1 = frag_in.color.get(vertex_index::first);
  auto c2 = frag_in.color.get(vertex_index::second);
  auto c3 = frag_in.color.get(vertex_index::third);
  frag_out.color = c1 * bc.x + c2 * bc.y + c3 * bc.z;
  return frag_out;
}
```

2.20 Mesh Shader Types

All OS: Metal 3 and later support mesh shader types. Metal uses these types in the mesh pipeline to render geometry and defines them in the header `<metal_mesh>`.

2.20.1 Mesh Grid Property Type

All OS: Metal 3 and later support mesh grid property types.

An object function (see section 5.1.7) can use the `mesh_grid_properties` type to specify the size of the mesh grid to dispatch for a given threadgroup from the object stage.

Call the following member function to control the number of threadgroups of the mesh grid that will be dispatched.

```
void set_threadgroups_per_grid(uint3)
```

If the member function `set_threadgroups_per_grid` for a given threadgroup of the object grid is never called, then no mesh grid will be dispatched for the given object grid threadgroup. Calls to `set_threadgroups_per_grid` behave as a write to `threadgroup` memory performed by each thread.

2.20.2 Mesh Type

All OS: Metal 3 and later support mesh types.

A mesh function (see section 5.1.8) can use an argument of type `mesh<V, P, NV, NP, t>` structure type to represent the exported mesh data. Table 2.10 describes the `mesh` template parameters.

Table 2.10. Mesh template parameter

Template parameter	Description
`V`	`V` is the vertex type.
`P`	`P` is the primitive type.
`NV`	`NV` is the maximum number of vertices.
`NP`	`NP` is the maximum number of primitives.

`t`	`t` specifies the topology of the mesh. It is one of the following enumeration values:
	`enum topology {`
	`  point,`
	`  line,`
	`  triangle`
	`}`

A valid vertex type `V` follows the same rules as the vertex function return type defined in section 5.2.3.3 with the following restrictions. The vertex type can be either

- A float4 represents the vertex position

or a user defined structure:

- Includes a field with the `[[position]]` attribute.
- May include other fields of scalar or vector of integer or floating-point type.
- Supports the following attributes from Table 2.11. Each attribute can be used once within the vertex type.

Table 2.11. Mesh vertex attributes

Attribute	Corresponding data types	Description
`clip_distance`	`float` or `float[n]` `n` needs to be known at compile time	Distance from the vertex to the clipping plane.
`invariant`	Not applicable; needs to be used with `[[position]]`	Marks the output position such that if the sequence of operations used to compute the output position in multiple vertex shaders is identical, there is a high likelihood that the resulting output position computed by these vertex shaders are the same value. Requires users to pass `-fpreserve-invariance`. See the description below for more information.
`point_size`	`float`	Size of a point primitive.
`position`	`float4`	The transformed vertex position.
`shared`	Not applicable	If present, then for every `amplification_id`, the output shall have the same value.

A valid primitive type follows the same rules as fragment input section 5.2.3.4. A valid primitive type is either:

- `void` indicating no per-primitive type.

or a user-defined structure:

- Includes fields of scalar or vector of integer or floating-point type.
- Supports only the following attributes from Table 2.12. Each attribute can be used once within the primitive type.

Table 2.12. Mesh primitive attributes

Attribute	Corresponding data types	Description
`primitive_culled`	`bool`	If set to true, the primitive is not rendered.

Attribute	Corresponding data types	Description
`primitive_id`	`uint`	The per-primitive identifier used with barycentric coordinates.
`render_target_array_index`	`uchar`, `ushort`, or `uint`	The render target array index, which refers to the face of a cubemap, data at a specified depth of a 3D texture, an array slice of a texture array, an array slice, or face of a cubemap array. For a cubemap, the render target array index is the face index, which is a value from `0` to `5`. For a cubemap array the render target array index is computed as: array slice index * 6 + face index.
`viewport_array_index`	`uchar`, `ushort`, or `uint`	The viewport (and scissor rectangle) index value of the primitive.

If the `mesh<V, P, NV, NP, t>` does not specify a field with `[[primitive_culled]]`, the behavior is the primitive is rendered. If the fragment shader reads the field, the value read is `false` because that fragment invocation belongs to a nonculled primitive.

Interpolation and sampling qualifiers are accepted on the vertex and primitive type members. The behavior is specified in section 5.2.3.4.

To minimize the possible user errors in mesh-fragment linking, the names of fields for user-defined vertex and primitive type needs to be unique between the vertex and primitive type.

An example of `mesh<V, P, NV, NP, t>` is:

```
struct VertexOut {
    float4 position [[position]];
};
struct PrimitiveOut
{
  float color [[flat]];
};

using custom_mesh_t = metal::mesh<VertexOut, PrimitiveOut, 64, 64,
                                  metal::topology::triangle>;
```

The mesh types contain the following static data member below.

Table 2.13. Mesh static members

Member variable	Description
`uint max_vertices`	The maximum number of vertices in the mesh (`NV`).
`uint max_primitive`	The maximum number of primitives in the mesh (`NP`).
`uint indices_per_primitive`	The number of indices per primitive based on topology `t`.
`uint max_indices`	The maximum number of indices (`max_primitive * indices_per_primitive`).

Call the following member function to set the vertex at index `I` in the range [`0`, `max_vertices`):

```
void set_vertex(uint I, V v);
```

If P is not `void`, call the following member function to set the primitive at index I in the range [`0`, `max_primitive`):

```
void set_primitive(uint I, P p);
```

Call the following member to set the primitive count where `c` is in the range `[0, max_primitive]`:

```
void set_primitive_count(uint c);
```

Call the following member to set the index where I is in the range [`0`, `max_indices`):

```
void set_index(uint I, uchar v);
```

It is legal to call the following `set_indices` functions to set the indices if the position in the index buffer is valid and if the position in the index buffer is a multiple of 2 (`uchar2` overload) or 2 (`uchar4` overload). The index `I` needs to be in the range [`0`, `max_indices`).

```
void set_indices(uint I, uchar2 v);
void set_indices(uint I, uchar4 v);
```

2.21 Packed Numeric Type

All OS: Metal 4.0 with SDK 26.4 or later support packed numeric.

All OS: Metal 4.1 extends the number of format types.

The header `<metal_packed_numeric>` defines format types and the template `packed_numeric_type<Format, N>`. See the Metal Feature Set Tables to determine which GPUs support this feature.

A format type is a tag type[1] that identifies a packed numeric data format, such as `int4b_format` for a 4-bit integer. Use a format type as:

- a `Format` parameter in `packed_numeric_type<Format, N>`.
- the `ElementType` of a `tensor<…>` (see section 2.22.2).

Table 2.14. Format types

Format type	Description
`int4b_format` All OS: Metal 4.0 SDK26.4	4-bit signed integer.
`uint4b_format` All OS: Metal 4.0 SDK26.4	4-bit unsigned integer.
`int2b_format` All OS: Metal 4.1	2-bit signed integer.
`uint2b_format` All OS: Metal 4.1	2-bit unsigned integer.
`metal_fp4_e2m1_format` All OS: Metal 4.1	4-bit signed floating-point (1-bit sign, 2-bit exponent, 1-bit mantissa).
`metal_fp8_e4m3_format` All OS: Metal 4.1	8-bit signed floating-point (1-bit sign, 4-bit exponent, 3-bit mantissa).
`metal_fp8_e5m2_format` All OS: Metal 4.1	8-bit signed floating-point (1-bit sign, 5-bit exponent, 2-bit mantissa).
`metal_fp8_ue8m0_format` All OS: Metal 4.1	8-bit unsigned floating-point (8-bit exponent).

[1] A tag type is a compile-time marker that carries no data and has no runtime behavior.

Table 2.15 and Table 2.16 describes the semantics of the fp4 and fp8 formats.

Table 2.15 FP4 format type properties

Properties	`metal_fp4_e2m1_format`
Exponent bias	1
Infinities	N/A
NaN	N/A
Zeros	S 00 0_2
Max normal value	S 11 1_2 = $\pm 2^2 * 1.5 = \pm 6.0$
Min normal value	S 01 1_2 = $\pm 2^0 * 1.0 = \pm 1.0$
Max subnormal value	S 00 1_2 = $\pm 2^0 * 0.5 = \pm 0.5$
Min subnormal value	S 00 1_2 = $\pm 2^0 * 0.5 = \pm 0.5$

Table 2.16 FP8 format type properties

Properties	`metal_fp8_e4m3_format`	`metal_fp8_e5m2_format`	`metal_fp8_ue8m0_format`
Exponent bias	7	15	127
Infinities	N/A	S 11111 00_2	N/A
NaN	S 1111 111_2	S 11111 $\{00, 10, 11\}_2$	11111111_2
Zeros	S 0000 000_2	S 0000 00 $_2$	N/A
Max normal value	S 1111 110_2 = $\pm 2^8 * 1.75 = \pm 448$	S 11110 11_2 = $\pm 2^{15} * 1.75 = \pm 57344$	N/A
Min normal value	S 0001 000_2 = $\pm 2^{-6}$	S 00001 00_2 = $\pm 2^{-14}$	N/A
Max subnormal value	S 0000 111_2 = $\pm 2^{-6} * 0.875$	S 00000 11_2 = $\pm 2^{-14} * 0.75$	N/A
Min subnormal value	S 0000 001_2 = $\pm 2^{-9}$	S 00000 01_2 = $\pm 2^{-16}$	N/A

The template `packed_numeric_type<Format,N>` represents `N` packed values of type `Format`, where Format can be a Format Type or a signed / unsigned as described in Table 2.17. `N` must be in `[1,32]`, with the total bit count (`N * bits per format`) divisible by 8.

Table 2.17. Packed numeric constraints

Format type	N
`int4b_format`	2, 4, ..., 32 (Multiples of 2)
`uint4b_format`	2, 4, ..., 32 (Multiples of 2)
`int2b_format`	4, 8, ..., 32 (Multiples of 4)
`uint2b_format`	4, 8, ..., 32 (Multiples of 4)
`char`	1-32
`signed char`	1-32
`uchar` `unsigned char`	1-32
`metal_fp4_e2m1_format`	2, 4, ..., 32 (Multiples of 2)
`metal_fp8_e4m3_format`	1-32
`metal_fp8_e5m2_format`	1-32
`metal_fp8_ue8m0_format`	1-32

Table 2.18 describes the member type defined by packed.

Table 2.18. Packed numeric member types

Type	Description
`storage_type`	The storage type numeric: `packed_vec<T, Format.NumBits *B/8>`

The packed numeric type supports the following functions:

```
// Constructors:
packed_numeric_type(storage_type storage) thread;
packed_numeric_type(const thread packed_numeric_type &o) thread;

// Assignment operator:
```

```
  thread packed_numeric_type &operator=(
           const thread packed_numeric_type &o) thread;
```

Call the following memory function to return the underlying storage:

```
  storage_type as_storage_type() thread;
```

Metal 4.1 supports the following template member function to extract `K` packed values starting from `start_pos`:

```
template <size_t K>
  packed_numeric_type<Format, K>
  extract(ushort start_pos = 0) const thread;
```

Metal 4.1 supports the following template function to unpack a packed numeric type to a basic floating-point type.

- `T` is the output floating-point type that can be `float`, `half`, or `bfloat`.
- `Format` is the supported format type in Table 2.19.
- N can be one of the values in Table 2.19.

```
template <class T, class Format, size_t N>
  vec<T, N> unpack(packed_numeric_type<Format, N> v);
```

Metal 4.1 supports the following template function to pack a floating-point type to a packed numeric type.

- `Format` is the supported format type in Table 2.19.
- `Rm` is the rounding mode.
- `Sm` is the saturation mode.
- `T` is the input floating-point that can be `float`, `half`, or `bfloat`.
- `N` can be one of the values in Table 2.19.

```
  enum class rounding_mode {
    to_nearest_even,
    to_nearest_away,
    toward_zero,
    toward_pos_inf,
    toward_neg_inf
  };

  enum class saturation_mode {
    none,
```

```
  saturate,
  symmetric_saturate
};

template <class Format,
          rounding_mode Rm = /* default rounding mode */,
          saturation_mode Sm = /* default saturation mode */,
          class T, size_t N>
 packed_numeric_type<Format, N> pack(vec<T, N> v);
```

Table 2.19. Unpack and pack supported format type

Format type	N
int4b_format	8, 16
uint4b_format	8, 16
char	4, 8
signed char	4, 8
uchar unsigned char	4, 8
metal_fp4_e2m1_format	8, 16
metal_fp8_e4m3_format	4, 8
metal_fp8_e5m2_format	4, 8

Table 2.20. Packed default rounding and saturation mode

Format type	Default rounding mode	Default saturation mode
int4b_format	toward_zero	saturate
uint4b_format	toward_zero	saturate
char	toward_zero	saturate
signed char	toward_zero	saturate
uchar unsigned char	toward_zero	saturate

Format type	Default rounding mode	Default saturation mode
`metal_fp4_e2m1_format`	to_nearest_even	`saturate`
`metal_fp8_e4m3_format`	to_nearest_even	`saturate`
`metal_fp8_e5m2_format`	to_nearest_even	`none`

2.22 Tensor Types

All OS: Metal 4 and later support tensor types.

Tensors are multidimensional data structures that are fundamental for machine learning. Every tensor contains a primary data plane. In Metal 4.1 and later, it may also include additional auxiliary planes. A tensor that has more than one plane is a multiplane tensor.

Each plane of the tensor has:

- a data type, where all elements are of the same type.
- a rank that represents the number of dimensions.
- a layout that represents the extents (size of each dimension) and strides (number of elements to skip past to get to the next element).

Metal defines two types of tensors:

- `tensor<…>` passed to shaders via arguments, global bindings, argument buffers, or allocated in the shader. Threads can access the storage based on the address space (`constant`, `device`, `threadgroup`, or `thread`) of the tensor element type.
- `cooperative_tensor<…>` whose storage is in `thread` and pre-partitioned among a set of participating threads.

2.22.1 Extents Type

The header `<metal_tensor>` defines the extents type. The type `extents<IndexType, size_t… Extents>` represents the multidimensional index space of tensors.

Table 2.21. Extents template parameters

Template parameter	Description
`IndexType`	`IndexType` is the type used for the size of each dimension and for index calculations. It can be any signed or unsigned integer type.

Template parameter	Description
`Extents`	Extents represent the extent (size of an integer interval) for each dimension (rank index). If the extent is determined dynamically (for example, if the size of the dimension is unknown at compile time), use `dynamic_extent`. Otherwise, the value must be representable in `IndexType`.

Table 2.22. Extents member types

Type	Description
`index_type`	Type used for the size of each dimension and for index calculations based on `IndexType`.
`size_type`	Type used to describe extents.
`rank_type`	Type used for rank.

A convenient alias template `dextents<class IndexType, size_t Rank>` is provided for extents where `Extents` for all dimensions is `dynamic_extent`.

Call the following member function to get the number of dimensions in extents:

```
static constexpr rank_type rank();
```

Call the following member function to get the number of dimensions in extents that are dynamic:

```
static constexpr rank_type rank_dynamic();
```

Call the following member function to get the size of an `extents` at a certain rank index:

```
static constexpr size_t static_extent(rank_type r);
```

Call the following member function to get dynamic extent size of an `extents` at a certain rank index:

```
constexpr index_type extent(rank_type r);
```

2.22.2 Tensor Type

The header `<metal_tensor>` defines the `tensor` type. In Metal 4.1 and later, the header also defines the `tensor_blockwise` type. A `tensor_blockwise` tag adds an auxiliary plane of elements to a `tensor`.

2.22.2.1 Tensors

The `tensor` is a class template type. Table 2.23 describes the template parameters you can specify when instantiating it.

```
template <class ElementType,
          class Extents,
          class DescriptorType,
          class… Tags>
struct tensor;
```

Use this type to pass tensors to shaders via arguments, global bindings, or argument buffers. You can also use it to create tensors in the shader. All specializations of the tensor type represent non-owning views into the tensor data. This means the data the tensor refers to isn't bound to the lifetime of the tensor object. The memory for all elements of the tensor at valid indices must be accessible; otherwise, the tensor is invalid and accessing it may result in undefined behavior.

Table 2.23. Tensor template parameters

Template parameter	Description
`ElementType`	`ElementType` is the fully qualified type of the tensor. A fully qualified type includes the value type contained in the tensor and the address space of the underlying storage, and its coherency. • The value type can be one of `half`, `bfloat`, `float`, `char`, `uchar`, `short`, `ushort`, `int`, or `uint`. In Metal 4.1 and later, the value type can also be a format type (section 2.21). Table 2.24 lists the supported format types. If the tensor has a scale plane (see section 2.22.2.2), the value type can be at most 8 bits in size. • The address space can be `constant`, `device`, `threadgroup`, or `thread` (see section 4). • The value can be `const`, `volatile`, or `coherent(device)` (see section 4.8).
`Extents`	`Extents` describes the dimensions of the tensor using `extents<…>` (see section 2.22.1). The extent `IndexType` can be one of `short`, `ushort`, `int`, `uint`, `long`, or `ulong`. If the tensor uses a format type, extend_0 must be a multiple of F_{block} (see Table 2.24).

Template parameter	Description
`DescriptorType`	`DescriptorType` describes where the descriptor lives. It can be either: `tensor_handle`: tensor contains a handle to the tensor descriptor, or `tensor_inline`: tensor holds the tensor descriptor. The default is `tensor_handle`.
`Tags`	`Tags` contains the additional compile-time properties. The only supported tag is `tensor_offset` which you can only use if the `DescriptorType` is `tensor_handle.` A tensor marked with that tag holds a set of offsets that shift the origin of the tensor (see section 2.22.2.6).

Table 2.24 lists the format types (section 2.21) supported by a tensor, along with their restrictions on minimum number of elements and minimum block size.

Table 2.24. Format types supported by tensors

Format type	F_b [2]	F_v [3]	F_{block} [4]
`int4b_format` All OS: Metal 4.0 SDK26.4	4-bits	2 elements	32 elements
`uint4b_format` All OS: Metal 4.0 SDK26.4	4-bits	2 elements	32 elements
`int2b_format` All OS: Metal 4.1	2-bits	4 elements	32 elements
`uint2b_format` All OS: Metal 4.1	2-bits	4 elements	32 elements
`metal_fp4_e2m1_format` All OS: Metal 4.1	4-bits	2 elements	32 elements
`metal_fp8_e4m3_format` All OS: Metal 4.1	8-bits	1 element	32 elements
`metal_fp8_e5m2_format` All OS: Metal 4.1	8-bits	1 element	32 elements

[2] F_b is the bit size of each element

[3] F_v is the minimum number of elements to read with the `get` and `set` functions

[4] F_{block} is the minimum block size for the format type.

Table 2.25 lists the tags supported by a tensor.

Table 2.25. Tag allowed for tensors

Tag	Description
`tensor_offset`	`tensor_offset` indicates that the tensor holds a set of offsets that shift the origin (see section 2.22.2.6). You can only use this tag if the `DescriptorType` is `tensor_handle`.
`tensor_blockwise<…>` All OS: Metal 4.1	`tensor_blockwise<…>` (see section 2.22.2.2) indicates that the tensor's primary data plane is organized into regular blocks that maps to an element in an auxlilary plane.

Table 2.26 describes the member types defined by `tensor<ElementType, Extents, DescriptorType, Tags…>`.

Table 2.26. Tensor member type definition

Type defined	Description
`element_type`	The fully qualified element type with which you specialized the `tensor` type.
`value_type`	The unqualified equivalent to `element_type`.
`extents_type`	The `extents` type with which you the specialized the `tensor` type (section 2.22.1).
`index_type`	The type you use for extents, strides, and indices.
`size_type`	The unsigned equivalent of `index_type`.
`rank_type`	The type you used for the rank of the `tensor`.

2.22.2.2 Tensor Blockwise

All OS: Metal 4.1 and later support `tensor_blockwise`.

The `tensor_blockwise` template represents an auxiliary plane of elements where each element maps to a block of elements in the primary data plane. Each block of elements in the

primary data plane corresponds to a single element in the blockwise plane. Block sizes define the mapping between the primary data plane and this auxiliary plane. Blockwise planes are particularly useful for quantized tensor formats, where scale factors apply to blocks of quantized values.

```
template <class PlaneTag,
          class ElementType,
          size_t… BlockSizes>
struct tensor_blockwise;
```

Table 2.27. PlaneTag

PlaneTag	Description
`tensor_plane_data`	Identifies the primary data plane
`tensor_plane_scales`	Identifies the scales plane

Table 2.28 describes the template parameters you can specify when instantiating the `tensor_blockwise` template.

Table 2.28. Tensor blockwise template parameter

Template parameter	Description
`PlaneTag`	Specifies the purpose of the plane. Only `tensor_plane_scales` is supported.
`ElementType`	Specifies the qualified element type of this plane. Only `metal_fp8_ue8m0_format` is supported.
`BlockSizes…`	Specifies the compile-time sizes along each dimension that determine the block granularity. The first dimension must be 32. All other dimensions must be 1.

Table 2.29 describes the member types.

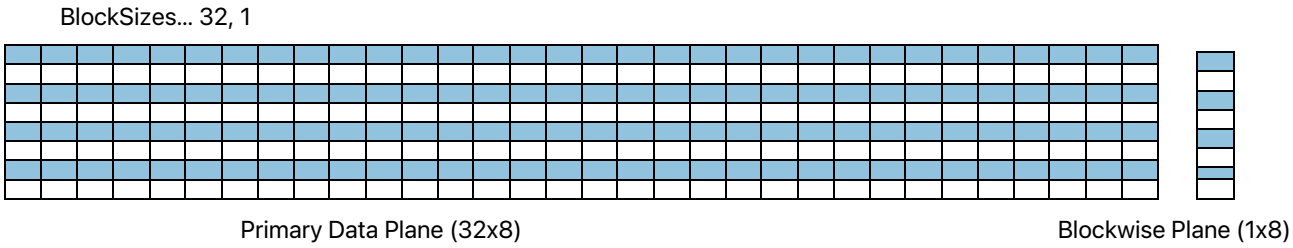
Table 2.29. Tensor blockwise member type definition

Type defined	Description
`value_type`	The fully qualified element type used to specialize the `tensor_blockwise` template.
`plane_tag_type`	The PlaneTag type used to specialize the `tensor_blockwise` template.

For example, you can create a tensor whose primary data plane is 4-bit floating-point organized into blocks of 32 elements, with an auxiliary scale plane of 8-bit floating-point values (see Figure 4).

```
template <class Extents, class Descriptor>
using mxfp4_tensor =
  tensor<metal_fp4_e2m1_format, Extents, Descriptor,
         tensor_blockwise<tensor_plane_scales,
                          device metal_fp8_ue8m0_format, 32, 1>>;
```

Figure 4. BlockSize example



Call the following constructor with a qualified ElementType pointer (`device` or `threadgroup`):

```
  tensor_blockwise(ElementType* plane_data) thread;
```

Call the following static member function to get the block size along the r^{th} dimension:

```
  static constexpr size_t get_block_size(size_t r);
```

2.22.2.3 Tensor Member Functions

The member functions described here operate on the primary data plane.

All tensors support the following constructors:

```
  tensor() thread;

  // Copy constructors.
  tensor(const thread tensor &) thread;
  tensor(const device tensor &) thread;
  tensor(const device coherent(device) tensor &) thread;
```

Tensor supports conversions from dynamic extents to static extents. Conversions from static extents to dynamic extents are not supported.

```
  // Conversion constructor dextent -> extent.
  tensor(const thread tensor<element_type,
                             OtherExtents,
                             tensor_handle,
                             Tags...> &other) thread;
  tensor(const device tensor<element_type,
                             OtherExtents,
                             tensor_handle,
                             Tags...> &other) thread;
  tensor(const device device(coherent)tensor<element_type,
                                             OtherExtents,
                                             tensor_handle,
                                             Tags...> &other) thread;
  tensor(const constant tensor<element_type,
                               OtherExtents,
                             tensor_handle,
                             Tags...> &other) thread;
```

The following are examples of dynamic extent conversions:

```
// Explicit conversions from dynamic to static extents.
void example_explicit_extent_conversions(device float *data) {
  // Tensor with dynamic extents.
  tensor<device float, dextents<int, 2>, tensor_inline>
  dynamic_tensor(data, extents<int, 2>(32, 64));

  // Explicit conversion to static extents (must match
  // runtime values).
  tensor<device float, extents<int, 32, 64>, tensor_inline>
  static_tensor = static_cast<tensor<
                               device float,
                               extents<int, 32, 64>,
                               tensor_inline>>(dynamic_tensor);
```

```
  // Explicit conversion to partially static extents.
  tensor<device float,
         extents<int, 32, dynamic_extent>,
         tensor_inline>
  partial_static = static_cast<tensor<
                               device float,
                               extents<int, 32, dynamic_extent>,
                               tensor_inline>>(dynamic_tensor);
}
```

Tensor supports conversions between different descriptor types. Implicit conversion from `tensor_handle` to `tensor_inline` is supported. This conversion can be expensive because it involves allocating and initializing a descriptor in the shader. Copies between `tensor_inline` instances may also be expensive for the same reason and should be avoided. Pass tensors by reference to avoid unnecessary copies.

```
  // Conversion constructor tensor_handle,
  // tensor_offset <- tensor_handle.
  tensor(const thread tensor<element_type,
                             OtherExtents,
                             tensor_handle> &other) thread;
  tensor(const device tensor<element_type,
                             OtherExtents,
                             tensor_handle> &other) thread;
  tensor(const device coherent(device)
                       tensor<element_type,
                              OtherExtents,
                              tensor_handle> &other) thread;
  tensor(const constant tensor<element_type,
                              OtherExtents,
                              tensor_handle> &other) thread;
```

Call the following member function to get the rank (number of dimensions) of the tensor:

```
  static constexpr size_t get_rank();
```

Call the following member function to get the static extent size (size of a dimension) of the tensor along the r^{th} dimension:

```
  static constexpr size_t get_static_extent(rank_type r);
```

For example, if `extents<int, 32, 64>` of the tensor then `get_static_extent(0)` returns `32` and `get_static_extent(1)` is `64`.

Call the following member function to get the extent of the tensor along the r^{th} dimension:

```
  index_type get_extent(rank_type r) const;
```

Call the following member function to get the stride of the tensor along the r^{th} dimension:

```
index_type get_stride(rank_type r) const;
```

Note: `get_stride(0)` always returns `1`.

In Metal 4.1 and later, call the following member function to get the stride, in bytes, of the tensor along the r^{th} dimension:

```
size_type get_stride_bytes(rank_type r) const;
```

Call the `[]` operator to get a reference to an element of a tensor at multidimensional `index`. If the index is out of bounds of the tensor, access to the element results in undefined behavior.

```
template<class… OtherIndexTypes>
  reference operator[](OtherIndexTypes…index) const;

template<class OtherIndexType>
  reference operator[](
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to load an element of the tensor at the given `index`. The `get` function supports broadcast semantics. If the multidimensional index at i^{th} dimension is greater than zero and `get_extent(i)` is `1`, the effective index is `0` along that dimension. It also supports bound-checking behavior. If the effective index is out of bounds of the tensor, the load returns the default value. If the `element_type` is a format type (see section 2.21), this function is not available.

```
template<class… OtherIndexTypes>
  value_type get(OtherIndexTypes…index) const;

template<class OtherIndexType>
  value_type get(
    thread const array<OtherIndexType, get_rank()> &index) const;
```

In Metal 4 in iOS, iPadOS, macOS, tvOS, watchOS, and visionOS 26.4 and later, call the following member function to load consecutive elements of the primary data plane of the tensor starting at the given `index`, returning them as a vector. `ValueType` can be the tensor's value type, a vector or packed-vector of the value type, or a format type (see section 2.21).

```
template<class ValueType, class… OtherIndexTypes>
  ValueType get(OtherIndexTypes… index) const;

template<class ValueType, class OtherIndexType>
  ValueType get(
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to store a value `v` to an element of a tensor at `index`. If the index is out of bounds of the tensor, the GPU drops the store.

```
template<class… OtherIndexTypes>
  void set(value_type v, OtherIndexTypes…index) const;

template<class OtherIndexType>
  void set(value_type, v,
    thread const array<OtherIndexType, get_rank()> &index) const;
```

In Metal 4 on iOS, iPadOS, macOS, tvOS, watchOS, and visionOS 26.4 and later, call the following member function to store consecutive elements into the primary data plane of the tensor starting at the given `index`. `ValueType` follows the same allowed types as the `get` function above.

```
template<class ValueType, class… OtherIndexTypes>
  void set(ValueType v, OtherIndexTypes… index) const;

template<class ValueType, class OtherIndexType>
  void set(ValueType v,
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to get a slice of a tensor whose origin is shifted by `index` and whose extents are `SliceExtents`. The returned slice tensor has the same `DescriptorType` as the original tensor and is either an origin-shifted tensor (see section 2.22.2.6) or a shader-allocated tensor (see section 2.22.2.7). If `OtherExtents` is `dynamic_extent`, slice returns the remaining elements starting from `index`. If this causes the tensor to be out of bounds of the input tensor, it results in undefined behavior.

```
template<size_t… SliceExtents, class… OtherIndexTypes>
  tensor<ElementType, SliceExtents, DescriptorType, SliceTags…>
  slice(OtherIndexTypes… index);
```

The following shows some example calls to the `slice` function:

```
// Assume t has extents [64, 32].
tensor<device half, dextents<int, 2>, tensor_handle> t;

// Creates a [32, 16] view starting at (8, 16)
//   tensor<device half, extents<int, 32, 16>,
//          tensor_handle, tensor_offset>.
auto s1 = t.slice<32, 16>(8, 16);

// Creates a [32, 8] view starting at (8, 24)
//tensor<device half, extents<int, 32, dynamic_extent>,
//       tensor_handle, tensor_offset>.
auto s2 = t.slice<32, dynamic_extent>(8, 24);

// Illegal: 48 + 32 > 64
// tensor<device half, extents<int, 32, 16>,
```

```
//        tensor_handle, tensor_offset>.
auto s3 = t.slice<32, 16>(48, 16);

// Creates a [48, 24] view starting at (16, 8)
// tensor<device half, dextents<int, 2>,
//        tensor_handle, tensor_offset>.
auto s4 = t.slice(16, 8);

// Illegal: 40 > 31
// tensor<device half, dextents<int, 2>,
//        tensor_handle, tensor_offset>.
auto s5 = t.slice(48, 40);

// Assume t2 has extents [32, 32].
tensor<device half, extents<int, 32, dynamic_extent>,
       tensor_handle> t2;

// Illegal: 64 > 32.
// tensor<device half, extents<int, 64, 16>,
//        tensor_handle, tensor_offset>.
auto s6 = t2.slice<64, 16>(8, 16);

// Illegal: Cannot dynamic slice a static extent
// tensor<device half, extents<int, dynamic_extent, 8>,
//        tensor_handle, tensor_offset>.
auto s7 = t2.slice<dynamic_extent, 8>(0, 16);

// Assume t3 has extents [32, 32].
tensor<device half, extents<int, dynamic_extent, 32>,
       tensor_handle> t3;

// Illegal: 28 + 8 > 32
// tensor<device half, extents<int, dynamic_extent, 8>,
//        tensor_handle, tensor_offset>.
auto s8 = t3.slice<dynamic_extent, 8>(0, 28);

// Slicing format-type tensors - indices and extents must align
// with 32.

// Assume t_int4 has extents [256, 64].
tensor<device int4b_format, dextents<int, 2>, tensor_handle> t_int4;

// Valid: extent[0] (64) is a multiple of 32; index[0] (128) is a
// multiple of 32.
auto s9 = t_int4.slice<64, 32>(128, 0);

// Valid: extent[0] (192) is a multiple of 32; index[0] (64) is a
// multiple of 32.
auto s10 = t_int4.slice(64, 8);
```

```
// Illegal: extent[0] (48) is not a multiple of 32.
auto s11 = t_int4.slice<48, 32>(16, 0);

// Illegal: index[0] (16) is not a multiple of 32.
auto s12 = t_int4.slice<128, 32>(16, 0);
```

2.22.2.4 Tensor Multiplane Member Functions

In Metal 4.1 and later, tensor supports the following template member functions for multiplane tensors. The template parameter `PlaneTag` identifies the plane to work on.

Call the following member function to get the static extent (size of a dimension) of the `PlaneTag` plane of the tensor along the r^{th} dimension:

```
  template<class PlaneTag>
    static constexpr size_type get_static_extent(rank_type r);
```

Call the following member function to get the block size of the `PlaneTag` plane of the tensor along the r^{th} dimension. This function isn't available if `PlaneTag` is not valid for the tensor.

```
  template<class PlaneTag>
    static constexpr size_type get_block_size(rank_type r);
```

The following function is equivalent to `get_block_size<tensor_plane_data>(r)`:

```
  static constexpr size_type get_block_size(rank_type r);
```

The following shows some example calls to these functions and their expected results:

```
// Create a 2d tensor whose data plane is half.
using simple_half_tensor =
  tensor<device half, dextents<int, 2>, tensor_handle>;

simple_half_tensor::get_block_size<tensor_plane_data>(0) == 1;
simple_half_tensor::get_block_size<tensor_plane_data>(1) == 1;

// Create a 2d tensor whose data plane is 4-bit integer.
using simple_int4_tensor =
  tensor<device int4b_format, dextents<int, 2>, tensor_handle>;
simple_int4_tensor::get_block_size<tensor_plane_data>(0) == 1;
simple_int4_tensor::get_block_size<tensor_plane_data>(1) == 1;
```

```
// Create a 2d multiplane tensor whose primary data plane is half
// and whose auxiliary scale plane is 8-bit floating-point with
// a 32x1 block size.
using multiplane_half_tensor =
  tensor<device half, dextents<int, 2>, tensor_handle,
         tensor_blockwise<tensor_plane_scales,
                          device metal_fp8_ue8m0_format, 32, 1>>;
multiplane_half_tensor::get_block_size<tensor_plane_data>(0) == 1;
multiplane_half_tensor::get_block_size<tensor_plane_data>(1) == 1;
multiplane_half_tensor::get_block_size<tensor_plane_scales>(0)
  == 32;
multiplane_half_tensor::get_block_size<tensor_plane_scales>(1) == 1;
multiplane_half_tensor::get_block_size(0) == 1;
multiplane_half_tensor::get_block_size(1) == 1;

// Create a 2d multiplane tensor whose primary data plane is half
// and whose auxiliary scale plane is 8-bit floating-point with
// a 32x1 block size.
using multiplane_int4_tensor =
  tensor<device int4b_format, dextents<int, 2>, tensor_handle,
         tensor_blockwise<tensor_plane_scales,
                          device metal_fp8_ue8m0_format, 32, 1>>;
multiplane_int4_tensor::get_block_size<tensor_plane_data>(0) == 1;
multiplane_int4_tensor::get_block_size<tensor_plane_data>(1) == 1;
multiplane_int4_tensor::get_block_size<tensor_plane_scales>(0)
  == 32;
multiplane_int4_tensor::get_block_size<tensor_plane_scales>(1) == 1;
multiplane_int4_tensor::get_block_size(0) == 1;
multiplane_int4_tensor::get_block_size(1) == 1;
```

Call the following member function to get the runtime extent of the `PlaneTag` plane of the tensor along the r^{th} dimension:

```
  template<class PlaneTag>
    index_type get_extent(rank_type r) const;
```

Note: If the extent is static (not `dynamic_extent`), this function returns the same value as `get_static_extent`.

Call the following member function to get the stride of the `PlaneTag` plane of the tensor along the r^{th} dimension:

```
  template<class PlaneTag>
    index_type get_stride(rank_type r) const;
```

Note: `get_stride(0)` always returns 1.

Call the following member function to get the stride, in bytes, of the `PlaneTag` plane of the tensor along the r^{th} dimension:

```
template<class PlaneTag>
  size_type get_stride_bytes(rank_type r) const;
```

Call the `[]` operator to get a reference to an element of the `PlaneTag` plane of the tensor at the given multidimensional `index`. If the index is out of bounds of the tensor, access to the element results in undefined behavior.

```
template<class PlaneTag, class… OtherIndexTypes>
  reference operator[](OtherIndexTypes…index) const;

template<class PlaneTag, class OtherIndexType>
  reference operator[](
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to load an element of the `PlaneTag` plane of the tensor at the given `index`. The broadcast and bounds-check behaviors match those of the `get` function, described earlier.

```
template<class PlaneTag, class… OtherIndexTypes>
  value_type get(OtherIndexTypes…index) const;

template<class PlaneTag, class OtherIndexType>
  value_type get(
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to load consecutive elements of the `PlaneTag` plane of the tensor starting at the given `index`, returning them as a vector. `ValueType` can be the tensor's value type, a vector or packed-vector of the value type, or a format type (see section 2.21).

```
template<class PlaneTag, class ValueType, class… OtherIndexTypes>
  ValueType get(OtherIndexTypes… index) const;

template<class PlaneTag, class ValueType, class OtherIndexType>
  ValueType get(
    thread const array<OtherIndexType, get_rank()> &index) const;
```

Call the following member function to store consecutive elements into the `PlaneTag` plane of the tensor starting at the given `index`. `ValueType` follows the same allowed types as the `get` function, above.

```
  template<class PlaneTag, class ValueType, class… OtherIndexTypes>
    void set(ValueType v, OtherIndexTypes… index) const;

  template<class PlaneTag, class ValueType, class OtherIndexType>
    void set(ValueType v,
      thread const array<OtherIndexType, get_rank()> &index) const;
```

The following are examples of calls to the `slice` function on a multiplane tensor:

```
// Slicing multi-plane tensors - indices/extents must align with
// scale block sizes.

// Assume t_mp has extents [256, 37].
tensor<device int4b_format, dextents<int, 2>, tensor_handle,
       tensor_blockwise<tensor_plane_scales,
                        device metal_fp8_ue8m0_format, 32, 1>> t_mp;

// Valid: extent[0] (64) is a multiple of 32 && index[0] is a
// multiple of 32.
auto s13 = t_mp.slice<64, 7>(64, 14);

// Illegal: index[0] (16) not multiple of block size 32.
auto s14 = t_mp.slice<64, 13>(16, 0);


// Dynamic extent slicing - takes remaining elements from index to
// end.

// Valid: index[0] is a multiple of 32.
auto s15 = t_mp.slice(128, 0);

// Illegal: index[0] is not a multiple of 32.
auto s16 = t_mp.slice(48, 0);
```

2.22.2.5 Host-bound Tensors

Host-bound tensors are tensors that are allocated and set up on the host. To declare a host-bound tensor, specify `tensor_handle` to the `DescriptorType` template parameter. The `ElementType` may be qualified with either the `device` or `constant` address spaces.

```
[[kernel]]
void gemm(tensor<device half, dextents<int, 2>,
```

```
                tensor_handle>                    ta [[buffer(0)]],
          tensor<constant float, dextents<int, 2>> tb [[buffer(1)]])
{…}
```

The example above defines `ta` as a tensor allocated in `device` memory with value type of `half`. It defines `tb` as a tensor allocated in `constant` memory with value type of `float`. Note that since the default `DescriptorType` is `tensor_handle`, it is unnecessary to pass it in this case.

2.22.2.6 Origin-shifted Tensors

Origin-shifted tensors are host-bound tensors tagged with `tensor_offset`. Origin-shifted tensors have their origin shifted by a set of offsets (in number of elements) relative to the base tensor. Calculate the new extents of the tensor relative to the origin, that is, for dimension `i`:

```
   get_extent(i) = base.get_extent(i) – offset(i);
```

For example, you can get an origin-shifted tensor using the `slice` member function of `tensor`. The return tensor aliases the memory of the base tensor. The first call to slice returns a tensor with dynamic extents because the remaining number of elements in the tensor is based on the original tensor and the shifted origin. The second call returns a 16x16x16 tensor whose origin starts at `(32, 32, 32)` of the base tensor. The last call returns a 16x16x16 tensor whose origin starts at `(16, 16, 32)` of the base tensor.

```
[[ kernel ]]
void offsetTensor(tensor<device float,
                         extents<int, 64, 128, 256>> tbase) {
  // Origin-shifted tensor.
  tensor<device float, dextents<int,3>,
         tensor_handle, tensor_offset> t3 = tbase.slice(8, 16, 32);

  // Origin-shifted 16x16x16 tensor.
  tensor<device float, extents<int, 16, 16, 16>,
         tensor_handle, tensor_offset> t4 =
    tbase.slice<16, 16, 16>(32, 32, 32);

  // Origin-shifted tensor.
  auto t5 = tbase.slice<16, 16, 16>(16, 16, 32);
}
```

2.22.2.7 Shader-Allocated Tensors

Shader-allocated (inline) tensors are tensors allocated directly inside a shader. To declare a shader-allocated tensor, specify `tensor_inline` to the `DescriptorType` template parameter. You may qualify `ElementType` with either the `device`, `constant`,

`threadgroup`, or `thread` address spaces. You can't define shader allocated tensor types in an aggregate type (see section 2.12).

Shader-allocated tensors support the following additional constructors. Table 2.30 lists the parameter description.

```
  // Raw constructor with pointer, extents, strides.
template <class OtherExtentsType, class OtherStrideType>
  tensor(data_handle_type ptr,
         thread const OtherExtentsType &_extents,
         thread const array<OtherStrideType, get_rank()>&_strides)
         thread;

  // Raw constructor with pointer, extents (with implied packed
  // layout for strides).
  template <class OtherExtentsType,
  tensor(data_handle_type ptr,
         thread const OtherExtentsType &_extents) thread;
```

In Metal 4.1 and later, you can allocate a shader-allocated tensor with a blockwise plane.

```
template <class OtherExtentsType, class OtherStrideType,
          class PlaneType>
  tensor(data_handle_type ptr,
         thread const OtherExtentsType&_extents,
         thread const array<OtherStrideType, get_rank()>&_strides,
         thread PlaneType &plane)
         thread;
```

Table 2.30. Shader-allocated tensor parameters

Parameter	Description
`ptr`	Pointer to the primary data plane. It must be non-null and point to a valid memory region large enough to accommodate all elements in the tensor defined by `extents` and `strides`. Otherwise, the behavior is undefined. If the `element_type` is a format type in `device`, `constant`, or `threadgroup`, the pointer must point to 128-byte-aligned memory.
`extents`	Dimensions of the tensor. If the `element_type` is a format type, `extents[0]` must be a multiple of F_{block} (see Table 2.24).
`strides`	Stride in terms of elements for each dimension. `strides[0]` is always 1. If the `element_type` is a format type, the row stride in bytes (`strides[1]`

Parameter	Description
	* F_b / CHAR_BIT), where F_b is the format's bit size (see Table 2.24), must satisfy: • In thread, the row stride in bytes must be a whole number of bytes. • In device, constant, or threadgroup, the row stride in bytes must be a multiple of 128 bytes.
plane	Blockwise plane data configuration. The plane is a specialization of tensor_blockwise. The address space of PlaneType::element_type must match the address space of ptr.

The example below shows a use of the constructor:

```
[[kernel]] void func1(threadgroup half *buf) {
   tensor<threadgroup half, dextents<int, 3>, tensor_inline>
     t1(buf, dextents<int, 3>(16, 32, 64));
   auto t2 = tensor(buf, dextents<int, 3>(16, 32, 64));
   …
}
```

The following are examples of valid and invalid cases:

```
void test(device char *devmem, threadgroup char *tgmem,
          thread char *tmem,
          device float large_buffer[2048],
          device float large_strided_buffer[2048],
          device float buffer[128],
          device float strided_buffer[1024]) {
   auto extents = …;

  // Valid: Generic element_type with device, threadgroup and
  // thread.
  tensor<char, dextents<int, 2>, tensor_inline>
  t1(devmem, extents); // OK.

  tensor<char, dextents<int, 2>, tensor_inline>
  t2(tgmem, extents); // OK.

  tensor<char, dextents<int, 2>, tensor_inline>
  t3(tmem, extents); // OK.
```

```
// Valid: Device element_type with device pointer.
tensor<device char, dextents<int, 2>, tensor_inline>
t5(devmem, extents); // OK.

// Valid: The threadgroup element_type with threadgroup pointer.
tensor<threadgroup char, dextents<int, 2>, tensor_inline>.
t7(tgmem, extents); // OK.

// Valid: The thread element_type with thread pointer.
tensor<thread char, dextents<int, 2>, tensor_inline>.
t8(tmem, extents); // OK.

// Invalid: The device element_type with threadgroup and thread
// pointer.
tensor<device char, dextents<int, 2>, tensor_inline>
t9(tgmem, extents); // Error.

tensor<device char, dextents<int, 2>, tensor_inline>
t9b(tmem, extents); // Error.

// Invalid: The threadgroup element_type with device and
// thread pointer.
tensor<threadgroup char, dextents<int, 2>, tensor_inline>
t10(devmem, extents); // Error.

tensor<threadgroup char, dextents<int, 2>, tensor_inline>
t10b(tmem, extents); // Error.

// Invalid: The thread element_type with device/threadgroup
// pointer.
tensor<thread char, dextents<int, 2>, tensor_inline>
t10c(devmem, extents); // Error.

tensor<thread char, dextents<int, 2>, tensor_inline>
t10d(tgmem, extents); // Error.

// Valid: Sufficient memory backing: 64*32 = 2048 elements.
tensor<device float, dextents<int, 2>, tensor_inline>
t_valid(large_buffer, extents<int, 2>(64, 32));

// Valid: Sufficient memory with non-unit strides:
// stride[1]=128, needs 128*16=2048.
tensor<device float, dextents<int, 2>, tensor_inline>
t_strided_valid(large_strided_buffer,
                extents<int, 2>(32, 16),
                array<int, 2>{1, 128});

// Invalid: A null pointer.
device float *null_ptr = nullptr;
```

```
  tensor<device float, dextents<int, 2>, tensor_inline>
  t_null(null_ptr, extents<int, 2>(64, 32)); // Undefined behavior.

  // Invalid: Insufficient memory backing.
  // Requires 64*32 = 2048 floats,
  // the buffer only has 128 floats (512 bytes).
  tensor<device float, dextents<int, 2>, tensor_inline>
  t_small(buffer, extents<int, 2>(64, 32)); // Undefined behavior.

  // Invalid: Insufficient memory with strides.
  // Requires stride[1]*extent[1] = 128*16 = 2048 elements;
  // the strided_buffer has 1024 elements.
  tensor<device float, dextents<int, 2>, tensor_inline>
  t_strided(strided_buffer, extents<int, 2>(32, 16),
            array<int, 2>{1, 128}); // Undefined behavior.
}
```

2.22.3 Cooperative Tensor Type

The header `<metal_cooperative_tensor>` defines the `cooperative_tensor<ElementType, Extents, Layout>` type. The `cooperative_tensor` represents a tensor with elements that are partitioned across a set of participating threads in thread memory. Each thread has access to only the elements in its partition. These threads belong to the same threadgroup and may be spread across consecutive SIMD-groups. You can't define a `cooperative_tensor` in an aggregate type (see section 2.12).

Table 2.31. Cooperative tensor template parameters

Template parameter	Description
`ElementType`	`ElementType` is the type of the underlying type in the tensor. For cooperative tensor, the address space is `thread`.
`Extents`	`Extents` describes the dimensions of the tensor using `extents<…>` (see section 2.22.1).
`Layout`	`Layout` specifies the mapping of the multidimensional coordinate space of the tensor to the prepartitioned storage for each thread.

You typically don't construct `cooperative_tensor` directly as the `Layout` is device specific. Instead, you use libraries such as Metal Performance Primitives (see section 7), a library of optimized primitives that include operators that work on tensors such as matrix multiplication and convolution. You create them using the tensor operations, which use them to store intermediate results. The tensor operation determines an efficient and performant `Layout` for a `cooperative_tensor` based on its usage and the GPU.

2.22.3.1 Layout

Layout is an opaque object that provides the following interface that describes the configuration of the `cooperative_tensor`. The layout is used by the `cooperative_tensor` to implement its various functions. You don't usually need to call these functions.

Call the following function to return the amount of storage each thread needs to allocate for the `cooperative_tensor`:

```
static size_t thread_storage_size();
```

Call the following function to return the alignment of storage each thread needs to allocate for the `cooperative_tensor`:

```
static const_expr size_t thread_storage_align();
```

Call the following function to return the maximum number of elements that the `cooperative_tensor` can hold per thread:

```
static thread_size_type get_capacity(const thread void *this);
```

Call the following function to determine if the element at `idx` is valid:

```
static bool is_valid_element(const thread void *, uint16 idx);
```

Call the following function to get the pointer to the element at `idx`. If the `idx` is invalid, the result is undefined. :

```
static thread void *
get_element_pointer(const thread void *, uint16_t idx);
```

Call the following function to return the index given the pointer to the element. If the pointer is not a valid element of the `cooperative_tensor`, the result is undefined.

```
static uint16_t
get_element_index(const thread void *storage,
                  const thread void *element);
```

Call the following function to return the set of multi-dimensional index at `idx`:

```
template <class OtherIndexType, size_t Rank>
  static array<OtherIndexType, Rank>
  get_multidimensional_index(const thread void *, uint16_t idx);
```

Call the following function to load elements belonging to this thread into per-thread storage:

```
template <class T, class E, class D, class... Tags>
  static void load(thread void *storage,
                   const thread tensor<T, E, D, Tags...> &);
```

Call the following function to store elements belonging to this thread from per-thread storage into the destination tensor:

```
template <class T, class E, class D, class... Tags>
  static void store(const thread void *storage,
                    const thread tensor<T, E, D, Tags...> &);
```

The following function implements this interface when `FromIterator` can be converted to `ToIterator`:

```
 template <class FromIterator, class ToIterator>
 static uint16_t map_index(const thread void *from_storage,
                           uint16_t from_idx,
                           const thread void *to_storage);
```

2.22.3.2 Cooperative Tensor

Table 2.32. Cooperative tensor type definition

Type defined	Description
`element_type`	The fully qualified element type with which you specialized the `cooperative_tensor` type.
`value_type`	The unqualified equivalent to `element_type`.
`extents_type`	The `extents` type with which you specialized the `cooperative_tensor` type (section 2.22.1).
`index_type`	The index type you used for `extents`.
`size_type`	The unsigned equivalent of `index_type`.
`rank_type`	The type you used for the rank of the `cooperative_tensor` (via `extents`).
`thread_index_type`	The index type you used to index per-thread storage.
`thread_size_type`	The unsigned equivalent of `thread_index_type`.
`data_handle_type`	Pointer to the `element_type`.
`reference`	Reference to the `element_type`.
`const_reference`	`const` equivalent of `reference`.
`iterator`	Random access iterator to `element_type`.
`const_iterator`	`const` equivalent of `iterator`.
`layout`	The layout with which you specialized the `cooperative_tensor` type.

Call the following member function to get the rank of the cooperative tensor:

```
static constexpr rank_type get_rank();
```

Call the following member function to cooperatively load all elements from a tensor `t` into the cooperative tensor. The function supports broadcast semantics where a tensor is expanded into a compatible cooperative tensor. Two tensors are compatible for broadcasting if they have the same rank and when iterating over the dimensions, the sizes are equal or the tensor we are loading from is size `1`. For example, you can load a tensor a `64x1 tensor` into a `64x2 cooperative tensor`.

```
template<class T, class E, calls D, class…>
  void load(const thread tensor<T, E, D, …> &t) thread;
```

Call the following member function to cooperatively store all elements from a cooperative tensor into the tensor `t`. The function supports broadcast semantics as described in the load. For example, you can store a 64x1 `cooperative tensor` to a 64x2 `tensor`.

```
template<class T, class E, calls D, class…>
  void store(thread tensor<T, E, D, …> &t) thread const;
```

Call the following member function to the maximum number of elements that are private to this thread. This value is uniform across all threads participating in the cooperative tensor.

```
thread_size_type get_capacity() thread const;
```

Call the `[]` operator to get a reference to an element of a cooperative tensor at `idx`. If the `idx` is out-of-bound of the cooperative tensor, access to the element results in undefined behavior.

```
reference operator[](thread_index_type idx);
const_reference operator[](thread_index_type idx) const;
```

Call the following member function to get the value at `it`, `idx`, or `ptr` from memory owned by this thread:

```
value_type get(const_iterator it) thread const;
value_type get(thread_index_type idx) thread const;
value_type get(const thread element_type *ptr) thread const;
```

Call the following member function to set the value at `it`, `idx`, or `ptr` from memory owned by this thread:

```
void set(iterator it, value_type v) thread;
void set(thread_index_type idx, value_type v) thread;
void set(thread element_type *ptr, value_type v) thread;
```

Call the following member function to get the logical multidimensional index that corresponds to the element at `it`, `idx`, or `ptr`:

```
array<index_type, get_rank()>
get_multidimensional_index(const_iterator it) thread const;

array<index_type, get_rank()>
get_multidimensional_index(thread_index_type idx) thread const;

array<index_type, get_rank()>
get_multidimensional_index(
  const thread element_type *ptr) thread const;
```

Call the following member function to determine if the element pointed to by `it`, `idx`, or `ptr` is valid. If the return value is false, the element is invalid, and access to it is undefined behavior.

```
bool is_valid_element(const_iterator it) const;
bool is_valid_element(thread_index_type idx) const;
bool is_valid_element(const thread element_type *ptr) const;
```

Call the following member functions to return an iterator to the beginning, which corresponds to the same element at index `0`:

```
iterator begin() thread;
const_iterator begin() thread const;
```

Call the following member functions to return an iterator to the end:

```
iterator end() thread;
const_iterator end() thread const;
```

Call the following member functions to return an iterator corresponding to the element corresponding to `idx` or `ptr`:

```
iterator get_iterator(thread_index_type idx) thread;
const_iterator get_iterator(thread_index_type idx) thread const;

iterator get_iterator(const thread element_type *ptr) thread;
const_iterator get_iterator(
                const thread element_type *ptr) thread const;
```

Call the following functions that point to the element in this `cooperative_tensor` that corresponds to the element pointed to by `it` from another `cooperative_tensor`. These functions may be exposed if the layout of two `cooperative_tensor`s are compatible.

```
template<class OtherIterator>
  iterator map_iterator(const thread OtherIterator &it);

template<class OtherIterator>
  const_iterator map_iterator(
                  const thread OtherIterator &it) const;
```

2.23 Type Conversions and Reinterpreting Data

The `static_cast` operator converts from a scalar or vector type to another scalar or vector type using the default rounding mode with no saturation (when converting to floating-point, round ties to even; when converting to an integer, round toward zero). If the source type is a scalar or vector Boolean, the value `false` is converted to zero, and the value `true` is converted to one.

Metal adds an `as_type<type-id>` operator to allow any scalar or vector data type (that is not a pointer) to be reinterpreted as another scalar or vector data type of the same size. The bits in the operand are returned directly without modification as the new type. The usual type promotion for function arguments is not performed.

For example, `as_type<float>(0x3f800000)` returns `1.0f`, which is the value of the bit pattern `0x3f800000` if viewed as an IEEE-754 single precision value.

Using the `as_type<type-id>` operator to reinterpret data to a type with a different number of bytes results in an error.

Examples of legal and illegal type conversions:

```
float f = 1.0f;
// Legal. Contains: 0x3f800000
uint u = as_type<uint>(f);

// Legal. Contains:
// (int4)(0x3f800000, 0x40000000, 0x40400000, 0x40800000)
```

```
float4 f = float4(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_type<int4>(f);

int i;
// Legal.
short2 j = as_type<short2>(i);

half4 f;
// Error. Result and operand have different sizes
float4 g = as_type<float4>(f);

float4 f;
// Legal. g.xyz has same values as f.xyz.
float3 g = as_type<float3>(f);
```

2.24 Implicit Type Conversions

Implicit conversions between scalar built-in types (except void) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value `5` is converted to the floating-point value `5.0`. A `bfloat` is an extended floating-point type that only allows implicit conversion to a type of greater floating-point rank. While `bfloat` can be implicitly converted to float, it cannot be implicitly converted to half, and neither float nor half can be implicitly converted to `bfloat`.

All vector types are considered to have a higher conversion rank than scalar types. Implicit conversions from a vector type to another vector or scalar type are not permitted and a compilation error results. For example, the following attempt to convert from a 4-component integer vector to a 4-component floating-point vector fails.

```
int4 i;

float4 f = i; // Results in a compile error.
```

Implicit conversions from scalar-to-vector types are supported. The scalar value is replicated in each element of the vector. The scalar may also be subject to the usual arithmetic conversion to the element type used by the vector.

For example:

```
float4 f = 2.0f; // f = (2.0f, 2.0f, 2.0f, 2.0f)
```

Implicit conversions from scalar-to-matrix types and vector-to-matrix types are not supported and a compilation error results. Implicit conversions from a matrix type to another matrix, vector or scalar type are not permitted and a compilation error results.

Implicit conversions for pointer types follow the rules described in the C++17 Specification.

3 Operators

All OS: Metal 1 and later support scalar, vector, and matrix operators.

For indirect command buffers, the assignment operator (=) does not copy the contents of a command. For more about copying commands in indirect command buffers, see section 6.17.3.

3.1 Scalar and Vector Operators

This section lists both binary and unary operators and describes their actions on scalar and vector operands.

1. The arithmetic binary operators, add (+), subtract (—), multiply (∗) and divide (/), act upon scalar and vector, integer, and floating-point data type operands. Following the usual arithmetic conversions, all arithmetic operators return a result of the same built-in type (integer or floating-point) as the type of the operands. After conversion, the following cases are valid:

    - If the two operands of the arithmetic binary operator are scalars, the result of the operation is a scalar.
    - If one operand is a scalar, and the other operand is a vector,
        - The scalar converts to the element type that the vector operand uses.
        - The scalar type then widens to a vector that has the same number of components as the vector operand.
        - Metal performs the operation componentwise, which results in a same size vector.
    - If the two operands are vectors of the same size, Metal performs the operation componentwise, which results in a same size vector.

    Division on integer types that result in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type, such as `TYPE_MIN/−1` for signed integer types or division by zero, does not cause an exception but results in an unspecified value. Division by zero for floating-point types results in ±∞ or `NaN`, as prescribed by IEEE-754. (For more about the numerical accuracy of floating-point operations, see section 8.)

    Because `bfloat` and `half` are not implicitly convertible to each other, the operators do not support mixing `bfloat` and `half`.

2. The modulus operator (%) acts upon scalar and vector integer data type operands. The modulus operator returns a result of the same built-in type as the type of the operands, after the usual arithmetic conversions. The following cases are valid:

    - If the two operands of the modulus operator are scalars, the result of the operation is a scalar.
    - If one operand is a scalar, and the other is a vector:
        - The scalar converts to the element type of the vector operand.

- The scalar type then widens to a vector that has the same number of components as the vector operand.
- Metal performs the operation componentwise, which results in a same-size vector.

- If the two operands are vectors of the same size, Metal performs the operation componentwise, which results in a same-size vector.

For any component computed with a second operand that is zero, the modulus operator result is undefined. If one or both operands are negative, the results are undefined. Results for other components with nonzero operands remain defined.

If both operands are nonnegative, the remainder is nonnegative.

3. The arithmetic unary operators (`+` and `–`) act upon scalar and vector, integer, and floating-point type operands.
4. The arithmetic post- and pre-increment and decrement operators (`--` and `++`) have scalar and vector integer type operands. All unary operators work componentwise on their operands. The result is the same type as the operand. For post- and pre-increment and decrement, the expression needs to be assignable to an `lvalue`. Pre-increment and predecrement add or subtract 1 to the contents of the expression on which they operate, and the value of the pre-increment or predecrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 to the contents of the expression on which they operate, but the resulting expression has the expression's value before execution of the post-increment or post-decrement.
5. The relational operators [greater-than (`>`), less-than (`<`), greater-than or equal to (`>=`), and less-than or equal to (`<=`)] act upon scalar and vector, integer, and floating-point type operands. The result is a Boolean (`bool` type) scalar or vector. After converting the operand type, the following cases are valid:

   - If the two operands of the relational operator are scalars, the result of the operation is a Boolean.
   - If one operand is a scalar, and the other is a vector:

     - The scalar converts to the element type of the vector operand.
     - The scalar type then widens to a vector that has the same number of components as the vector operand.
     - Metal performs the operation componentwise, which results in a Boolean vector.

   - If the two operands are vectors of the same size, Metal performs the operation componentwise, which results in a same-size Boolean vector.

   If either argument is a NaN, the relational operator returns `false`. To test a relational operation on any or all elements of a vector, use the `any` and `all` built-in functions in the context of an `if(…)` statement. (For more about `any` and `all` functions, see section 6.5.)
6. The equality operators, equal (`==`) and not equal (`!=`), act upon scalar and vector, integer and floating-point type operands. All equality operators result in a Boolean scalar or vector. After converting the operand type, the following cases are valid:

- If the two operands of the equality operator are scalars, the result of the operation is a Boolean.
- If one operand is a scalar, and the other is a vector:
  - The scalar converts to the element type of the vector operand.
  - The scalar type then widens to a vector that has the same number of components as the vector operand.
  - Metal performs the operation componentwise, which results in a Boolean vector.
- If the two operands are vectors of the same size, Metal performs the operation componentwise, which results in a same-size Boolean vector.

All other cases of implicit conversions are illegal. If one or both arguments is NaN, the equality operator equal (==) returns `false`. If one or both arguments is NaN, the equality operator not equal (`!=`) returns `true`.

7. The bitwise operators [and (&), or (|), exclusive or (^), not (~)] can act upon all scalar and vector built-in type operands, except the built-in scalar and vector floating-point types.
   - For built-in vector types, Metal applies the bitwise operators componentwise.
   - If one operand is a scalar and the other is a vector,
     - The scalar converts to the element type used by the vector operand.
     - The scalar type then widens to a vector that has the same number of components as the vector operand.
     - Metal performs the bitwise operation componentwise resulting in a same-size vector.
8. The logical operators [and (&&), or (||)] act upon two operands that are Boolean expressions. The result is a scalar or vector Boolean.
9. The logical unary operator not (`!`) acts upon one operand that is a Boolean expression. The result is a scalar or vector Boolean.
10. The ternary selection operator (`?:`) acts upon three operands that are expressions (`exp1?exp2:exp3`). This operator evaluates the first expression `exp1`, which must result in a scalar Boolean. If the result is `true`, the second expression is evaluated; if `false`, the third expression is evaluated. Metal evaluates only one of the second and third expressions. The second and third expressions can be of any type if:
    - The types of the second and third expressions match.
    - There is a type conversion for one of the expressions that can make their types match (for more about type conversions, see section 2.12).
    - One expression is a vector and the other is a scalar, and the scalar can be widened to the same type as the vector type. The resulting matching type is the type of the entire expression.
11. The ones' complement operator (~) acts upon one operand that needs to be of a scalar or vector integer type. The result is the ones' complement of its operand.

    The right-shift (>>) and left-shift (<<) operators act upon all scalar and vector integer type operands. For built-in vector types, Metal applies the operators componentwise. For the right-shift (>>) and left-shift (<<) operators, if the first operand is a scalar, the

rightmost operand needs to be a scalar. If the first operand is a vector, the rightmost operand can be a vector or scalar.

The result of `E1 << E2` is `E1` left-shifted by the `log2(N)` least significant bits in `E2` viewed as an unsigned integer value:

- If `E1` is a scalar, `N` is the number of bits used to represent the data type of `E1`.
- Or if `E1` is a vector, `N` is the number of bits used to represent the type of `E1` elements.

For the left-shift operator, the vacated bits are filled with zeros.

The result of `E1 >> E2` is `E1` right-shifted by the `log2(N)` least significant bits in `E2` viewed as an unsigned integer value:

- If `E1` is a scalar, `N` is the number of bits used to represent the data type of `E1`.
- Or if `E1` is a vector, `N` is the number of bits used to represent the data type of `E1` elements.

For the right-shift operator, if `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the vacated bits are filled with zeros. If `E1` has a signed type and a negative value, the vacated bits are filled with ones.

12. The assignment operator behaves as described by the C++17 Specification. For the `lvalue = expression` assignment operation, if `expression` is a scalar type and `lvalue` is a vector type, the scalar converts to the element type used by the vector operand. The scalar type then widens to a vector that has the same number of components as the vector operand. Metal performs the operation componentwise, which results in a same size vector.

Other C++17 operators that are not detailed above — such as `sizeof(T)`, unary (`&`) operator, and comma (`,`) operator — behave as described in the C++17 Specification.

Unsigned integers shall obey the laws of arithmetic modulo `2n`, where `n` is the number of bits in the value representation of that particular size of integer. The result of signed integer overflow is undefined.

For integral operands the divide (`/`) operator yields the algebraic quotient with any fractional part discarded. (This is often called truncation towards zero.) If the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`.

3.2 Matrix Operators

The arithmetic operators add (`+`), subtract (`–`) operate on matrices. Both matrices must have the same numbers of rows and columns. Metal applies the operation componentwise resulting in the same size matrix. The arithmetic operator multiply (`*`) acts upon:

- a scalar and a matrix
- a matrix and a scalar
- a vector and a matrix
- a matrix and a vector
- a matrix and a matrix

If one operand is a scalar, the scalar value is multiplied to each component of the matrix resulting in the same-size matrix. A right vector operand is treated as a column vector and a left vector operand as a row vector. For vector-to-matrix, matrix-to-vector, and matrix-to-matrix multiplication, the number of columns of the left operand needs to be equal to the number of rows of the right operand. The multiply operation does a linear algebraic multiply, yielding a vector or a matrix that has the same number of rows as the left operand and the same number of columns as the right operand.

The following examples presume these vector, matrix, and scalar variables are initialized. The order of partial sums for the vector-to-matrix, matrix-to-vector, and matrix-to-matrix multiplication operations described below is undefined.

```
float3 v;
float3x3 m, n;
float a = 3.0f;
```

The matrix-to-scalar multiplication:

```
float3x3 m1 = m * a;
```

is equivalent to:

```
m1[0][0] = m[0][0] * a;
m1[0][1] = m[0][1] * a;
m1[0][2] = m[0][2] * a;
m1[1][0] = m[1][0] * a;
m1[1][1] = m[1][1] * a;
m1[1][2] = m[1][2] * a;
m1[2][0] = m[2][0] * a;
m1[2][1] = m[2][1] * a;
m1[2][2] = m[2][2] * a;
```

The vector-to-matrix multiplication:

```
float3 u = v * m;
```

is equivalent to:

```
u.x = dot(v, m[0]);
u.y = dot(v, m[1]);
u.z = dot(v, m[2]);
```

The matrix-to-vector multiplication:

```
float3 u = m * v;
```

is equivalent to:

```
u.x = m[0].x * v.x + m[1].x * v.y + m[2].x * v.z;
u.y = m[0].y * v.x + m[1].y * v.y + m[2].y * v.z;
u.z = m[0].z * v.x + m[1].z * v.y + m[2].z * v.z;
```

The matrix-to-matrix multiplication:

```
float3x3 r = m * n; // m, n are float3x3
```

is equivalent to:

```
r[0].x = m[0].x * n[0].x + m[1].x * n[0].y + m[2].x * n[0].z;
r[0].y = m[0].y * n[0].x + m[1].y * n[0].y + m[2].y * n[0].z;
r[0].z = m[0].z * n[0].x + m[1].z * n[0].y + m[2].z * n[0].z;
r[1].x = m[0].x * n[1].x + m[1].x * n[1].y + m[2].x * n[1].z;
r[1].y = m[0].y * n[1].x + m[1].y * n[1].y + m[2].y * n[1].z;
r[1].z = m[0].z * n[1].x + m[1].z * n[1].y + m[2].z * n[1].z;
r[2].x = m[0].x * n[2].x + m[1].x * n[2].y + m[2].x * n[2].z;
r[2].y = m[0].y * n[2].x + m[1].y * n[2].y + m[2].y * n[2].z;
r[2].x = m[0].z * n[2].x + m[1].z * n[2].y + m[2].z * n[2].z;
```

4 Address Spaces

The Metal memory model describes the behavior and structure of memory objects in MSL programs. An address space attribute specifies the region of memory from where buffer memory objects are allocated. These attributes describe disjoint address spaces that can also specify access restrictions:

- `device` (see section 4.1)
- `constant` (see section 4.2)
- `thread` (see section 4.3)
- `threadgroup` (see section 4.4)
- `threadgroup_imageblock` (see section 4.5)
- `ray_data` (see section 4.6)
- `object_data` (see section 4.7)

All OS: Metal 1 and later support the `device`, `threadgroup`, `constant`, and `thread` attributes. Metal 2.3 and later support `ray_data` attributes. Metal 3 and later support `object_data` attributes.
iOS: Metal 2 and later support the `threadgroup_imageblock` attribute.
macOS: Metal 2.3 and later support the `threadgroup_imageblock` attribute.
iPadOS and visionOS: Metal supports the `threadgroup_imageblock` attribute.

All arguments to a graphics or kernel function that are a pointer or reference to a type needs to be declared with an address space attribute. For graphics functions, an argument that is a pointer or reference to a type needs to be declared in the `device` or `constant` address space. For kernel functions, an argument that is a pointer or reference to a type needs to be declared in the `device`, `threadgroup`, `threadgroup_imageblock`, or `constant` address space. The following example introduces the use of several address space attributes. (The `threadgroup` attribute is supported here for the pointer `l_data` only if `foo` is called by a kernel function, as detailed in section 4.4.)

```
void foo(device int *g_data,
         threadgroup int *l_data,
         constant float *c_data)
{…}
```

The address space for a variable at program scope needs to be `constant`.

Any variable that is a pointer or reference needs to be declared with one of the address space attributes discussed in this section. If an address space attribute is missing on a pointer or reference type declaration, a compilation error occurs.

4.1 Device Address Space

The `device` address space name refers to buffer memory objects allocated from the device memory pool that are both readable and writeable.

A buffer memory object can be declared as a pointer or reference to a scalar, vector or user-defined structure. In an app, Metal API calls allocate the memory for the buffer object, which determines the actual size of the buffer memory.

Some examples are:

```
// An array of a float vector with four components.
device float4 *color;

struct Foo {
     float a[3];
     int b[2];
};


// An array of Foo elements.
device Foo *my_info;
```

Because you always allocate texture objects from the device address space, you don't need the `device` address attribute for texture types. You cannot directly access the elements of a texture object, so use the built-in functions to read from and write to a texture object (see section 6.13).

4.2 Constant Address Space

The `constant` address space name refers to buffer memory objects allocated from the device memory pool that are read-only. You must declare variables in program scope in the `constant` address space and initialize them during the declaration statement. The initializer(s) expression must be a core constant expression. (Refer to section 5.20 of the C++17 specification.) The compiler may evaluate a core constant expression at compile time. Variables in program scope have the same lifetime as the program, and their values persist between calls to any of the compute or graphics functions in the program.

```
constant float samples[] = { 1.0f, 2.0f, 3.0f, 4.0f };
```

Pointers or references to the `constant` address space are allowed as arguments to functions.

Writing to variables declared in the `constant` address space is a compile-time error. Declaring such a variable without initialization is also a compile-time error.

Buffers in the `constant` address space passed to kernel, vertex, and fragment functions have minimum alignment requirements based on the GPU. See "Minimum constant buffer offset alignment" in the Metal Feature Set Tables for more information.

4.3 Thread Address Space

The `thread` address space refers to the per-thread memory address space. Variables allocated in this address space are not visible to other threads. Variables declared inside a graphics or kernel function are allocated in the `thread` address space.

```
[[kernel]] void
my_kernel(…)
{
     // A float allocated in the per-thread address space
     float x;

     // A pointer to variable x in per-thread address space
     thread float * p = &x;
     …
}
```

4.4 Threadgroup Address Space

A GPU compute unit can execute multiple threads concurrently in a *threadgroup*, and a GPU can execute a separate threadgroup for each of its compute units.

Threads in a threadgroup can work together by sharing data in `threadgroup` memory, which is faster on most devices than sharing data in `device` memory. Use the `threadgroup` address space to:

- Allocate a threadgroup variable in a kernel, mesh, or object function.
- Define a kernel, fragment, or object function parameter that's a pointer to a threadgroup address.

See the Metal Feature Set Tables to learn which GPUs support `threadgroup` space arguments for fragment shaders.

Threadgroup variables in a kernel, mesh, or object function only exist for the lifetime of the threadgroup that executes the kernel. Threadgroup variables in a mid-render kernel function are persistent across mid-render and fragment kernel functions over a tile.

This example kernel demonstrates how to declare both variables and arguments in the `threadgroup` address space. (The `[[threadgroup]]` attribute in the code below is explained in section 5.2.1.)

```
kernel void
my_kernel(threadgroup float *sharedParameter [[threadgroup(0)]],
          …)
{
     // Allocate a float in the threadgroup address space.
     threadgroup float sharedFloat;
```

```
    // Allocate an array of 10 floats in the threadgroup address
    space.
    threadgroup float sharedFloatArray[10];
    ...
}
```

For more information about the `[[threadgroup(0)]]` attribute, see section 5.2.1.

4.4.1 SIMD-Groups and Quad-Groups

macOS: Metal 2 and later support SIMD-group functions. Metal 2.1 and later support quad-group functions.
iOS: Metal 2 and later support quad-group functions. Metal 2.2 and later support some SIMD-group functions.
iPadOS and visionOS: Metal supports SIMD-group functions and support quad-group functions.

Within a threadgroup, you can divide threads into *SIMD-groups*, which are collections of threads that execute concurrently. The mapping to SIMD-groups is invariant for the duration of a kernel's execution, across dispatches of a given kernel with the same launch parameters, and from one threadgroup to another within the dispatch (excluding the trailing edge threadgroups in the presence of nonuniform threadgroup sizes). In addition, all SIMD-groups within a threadgroup needs to be the same size, apart from the SIMD-group with the maximum index, which may be smaller, if the size of the threadgroup is not evenly divisible by the size of the SIMD-groups.

A *quad-group* is a SIMD-group with the thread execution width of 4.

For more about kernel function attributes for SIMD-groups and quad-groups, see section 5.2.3.6. For more about threads and thread synchronization, see section 6.10 and its subsections:

- For more about thread synchronization functions, including a SIMD-group barrier, see section 6.10.1.
- For more about SIMD-group functions, see section 6.10.2.
- For more about quad-group functions, see section 6.10.3.

4.5 Threadgroup Imageblock Address Space

The `threadgroup_imageblock` address space refers to objects allocated in threadgroup memory that are only accessible using an `imageblock<T, L>` object (see section 2.11). A pointer to a user-defined type allocated in the `threadgroup_imageblock` address space can be an argument to a tile shading function (see section 5.1.9). There is exactly one threadgroup per tile, and each threadgroup can access the threadgroup memory and the imageblock associated with its tile.

- Variables allocated in the `threadgroup_imageblock` address space in a kernel function are allocated for each threadgroup executing the kernel, are shared by all threads in a threadgroup, and exist only for the lifetime of the threadgroup that executes the kernel. Each thread in the threadgroup uses explicit 2D coordinates to access imageblocks. Do not

assume any spatial relationship between the threads and the imageblock. The threadgroup dimensions may be smaller than the tile size.

4.6 Ray Data Address Space

All OS: Metal 2.3 and later support `ray_data` address space.

The `ray_data` address space refers to objects allocated in a memory that is only accessible in an intersection function (see section 5.1.6) for ray tracing. Intersection functions can read and write to a custom payload using `[[payload]]` attribute (see Table 5.10) in the `ray_data` address space. When a shader calls `intersect()` (see section 6.19.2) with a payload, the system copies the payload to the `ray_data` address space, calls the intersection function, and when the intersection function returns, it copies the payload back out.

4.7 Object Data Address Space

All OS: Metal 3 and later support `object_data` address space.

Object functions use the `object_data` address space to pass a payload to a mesh function (see section 5.2.3.9). The `object_data` address space behaves like the `threadgroup` address space in that the programming model is explicitly cooperative within the threadgroup. Use the threads in the threadgroup to efficiently compute the payload and value for `mesh_grid_properties::set_threadgroups_per_grid` (see section 2.20.1). The payload in the `object_data` address space is not explicitly bound or initialized, and the implementation manages its lifetime. Without any additional synchronization, all writes to `object_data` inside a `threadgroup` of an `object` function are visible to the `threadgroup` of a `mesh` function that the object function launches.

4.8 Memory Coherency

All OS: Metal 3.2 and later support `coherent(device)` qualifier and `memory_coherence` on textures for Apple silicon.

Memory operations in Metal have a concept of a scope of coherency. For a store, the scope of coherence describes the set of threads that may observe the result of the store if you properly synchronize them, and for a load, it describes the set of threads with stores the load may observe if you properly synchronize them. Metal has the following scope of coherence:

- Thread coherence — memory writes are only visible to the thread.
- Threadgroup coherence — memory writes are only visible to threads within their threadgroup.
- Device coherence — memory writes are visible to all threads on the device, that is, threads across threadgroups.

Memory in the `thread` address space has thread coherence, and memory in the `threadgroup` address space has threadgroup coherence. By default, memory in the `device` address space has threadgroup coherence.

Metal 3.2 and later support the `coherent(device)` qualifiers for buffers and `memory_coherence_device` for textures to indicate that the object has device coherence, that is, memory operations are visible across threads on the device if you properly synchronize them.

```
[[kernel]] void example(
  coherent device float          *dptr1,
  coherent(device) device float4 *dptr2,
  texture2d<float, access::read, memory_coherence_device>     tex,
  texture2d<float, access::read,
                   memory_coherence::memory_coherence_device> tex2)
{…}
```

5 Function and Variable Declarations

This chapter describes how you declare functions, arguments, and variables. It also details how you often use attributes to specify restrictions to functions, arguments, and variables.

5.1 Functions

Metal 1 and later support the `kernel`, `vertex`, and `fragment` attributes for every OS. Metal 2.3 and later support the C++ attributes:

- `[[vertex]]` or `vertex` (See section 5.1.1)
- `[[fragment]]` or `fragment` (See section 5.1.2)
- `[[kernel]]` or `kernel` (See section 5.1.3)
- `[[visible]]` (See section 5.1.4)
- `[[stitchable]]` (See section 5.1.5)
- `[[intersection(…)]]` (See section 5.1.6)
- `[[object]]` (See section 5.1.7)
- `[[mesh]]` (See section 5.1.8)

Make a function accessible to the Metal API by adding one of these function attributes at the start of a function, which makes it a *qualified* function. Kernel, vertex, and fragment functions can't call one another without triggering a compilation error, but they may call other functions that use the `[[visible]]` attribute. They can also call functions with the `[[intersection(…)]]` attribute by calling `intersect()` (see section 6.19.2).

In Metal 2.1 and earlier, the Metal compiler ignores namespace identifiers for kernel, vertex, and fragment functions. In Metal 2.2 and later, if you declare a qualified function within a namespace, you must include the namespace identifier with the function's name each time you refer it to a Metal Framework API. This example declares two kernel functions in different namespaces.

```
namespace outer {
  [[kernel]] void functionA() {…}
  namespace inner {
    [[kernel]] void functionB() {…}
  }
}
```

Refer to a function in a namespace by prepending the function's name with the namespace's identifier followed by two colons:

```
Outer::functionA
```

Similarly, refer to a function in a nested namespace by prepending the function's name with all namespaces in order and separating each with two colons:

```
Outer::inner::functionB
```

5.1.1 Vertex Functions

You can declare the `vertex` (or in Metal 2.3 and later, `[[vertex]])` attribute only for a graphics function. Metal executes a vertex function for each vertex in the vertex stream and generates per-vertex output. The following example shows the syntax for declaring a vertex:

```
vertex void
my_vertex_func(…)
{…}

[[vertex]] void
vertex_func2(…)
{…}
```

For a vertex function, the return type identifies the output generated by the function. If the vertex function does not generate output, it shall return `void` and can only be used in a render pipeline with rasterization disabled.

5.1.1.1 Post-Tessellation Vertex Functions

All OS: Metal 1.2 and later support post-tessellation vertex functions (`patch` attribute).

The post-tessellation vertex function calculates the vertex data for each surface sample on the patch produced by the fixed function tessellator. The inputs to the post-tessellation vertex function are:

- Per-patch data.
- Patch control point data.
- The tessellator stage output (the normalized vertex location on the patch).

The post-tessellation vertex function generates the final vertex data for the tessellated triangles. For example, to add additional detail (such as displacement mapping values) to the rendered geometry, the post-tessellation vertex function can sample a texture to modify the vertex position by a displacement value.

After the post-tessellation vertex function executes, the tessellated primitives rasterize.

The post-tessellation vertex function is a vertex function identified using the ordinary `vertex` function attribute.

5.1.1.2 Patch Type and Number of Control Points Per-Patch

The `[[patch]]` attribute is required for the post-tessellation vertex function.

For macOS, the `[[patch(patch-type, N)]]` attribute must specify both the patch type (`patch-type` is either `quad` or `triangle`) and the number of control points in the patch (`N` needs to be a value from `0` to `32`). For iOS, specifying the `patch-type` is required, but the number of control points is optional.

If the number of control points are specified in the post-tessellation vertex function, this number must match the number of control points provided to the `drawPatches` or `drawIndexedPatches` API.

Example:

```
[[patch(quad)]]
[[vertex]] vertex_output
my_post_tessellation_vertex(…)
{…}

[[patch(quad, 16)]]
[[vertex]] vertex_output
my_bezier_vertex(…)
{…}
```

5.1.2 Fragment Functions

You can declare the `fragment` or since Metal 2.3 `[[fragment]]` attribute only for a graphics function. Metal executes a fragment function for each fragment in the fragment stream and their associated data and generates per-fragment output. The following example shows the syntax for declaring a fragment function with the `fragment` attribute:

```
[[fragment]]
void my_fragment_func(…)
{…}

fragment
void my_fragment_func2(…)
{…}
```

For graphics functions, the return type identifies whether the output generated by the function is either per-vertex or per-fragment. If the fragment function does not generate output, it returns `void`.

To request performing fragment tests before the fragment function executes, use the `[[early_fragment_tests]]` function attribute with a fragment function, as shown in the example below.

```
[[early_fragment_tests]]
fragment float4
my_fragment( … )
{…}
```

It is an error if the return type of the fragment function declared with the `[[early_fragment_tests]]` attribute includes a depth or stencil value; that is, if the return type of this fragment function includes an element declared with the `[[depth(depth_argument)]]` or `[[stencil]]` attribute.

It is an error to use the `[[early_fragment_tests]]` attribute with any function that is not a fragment function; that is, not declared with the `fragment` attribute.

5.1.3 Compute Functions (Kernels)

A compute function (also called a *kernel*) is a data-parallel function that is executed over a 1-, 2-, or 3D grid. The following example shows the syntax for declaring a compute function with the `kernel` or since Metal 2.3 `[[kernel]]` attribute:

```
[[kernel]]
void my_kernel(…) {…}

kernel
void my_kernel2(…) {…}
```

Functions declared with the `kernel` or `[[kernel]]` attribute must return `void`.

You can use the `[[max_total_threads_per_threadgroup]]` function attribute with a kernel function to specify the maximum threads per threadgroup. The value must fit within 32 bits.

Below is an example of a kernel function that uses this attribute:

```
[[max_total_threads_per_threadgroup(x)]]
kernel void
my_kernel(…)
{…}
```

If the `[[max_total_threads_per_threadgroup]]` value is greater than the `[MTLDevice maxThreadsPerThreadgroup]` property, then compute pipeline state creation fails.

In Metal 4 and later, you can use the `[[required_threads_per_threadgroup]]` function attribute with a kernel function to specify the number of threads per threadgroup. The value must fit within 32 bits. If the `[[required_threads_per_threadgroup]]` value is set and the `[MTLDevice requiredThreadsPerThreadgroup]` property is set, the values must be the same; otherwise, the compute pipeline state creation fails.

5.1.4 Visible Functions

All OS: Metal 2.3 and later support `[[visible]]` functions.

A function with a `[[visible]]` attribute is a function that's visible from the Metal framework API; that is, you can get a `MTLFunction` object of this function. It is legal to take the address of a visible function and get a visible function pointer. You can use the visible function pointers with the `visible_function_table` type (section 2.15). It is legal for other functions to directly call a `visible` function. Note that `visible` function, like other *qualified* functions, is split into their own translation unit. When a function directly calls a `visible` function, pass it in the pipeline descriptor.

The following example with the `[[visible]]` attribute:

```
[[visible]] float my_visible(device int *data, int data_offset) {…}
```

5.1.5 Stitchable Functions

All OS: Metal 2.4 and later support `[[stitchable]]` functions.

A function with a `[[stitchable]]` attribute is a function that can be used in the `MTLFunctionStitchingGraph`. The `[[stitchable]]` attribute implies `[[visible]]`, which means that stitchable functions can be used in all contexts where a visible function can be used as described in Sec 5.1.4. The compiler generates additional metadata for stitchable functions to enable these functions to be used in the `MTLFunctionStitchingGraph`. Note that the metadata will increase the code size of this function.:

```
[[stitchable]] float my_func(device float *data, texture2d<float>
tex) {…}
```

5.1.6 Intersection Functions

All OS: Metal 2.3 and later support `[[intersection(primitive_type, intersection_tags…]]` functions.

You can declare a custom intersection function to use with ray tracing by using the `[[intersection(primitive_type, intersection_tags…)]]` attribute. Metal calls intersection functions when the shader calls `intersect()` (see section 6.19) to determine if a potential ray intersection is valid or if traversal should continue. Note that intersection functions can't start new rays. Table 5.1. Intersection function primitive types lists the intersection types Metal supports.

Table 5.1. Intersection function primitive types

Primitive type	Description
`triangle`	Indicates that this is an intersection function that extends the default triangle intersection test.
`bounding_box`	Indicates that this is an intersection function which is run when a ray intersects the bounding box.
`curve` All OS: Metal 3.1 and later.	Indicates that this is an intersection function that extends the default curve intersection test.

You may pass zero or more intersection tags as described in Table 2.9 from section 2.17. Some examples are:

```
[[intersection(triangle, triangle_data, instancing,
  world_space_data)]]
bool triangleIntersectionFunction(...) {…}

[[intersection(bounding_box, triangle_data, instancing,
  world_space_data)]]
UserResult boundingBoxIntersectionFunction(...) {…}
```

The intersection function `primitive_type` and `intersection_tags` control the allowable input and output attributes (see Section 5.2.3.7).

Intersection functions support passing buffer arguments from `device` and `constant` address space.

Intersection functions don't support passing texture arguments to an intersection function. However, you can pass a texture using an argument buffer.

Intersection functions don't support threadgroup memory.

Intersection functions don't support `threadgroup_barrier` or `simdgroup_barrier`. If they are used, the result is undefined.

Intersection functions may or may not be run in the same SIMD-group as the thread which launched the intersection operation: The implementation is permitted to regroup or repack candidate intersections to improve efficiency before launching SIMD-groups to do intersection testing.

If the acceleration structure traversal finds a procedural box primitive, and the intersection function is a triangle tester (or vice versa), this is an application error and behavior is undefined.

.

5.1.7 Object Functions

All OS: Metal 3 and later support `[[object]]` functions.

A function with an `[[object]]` attribute is an object function in the mesh pipeline. An object function is a data-parallel function executed over a 1-, 2-, or 3D compute grid that can launch compute grids to a second mesh stage and with a data payload. Object functions must return `void`.

Input built-in variables to object functions are described in section 5.2.3.9. The `[[payload]]` attribute tags a buffer that the object function exports to the mesh shader as a read-only buffer. It may be specified once per function.

You can use the `[[max_total_threads_per_threadgroup]]` function attribute with an object function to specify the maximum threads per threadgroup. The value must fit within 32 bits.

You can use the `[[max_total_threadgroups_per_mesh_grid(size)]]` on an object function to specify the maximum threadgroups per mesh grid. The following is an example using the `[[object]]` attribute.

```
#define kMeshThreadgroups 32
struct ObjectOutput {
  // User-defined payload; one entry for each mesh threadgroup.
  // This is an array because the data is shared by the mesh grid.
  float value[kMeshThreadgroups];
};

[[object, max_total_threadgroups_per_mesh_grid(kMeshThreadgroups)]]
void objectShader(uint threadgroup_size [[threads_per_threadgroup]],
                  uint lane [[thread_index_in_threadgroup]],
                  object_data ObjectOutput& output [[payload]],
                  mesh_grid_properties mgp) {…}
```

5.1.8 Mesh Functions

All OS: Metal 3 and later support `[[mesh]]` functions.

A function with a `[[mesh]]` attribute is a mesh function in the mesh pipeline. A mesh function is a data-parallel function that can optionally export a mesh object representing a chunk of geometry to the rasterization pipeline. The mesh object is a parameter of the mesh function. If no mesh object is exported, rasterization is disabled. Input built-in variables to mesh functions are described in section 5.2.3.10. Mesh functions must return `void`.

You can use the `[[max_total_threads_per_threadgroup]]` function attribute with a mesh function to specify the maximum threads per threadgroup. The value must fit within 32 bits. The following is an example using the `[[mesh]]` attribute:

```
struct vertex_t {
    float4 clip_pos [[position]];
    float3 world_pos;
    float3 color;
    // other user-defined properties
};
struct primitive_t {
    float3 normal;
};

// A mesh declaration that can export one cube.
using cube_mesh_t = metal::mesh<vertex_t, primitive_t,
                                8 /*corners*/,
                                6*2 /*faces*/,
                                metal::topology::triangle>;
struct view_info_t {
    float4x4 view_proj;
};
struct cube_info_t {
    float4x3 world;
    float3 color;
};

[[mesh, max_total_threads_per_threadgroup(12)]]
void cube_stage(cube_mesh_t output,
                const object_data cube_info_t &cube [[payload]],
                constant view_info_t &view [[buffer(0)]],
                uint gid [[threadgroup_position_in_grid]],
                uint lane [[thread_index_in_threadgroup]]) {…}
```

5.1.9 Tile Functions

iOS: Metal 2 and later support tile functions.
macOS: Metal 2.3 and later support tile functions.
iPadOS and visionOS: Metal supports tile functions.

A *tile shading function* is a special type of compute kernel or fragment function that can execute inline with graphics operations and take advantage of the Tile-Based Deferred Rendering (TBDR) architecture. With TBDR, commands are buffered until a large list of commands accumulates. The hardware divides the framebuffer into tiles and then renders only the primitives that are visible within each tile. Tile shading functions support performing compute operations in the middle of rendering, which can access memory more efficiently by reducing round trips to memory and utilizing high-bandwidth local memory.

A tile function launches a set of threads called a *dispatch*, which is organized into threadgroups and grids. You may launch threads at any point in a render pass and as often as needed. Tile functions barrier against previous and subsequent draws, so a tile function does not execute

until all earlier draws have completed. Likewise, later draws do not execute until the tile function completes.

GPUs always process each tile and each dispatch to completion. Before processing the next tile, all draws and dispatches for a tile launch in submission.

Tile functions have access to 32 KB of threadgroup memory that may be divided between imageblock storage and threadgroup storage. (For more information about the threadgroup memory size, see section 4.4.) The imageblock size is dependent on the tile width, tile height, and the bit depth of each sample. Either the render pass attachments (which use implicit imageblock layout; see section 5.6.3.1) or function-declared structures (which use explicit imageblock layout; see section 5.6.3.2) determines the bit depth of the sample. For more about how kernel functions utilize the `threadgroup_imageblock` address space, see section 4.5.

5.1.10 Host Name Attribute

All OS: Metal 2.2 and later support the host name attribute.

In Metal 2.2 and later, you can override the default name that the Metal Framework API uses to refer to a qualified function. Add the `[[host_name(name)]]` attribute to the function declaration, where name is the string literal that the Metal Framework API will use to reference the function name. The compiler raises a compile time error if you give different functions the same name. For example:

```
// Metal API name is abc
[[host_name("abc")]] [[kernel]] void funcA() {}

// Metal API name is xyz
[[host_name("xyz")]] [[kernel]] void funcX() {}
```

5.1.11 Templated Qualified Functions

All OS: Metal 2.2 and later support the template qualified functions.

In Metal 2.2 and later, you can use templates for qualified functions (for example, vertex, fragment, visible, and kernel functions) declarations. You must explicitly instantiate the template to force the compiler to emit code for a given specialization. For example:

```
template<typename T>
kernel void bar(device T *x) { … }
// Explicit specialization of `bar<T>` with [T = int]
template kernel void bar(device int *);
```

The compiler gives all specializations the same name unless one uses the `[[host_name(name)]]` attribute to provide a different name for each specialization.

```
// Explicit specialization of `bar<T>` with [T = int] and host_name
// "bar_int".
template [[host_name("bar_int")]] kernel void bar(device int *);

// Explicit specialization of `bar<T>` with [T = float] and
// host_name "bar_float".
template [[host_name("bar_float")]] kernel void bar(device float *);
```

5.1.12 User Annotation Attribute

All OS: Metal 4 and later support the user annotation attribute.

You can annotate a qualified function with a name and look it up using the Metal Framework reflection API. Add the `[[user_annotation("string")]]` attribute to a qualified function where string is the annotation you want to associate with a function. When you add a `user_annotation` attribute to a templated qualified function, all instantiations inherit that annotation unless you override it using a `user_annotation` attribute on that instantiation. For example:

```
[[user_annotation("basecase"), kernel]] void funcB() {…}

template<typename T> [[user_annotation("Tcase"), kernel]]
void funcTmpl(device T *x) {…}

// Inherit from user_annotation.
template [[host_name("funcImplInt"), kernel]]
void funcImpl(device int* x) {…}

// Override user_annotation.
template [[host_name("funcImplFloat"),
           user_annotation("FPoverride"),
           kernel]]]]
void funcImpl(device float* x) {…}
```

When looking up the annotation for the `funcB` using the Metal Framework API, you get back `basecase`. For `funcImplInt`, you get back `Tcase`, and for `funcImplFloat`, you get back `FPoverride`.

5.2 Function Arguments and Variables

Most inputs and outputs to graphics (vertex or fragment) and kernel functions are passed as arguments. (Initialized variables in the constant address space and samplers declared in program scope are inputs and outputs that do not have to be passed as arguments.)

In Metal 3.1 and later provide built-in input variables for kernel, mesh, and object shaders that you declare in program scope, avoiding the need for passing them as arguments. This applies if

you don't use them in a dynamic library or a separately compiled binary function. In Metal 3.2 and later provide built-in input variables that you can also use in a dynamic library or a separately compiled binary functions for Apple silicon.

In Metal 3.2 and later, you can declare `device`, `constant`, and `threadgroup` buffers, `texture`, and `sampler` in the program scope (see section 5.9). Unlike when passing as arguments in a shader, you can't assume different global variables are non-aliased. You need to specify the binding indexes because Metal can't set them automatically.

Arguments to graphics and kernel functions can be any of the following:

- Device buffer — A pointer or reference to any data type in the `device` address space (see section 2.8).
- Constant buffer — A pointer or reference to any data type in the `constant` address space (see section 2.8).
- A `texture` object (see section 2.9) or an array of textures.
- A `texture_buffer` object (see section 2.9.1) or an array of texture buffers.
- A `sampler` object (see section 2.10) or an array of samplers.
- A buffer shared between threads in a threadgroup — a pointer to a type in the `threadgroup` address space that can only be used as arguments for kernel functions.
- An imageblock (see section 2.11).
- An argument buffer (see section 2.13).
- A visible function table (see section 2.15) for kernel functions. In Metal 2.4 and later, visible function table can also be used in graphic functions.
- An intersection function table (see section 2.17.3) for kernel functions.
- An acceleration structure (see section 6.19.1) for intersection functions.
- A `tensor` (see section 2.22).
- A structure with elements that are buffers, textures, or texture buffers.

Buffers (device) specified as argument values to a graphics or kernel function cannot alias; that is, a buffer passed as an argument value cannot overlap another buffer passed to a separate argument of the same graphics or kernel function.

You cannot declare arguments to graphics and kernel functions to be of type `size_t`, `ptrdiff_t`, or a structure and/or union that contains members declared to be one of these built-in scalar types.

The arguments to these functions are often specified with attributes to provide further guidance on their use. Attributes are used to specify:

- The resource location for the argument (see section 5.2.1).
- Built-in variables that support communicating data between fixed-function and programmable pipeline stages (see section 5.2.3).
- Which data is sent down the pipeline from vertex function to fragment function (see section 5.2.4).

5.2.1 Locating Buffer, Texture, and Sampler Arguments

For each argument, an attribute can be optionally specified to identify the location of a buffer, texture, or sampler to use for this argument type. The Metal framework API uses this attribute to identify the location for these argument types.

- Device buffers, constant buffers, `acceleration_struct<…>`, `intersection_function_table<…>`, and `tensors`: `[[buffer(index)]]`
- Textures (including texture buffers): `[[texture(index)]]`
- Samplers: `[[sampler(index)]]`
- Threadgroup buffers: `[[threadgroup(index)]]`

The `index` value is an unsigned integer that identifies the location of an assigned buffer, texture or sampler argument. (A texture buffer is a specific type of texture.) The proper syntax is for the attribute to follow the argument or variable name.

The example below is a simple kernel function, `add_vectors`, that adds an array of two buffers in the device address space, `inA` and `inB`, and returns the result in the buffer out. The attributes (`buffer(index)`) specify the buffer locations for the function arguments.

```
[[kernel]] void
add_vectors(const device float4 *inA [[buffer(0)]],
            const device float4 *inB [[buffer(1)]],
            device float4 *out [[buffer(2)]],
            uint id [[thread_position_in_grid]])
{
     out[id] = inA[id] + inB[id];
}
```

The example below shows attributes used for function arguments of several different types (a buffer, a texture, and a sampler):

```
[[kernel]] void
my_kernel(device float4 *p [[buffer(0)]],
          texture2d<float> img [[texture(0)]],
          sampler sam [[sampler(1)]])
{…}
```

If the location indices are not specified, the Metal compiler assigns them using the first available location index. In the following example, `src` is assigned texture index `0`, `dst` texture index `1`, `s` sampler index `0`, and `u` buffer index `0`:

```
kernel void
my_kernel(texture2d<half> src,
              texture2d<half, access::write> dst,
              sampler s,
              device myUserInfo *u)
{…}
```

In the following example, some kernel arguments have explicitly assigned location indices and some don’t. `src` is explicitly assigned texture index `0`, and `f` is explicitly assigned buffer index `10`. If you assign location indices using function constants (section 5.8), the compiler doesn’t

consider those entries when assigning indices. The other arguments are assigned the first available location index: `dst` texture index `1`, `s` sampler index `0`, and `u` buffer index `0`.

```
kernel void
my_kernel(texture2d<half> src [[texture(0)]],
          texture2d<half, access::write> dst,
          sampler s,
          device myUserInfo *u,
          device float *f [[buffer(10)]])
{…}
```

Each attribute (`buffer`, `threadgroup`, `texture`, and `sampler`) represents a group of resources. The `index` values specified on the arguments shall be unique within each resource group. Multiple buffer, texture or sampler arguments with the same `index` value generate a compilation error unless they are declared with a function constant attribute (see section 5.8.1).

5.2.1.1 Vertex Function Example with Resources and Outputs to Device Memory

The following example is a vertex function, `render_vertex`, which outputs to device memory in the array `xform_output`, which is a function argument specified with the `device` attribute (introduced in section 4.1). All the `render_vertex` function arguments are specified with the `buffer(0)`, `buffer(1)`, `buffer(2)`, and `buffer(3)` attributes (introduced in section 5.2.1). For more about the `position` attribute shown in this example, see section 5.2.3.3.

```
struct VertexOutput {
     float4 position [[position]];
     float4 color;
     float2 texcoord;
};

struct VertexInput {
     float4 position;
     float3 normal;
     float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
     uint   num_lights;
     float4 light_position[MAX_LIGHTS];
     float4 light_color[MAX_LIGHTS];
     float4 light_attenuation_factors[MAX_LIGHTS];
};

vertex void
render_vertex(const device VertexInput* v_in [[buffer(0)]],
              constant float4x4& mvp_matrix [[buffer(1)]],
              constant LightDesc& light_desc [[buffer(2)]],
```

```
            device VertexOutput* xform_output [[buffer(3)]],
            uint v_id [[vertex_id]] )
{
    VertexOutput v_out;
    v_out.position = v_in[v_id].position * mvp_matrix;
    v_out.color = do_lighting(v_in[v_id].position,
    v_in[v_id].normal, light_desc);

    v_out.texcoord = v_in[v_id].texcoord;

    // Output the position to a buffer.
    xform_output[v_id] = v_out;
}
```

5.2.1.2 Raster Order Groups

All OS: Metal 2 and later support raster order group attributes.

Loads and stores to buffers (in `device` memory) and textures in a fragment function are unordered. The `[[raster_order_group(index)]]` attribute used for a buffer or texture guarantees the order of accesses for any overlapping fragments from different primitives that map to the same `(x,y)` pixel coordinate and sample, if per-sample shading is active.

The `[[raster_order_group(index)]]` attribute can be specified on a texture (which is always in `device` memory) or a buffer that is declared in `device` memory, but not in either the `threadgroup` or `constant` address space. The `[[raster_order_group(index)]]` attribute cannot be used with a structure or class.

Fragment function invocations that mark overlapping accesses to a buffer or texture with the `[[raster_order_group(index)]]` attribute are executed in the same order as the geometry is submitted. For overlapping fragment function invocations, writes performed by a fragment function invocation to a buffer or texture marked with the `[[raster_order_group(index)]]` attribute needs to be available to be read by a subsequent invocation and must not affect reads by a previous invocation. Similarly, reads performed by a fragment function invocation must reflect writes by a previous invocation and must not reflect writes by a subsequent invocation.

The `index` in `[[raster_order_group(index)]]` is an integer value that specifies a rasterizer order ID, which provides finer grained control over the ordering of loads and stores. For example, if two buffers A and B are marked with different rasterizer order ID values, then loads and stores to buffers A and B for overlapping fragments can be synchronized independently.

Example:

```
fragment void
my_fragment(texture2d<float, access::read_write> texA
                              [[raster_order_group(0), texture(0)]],
…)
{
    ushort2 coord;
    float4 clr = texA.read(coord);
```

```
    // do operations on clr
    clr = …;
    texA.write(clr, coord);
}
```

For an argument buffer, you can use the `[[raster_order_group(index)]]` attribute on a buffer or texture member in a structure.

5.2.2 Attributes to Locate Per-Vertex Inputs

A vertex function can read per-vertex inputs by indexing into a buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. In addition, you can also declare per-vertex input with the `[[stage_in]]` attribute and pass that input as an argument. For per-vertex input passed as an argument declared with the `[[stage_in]]` attribute, each element of the per-vertex input must specify the vertex attribute location as `[[attribute(index)]]`. For more about the `[[stage_in]]` attribute, see section 5.2.4.

The `index` value is an unsigned integer that identifies the assigned vertex input location. The proper syntax is for the attribute to follow the argument or variable name. The Metal API uses this attribute to identify the location of the vertex buffer and describe the vertex data such as the buffer to fetch the per-vertex data from, its data format, and its stride.

The following example shows how to assign vertex attributes to elements of a vertex input structure that is passed to a vertex function using the `stage_in` attribute:

```
struct VertexInput {
    float4 position [[attribute(0)]];
    float3 normal   [[attribute(1)]];
    half4 color     [[attribute(2)]];
    half2 texcoord  [[attribute(3)]];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput
render_vertex(VertexInput v_in [[stage_in]],
              constant float4x4& mvp_matrix [[buffer(1)]],
              constant LightDesc& lights [[buffer(2)]],
              uint v_id [[vertex_id]])
```

```
{
     VertexOutput v_out;
     …
     return v_out;
}
```

The example below shows how both buffers and the `stage_in` attribute can be used to fetch per-vertex inputs in a vertex function:

```
struct VertexInput {
     float4 position [[attribute(0)]];
     float3 normal   [[attribute(1)]];
};

struct VertexInput2 {
     half4 color;
     half2 texcoord[4];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
     uint   num_lights;
     float4 light_position[MAX_LIGHTS];
     float4 light_color[MAX_LIGHTS];
     float4 light_attenuation_factors[MAX_LIGHTS];
};

constexpr sampler s = sampler(coord::normalized,
address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput
render_vertex(VertexInput v_in [[stage_in]],
              VertexInput2 v_in2 [[buffer(0)]],
              constant float4x4& mvp_matrix [[buffer(1)]],
              constant LightDesc& lights [[buffer(2)]],
              uint v_id [[vertex_id]])
{
     VertexOutput vOut;
     …
     return vOut;
}
```

A post-tessellation vertex function can read the per-patch and patch control-point data. The post-tessellation vertex function specifies the patch control-point data as the following templated type:

```
patch_control_point<T>
```

Where `T` is a user defined structure. Each element of `T` must specify an attribute location using `[[attribute(index)]]`.

All OS: Metal 1.2 and later support patch control-point templated types.

The `patch_control_point<T>` type supports these member functions:

- `constexpr size_t size() const;`, which returns the number of control-points in the patch.
- `constexpr const_reference operator[] (size_t pos) const;`, which returns the data for a specific patch control point that `pos` identifies.

Example:

```
struct ControlPoint {
     int3 patchParam  [[attribute(0)]];
     float3 P         [[attribute(1)]];
     float3 P1        [[attribute(2)]];
     float3 P2        [[attribute(3)]];
     float2 vSegments [[attribute(4)]];
};

struct PerPatchData {
     float4 patchConstant          [[attribute(5)]];
     float4 someOtherPatchConstant [[attribute(6)]];
};

struct PatchData {
     patch_control_point<ControlPoint> cp; // Control-point data
     PerPatchData patchData; // Per-patch data
};

[[patch(quad)]]
vertex VertexOutput
post_tess_vertex_func(PatchData input [[stage_in ]}, …)
{…}
```

5.2.3 Attributes for Built-in Variables

Some graphics operations occur in the fixed-function pipeline stages and need to provide values to or receive values from graphics functions. *Built-in* input and output variables are used to communicate values between the graphics (vertex and fragment) functions and the fixed-function graphics pipeline stages. Attributes are used with arguments and the return type of graphics functions to identify these built-in variables.

5.2.3.1 Vertex Function Input Attributes

Table 5.2 lists the built-in attributes that can be specified for arguments to a vertex function and the corresponding data types with which they can be used.

Table 5.2. Attributes for vertex function input arguments

Attribute	Corresponding data types	Description
`amplification_count` macOS: Metal 2.3 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always.	`ushort` or `uint`	The number of output vertices produced for each vertex instance. The default value for `[[amplification_count]]` is 1, which indicates that vertex amplification is disabled.
`amplification_id` macOS: Metal 2.3 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always.	`ushort` or `uint`	The array index offset mappings for viewport and render target array indices, which enables routing an amplified vertex to a different viewport and render target. The value for `[[amplification_id]]` is in the range `[0, amplification_count)`.
`base_instance`	`ushort` or `uint`	The base instance value added to each instance identifier before reading per-instance data.
`base_vertex`	`ushort` or `uint`	The base vertex value added to each vertex identifier before reading per-vertex data.
`instance_id`	`ushort` or `uint`	The per-instance identifier, which includes the base instance value if one is specified. If the type for declaring `[[instance_id]]` is `uint`, the type for declaring `[[base_instance]]` needs to be `uint` or `ushort`. If the type for declaring `[[instance_id]]` is `ushort`, the type for declaring `[[base_instance]]` needs to be `ushort`.

Attribute	Corresponding data types	Description
`vertex_id`	`ushort` or `uint`	The per-vertex identifier, which includes the base vertex value if one is specified. If the type for declaring `[[vertex_id]]` is `uint`, the type for declaring `[[base_vertex]]` needs to be `uint` or `ushort`. If the type for declaring `[[vertex_id]]` is `ushort`, the type for declaring `[[base_vertex]]` needs to be `ushort`.

5.2.3.2 Post-Tessellation Vertex Function Input Attributes

Table 5.3 lists the built-in attributes that can be specified for arguments to a post-tessellation vertex function and the corresponding data types with which they can be used.

All OS: Metal 1.2 and later support all attributes in Table 5.3.

Table 5.3. Attributes for post-tessellation vertex function input arguments

Attribute	Corresponding data types	Description
`base_instance`	`ushort` or `uint`	The base instance value added to each instance identifier before reading per-instance data.
`instance_id`	`ushort` or `uint`	The per-instance identifier, which includes the base instance value if one is specified. If the type for declaring `[[instance_id]]` is `uint`, the type for declaring `[[base_instance]]` needs to be `uint` or `ushort`. If the type for declaring `[[instance_id]]` is `ushort`, the type for declaring `[[base_instance]]` needs to be `ushort`.

Attribute	Corresponding data types	Description
`patch_id`	`ushort` or `uint`	The patch identifier.
`position_in_patch`	`float2` or `float3`	Defines the location on the patch being evaluated. For quad patches, must be `float2`. For triangle patches, must be `float3`.

5.2.3.3 Vertex Function Output Attributes

Table 5.4 lists the built-in attributes that can be specified for a return type of a vertex function or the members of a structure that a vertex function returns (and their corresponding data types).

All OS: Metal 1 and later support all attributes in Table 5.4 unless otherwise indicated.

Table 5.4. Attributes for vertex function return type

Attribute	Corresponding data types	Description
`clip_distance`	`float` or `float[n]` `n` needs to be known at compile time	Distance from vertex to clipping plane.
`invariant` All OS: Metal 2.1 and later.	Not applicable; needs to be used with `[[position]]`	Marks the output position such that if the sequence of operations used to compute the output position in multiple vertex shaders is identical, there is a high likelihood that the resulting output position computed by these vertex shaders are the same value. Requires users to pass `-fpreserve-invariance`. See the description below for more information.
`point_size`	`float`	Size of a point primitive
`position`	`float4`	The transformed vertex position

Attribute	Corresponding data types	Description
`render_target_array_index` macOS: Metal 1.1 and later. iOS: Metal 2.1 and later. iPadOS and visionOS: Always.	`uchar`, `ushort`, or `uint`	The array index that refers to one of: 1) an array slice of a texture array, 2) data at a specified depth of a 3D texture, 3) the face of a cubemap, or 4) a specified face of a specified array slice of a cubemap array.
`shared` macOS: Metal 2.3 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Not applicable	If present, then for every `amplification_id`, the output has the same value.
`viewport_array_index` macOS: Metal 2 and later. iOS: Metal 2.1 and later. iPadOS and visionOS: Always.	`uchar`, `ushort`, or `uint`	The viewport (and scissor rectangle) index value of the primitive.

A cubemap is represented as a render target array with six layers, one for each face, and `[[render_target_array_index]]` is the face index, which is a value from `0` to `5`. For a cubemap array, the `[[render_target_array_index]]` is computed as: `array_slice_index * 6 + face_index`.

You must return the same value of `[[render_target_array_index]]` for every vertex in a primitive. If values differ, the behavior and value passed to the fragment function are undefined. The same behavior applies to primitives generated by tessellation. If `[[render_target_array_index]]` is out-of-bounds (that is, greater than or equal to `renderTargetArrayLength`), the hardware interprets this value as `0`. For more about `[[render_target_array_index]]` as fragment function input, see section 5.2.3.4.

`[[viewport_array_index]]` enables specifying one viewport and scissor rectangle from multiple active viewports and scissor rectangles. If the vertex function does not specify `[[viewport_array_index]]`, the output viewport array index value is `0`. For more about `[[viewport_array_index]]`, see section 5.10.

`[[invariant]]` indicates that the floating-point math used in multiple function passes must generate a vertex position that matches exactly for every pass. `[[invariant]]` may only be used for a position in a vertex function (fields with the `[[position]]` attribute) to indicate the result of the calculation for the output is invariant. Compilers prior to iOS 14 and macOS 11, the calculation is likely (although not guaranteed) to be invariant. This calculation is now guaranteed to be invariant when passing `-fpreserve-invariance` option or setting the `preserveInvariance` on the `MTLCompilerOptions` from the Metal API for runtime compilation. Note that `[[invariant]]` is ignored if the options are not passed. This position invariance is essential for techniques such as shadow volumes or a z-prepass.

If the return type of a vertex function is not `void`, it must include the vertex position. If the vertex return type is `float4`, then it always refers to the vertex position, and the

`[[position]]` attribute must not be specified. If the vertex return type is a structure, it must include an element declared with the `[[position]]` attribute.

The following example describes a vertex function called `process_vertex`. The function returns a user-defined structure called `VertexOutput`, which contains a built-in variable that represents the vertex position, so it requires the `[[position]]` attribute.

```
struct VertexOutput {
     float4 position [[position]];
     float4 color;
     float2 texcoord;
}

vertex VertexOutput
process_vertex(…)
{
     VertexOutput v_out;
     // Compute per-vertex output.
     …
     return v_out;
}
```

Post-tessellation vertex function outputs are the same as a regular vertex function.

If vertex amplification is enabled, and if a vertex output variable has the same value for every [[`amplification_id]]` attribute, the vertex output is considered *shared*. A vertex output that is shared may use a single varying output slot, which is a limited resource. Vertex outputs that are not shared consume more than one varying output slot. (The Metal framework call `[MTLRenderPipelineDescriptor maxVertexAmplificationCount]` returns the number of varying slots that may be used to pass the amplified data to fragment function invocations, which impacts the number of total available varying slots.)

By default, all built-in vertex outputs are shared, except for those with the `[[position]]` attribute. By default, all other vertex outputs are not shared. To explicitly specify that the output is shared, use the `[[shared]]` attribute with a vertex output variable.

If the shader compiler can deduce that a vertex output variable has the same value for every `amplification_id`, the compiler may mark that vertex output as shared. The compiler may not mark vertex outputs as shared in any of these cases:

- The output value depends on the `[[amplification_id]]`.
- An atomic read-modify-write operation returns the output value.
- The shader loads the output value from volatile memory.

5.2.3.4 Fragment Function Input Attributes

Table 5.5 lists the built-in attributes that can be specified for arguments of a fragment function (and their corresponding data types).

If the return type of a vertex function is not `void`, it must include the vertex position. If the vertex return type is `float4`, this always refers to the vertex position (and the `[[position]]`

attribute need not be specified). If the vertex return type is a structure, it must include an element declared with the `[[position]]` attribute.

Table 5.5. Attributes for fragment function input arguments

Attribute	Corresponding data types	Description
`amplification_count` macOS: Metal 2.3 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The number of output vertices produced for each vertex instance.
`amplification_id` macOS: Since Metal 2.3 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The array index offset mappings for viewport and render target array indices, which enables routing an amplified vertex to a different viewport and render target.
`barycentric_coord` macOS: Metal 2.2 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	`float`, `float2`, or `float3`	The barycentric coordinates.
`color(m)` macOS: Metal 2.3 and later. iOS: Metal 1 and later. iPadOS and visionOS: Always.	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` `m` needs to be known at compile time	The input value read from a color attachment. The index `m` indicates which color attachment to read from.
`front_facing`	`bool`	This value is `true` if the fragment belongs to a front-facing primitive.
`point_coord`	`float2`	Two-dimensional coordinates, which range from 0.0 to 1.0 across a point primitive, specifying the location of the current fragment within the point primitive.
`position`	`float4`	Describes the window-relative coordinate `(x, y, z, 1/w)` values for the fragment.
`primitive_id` macOS: Metal 2.2 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	`uint`	The per-primitive identifier used with barycentric coordinates.

Attribute	Corresponding data types	Description
`render_target_array_index` macOS: Metal 1.1 and later. iOS: Metal 2.1 and later. iPadOS and visionOS: Always.	`uchar`, `ushort`, or `uint`	The render target array index, which refers to the face of a cubemap, data at a specified depth of a 3D texture, an array slice of a texture array, an array slice, or face of a cubemap array. For a cubemap, the render target array index is the face index, which is a value from `0` to `5`. For a cubemap array the render target array index is computed as: `array slice index * 6 + face index`.
`sample_id`	`uint`	The sample number of the sample currently being processed.
`sample_mask`	`uint`	The set of samples covered by the primitive generating the fragment during multisample rasterization.
`sample_mask, post_depth_coverage` iOS: Metal 2 and later. macOS: Metal 2.3 and later. iPadOS and visionOS: Always.	`uint`	The set of samples covered by the primitive generating the fragment after application of the early depth and stencil tests during multisample rasterization. The `early_fragment_tests` attribute needs to be used on the fragment function; otherwise, the compilation fails.
`thread_index_in_quadgroup` All OS: Metal 2.2 and later.	`ushort` or `uint`	The scalar index of a thread within a quad-group.
`thread_index_in_simdgroup` All OS: Metal 2.2 and later.	`ushort` or `uint`	The scalar index of a thread within a SIMD-group.
`threads_per_simdgroup` All OS: Metal 2.2 and later.	`ushort` or `uint`	The thread execution width of a SIMD-group.
`viewport_array_index` macOS: Metal 2 and later. iOS: Metal 2.1 and later. iPadOS and visionOS: Always.	`uint`	The viewport (and scissor rectangle) index value of the primitive.

A variable declared with the `[[position]]` attribute as input to a fragment function can only be declared with the `center_no_perspective` sampling and interpolation attribute. (See section 5.4.)

For `[[color(m)]]`, `m` is used to specify the color attachment index when accessing (reading or writing) multiple color attachments in a fragment function.

The `[[sample_mask]]` attribute can only be declared once for a fragment function input.

The value of `[[render_target_array_index]]` in the fragment function is the same value written from the vertex function, even if the specified value is out of range.

For more about `[[viewport_array_index]]`, see section 5.10.

The default value for `[[amplification_count]]` is 1, which indicates that vertex amplification is disabled.

The value for `[[amplification_id]]` shall be in the range `[0, amplification_count)`.

For a specified `[[amplification_id]]` attribute value, the `[[viewport_array_index]]` and `[[render_target_array_index]]` built-in fragment input values are added to (offset by) the values that the corresponding `MTLVertexAmplificationViewMapping` structure provides.

The following example describes the structure `MyVertexOut` that is both a vertex function return type and a fragment function input type. `MyVertexOut` uses the `[[amplification_id]]` attribute for the input argument `amp_id` to amplify the `position` and `ampData` members. Use of the `[[shared]]` attribute explicitly ensures the `texcoord` member as having the same value for all varyings under vertex amplification, as described in section 5.2.3.3.

In the vertex function `myVertex`, the `[[amplification_id]]` and `[[amplification_count]]` attributes specify the vertex function input variables for vertex amplification, as detailed in section 5.2.3.1. The shader compiler deduces that the `normal` member has the same value for every `[[amplification_id]]`, so the compiler marks it as shared in vertex output.

In the fragment function `myFragment`, the same `[[amplification_id]]` and `[[amplification_count]]` attributes specify fragment function input variables. If vertex amplification is enabled, then `amp_id` determines the mapping (`MTLVertexAmplificationViewMapping` structure) from which to select the viewport array index (`viewportArrayIndexOffset` member).

```
struct MyVertexIn {
    float4 position [[attribute(0)]];
    float3 normal   [[attribute(1)]];
    float3 tangent  [[attribute(2)]];
    float2 texcoord [[attribute(3)]];
};

struct MyVertexOut {
    float4 position [[position]];
    float3 normal;
    float3 tangent;
```

```
    float3 bitangent;
    float2 texcoord [[shared]]; // Explicitly shared.
    float  ampData;
    ushort viewport [[viewport_array_index]]; // Implicitly shared.
};

constexpr ushort MAX_AMP = 2;

vertex MyVertexOut myVertex(MyVertexIn in [[stage_in]],
                            constant float4x4 view_proj[MAX_AMP],
                            constant float data[MAX_AMP],
                            ushort amp_id [[amplification_id]],
                            ushort amp_count
[[amplification_count]], ...)
{
    MyVertexOut vert;
    // Deduced amplified
    vert.position  = view_proj[amp_id] * in.position;
    vert.normal    = in.normal; // Deduced shared
    vert.tangent   = ...;
    vert.bitangent = ...;
    vert.texcoord  = ...;
    vert.ampData   = data[amp_id]; // Not shared.
    vert.viewport  = 1;
    return vert;
}

fragment float4 myFragment(MyVertexOut in [[ stage_in ]],
                           ushort amp_id [[amplification_id]],
                           ushort amp_count [[amplification_count]],
                           ...) {
    // For MTLVertexAmplificationViewMapping = {{1,3},{2,4}}
    //   when amp_id == 0, in.viewport == 2
    //   when amp_id == 1, in.viewport == 3
    ushort viewport = in.viewport;
    ...
}
```

A fragment function input declared with the `[[barycentric_coord]]` attribute can only be declared with either the `center_perspective` (default) or `center_no_perspective` sampling and interpolation attributes. The barycentric coordinates and per-pixel primitive ID can be passed as fragment function input in structures organized as shown in these examples:

```
struct FragmentInput0 {
    uint primitive_id [[primitive_id]];
    // [[center_perspective]] is the default, so it can be omitted.
    float3 barycentric_coord [[barycentric_coord,
center_perspective]];
};
```

```
struct FragmentInput1 {
    uint primitive_id [[primitive_id]];
    float2 linear_barycentric_coord [[barycentric_coord,
                                      center_no_perspective]];
};
```

By storing the barycentric coordinates and per-pixel primitive ID, your shader can manually read and interpolate the vertices of a drawn primitive within the fragment phase or defer this interpolation to a separate pass. In the deferred interpolation scenario, you can use a thin buffer during the geometry pass to store a minimal set of surface data, including pre-clipped barycentric coordinates. At a later stage, you must have enough data to reconstruct the original vertex indices from the primitive ID data and to correlate the barycentric coordinates to those vertex indices.

When applying the `barycentric_coord` attribute to an input argument (or to a field of an argument) with *more* components than the dimension of the primitive, the remaining elements are initialized with `0.0f`. For example, for

```
fragment float4
frag (float3 coord [[barycentric_coord]]) { ... }
```

- When drawing a point, `coord.yz` is `float2(0.0f)`.
- When drawing a line, `coord.z` is `0.0f`.

When applying the `barycentric_coord` attribute to an input argument (or to a field of an argument) with *fewer* components than the dimension of the primitive, the remaining elements are ignored.

Table 5.6 lists attributes that can be specified for tile arguments that are input to a fragment function. The data types for declaring `[[pixel_position_in_tile]]` and `[[pixels_per_tile]]` must match.

Table 5.6. Attributes for fragment function tile input arguments

Attribute	Corresponding data types	Description
`pixel_position_in_tile`	`ushort2` or `uint2`	(x, y) position of the fragment in the tile.
`pixels_per_tile`	`ushort2` or `uint2`	(width, height) of the tile in pixels.
`tile_index`	`ushort` or `uint`	1D tile index.
`render_target_array_index`	`uchar`, `ushort`, or `uint`	The render target array index, which refers to the face of a cubemap, data at a specified depth of a 3D texture, an array slice of a texture array, an array

Attribute	Corresponding data types	Description
		slice, or face of a cubemap array. For a cubemap, the render target array index is the face index, which is a value from `0` to `5`. For a cubemap array the render target array index is computed as: `array slice index * 6 + face index.`

macOS: Metal 2.3 and later support all attributes in Table 5.6.
iOS: Metal 2 and later support all attributes in Table 5.6.
iPadOS and visionOS: Metal support all attributes in Table 5.6.

`[[tile_index]]` is a value from `[0, n)`, where *n* is the number of tiles in the render target.

5.2.3.5 Fragment Function Output Attributes

The return type of a fragment function describes the per-fragment output. You must use the attributes listed in Table 5.7 to specify that a fragment function can output one or more render-target color values, a depth value, a sampling coverage mask, or a stencil reference value. If the depth value is not output by the fragment function, the depth value generated by the rasterizer is output to the depth attachment.

Table 5.7. Attributes for fragment function return types

Attribute	Corresponding data types	Description

`color(m)`
All OS: Metal 1 and later.
 	
`color(m), index(i)`
All OS: Metal 1.2 and later.
	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn`	Color value output for a color attachment.
		
		`m` is the color attachment index and needs to be known at compile time. The index `i` can be used to specify one or more colors output by a fragment function for a given color attachment and is an input to the blend equation.

`depth(depth_argument)` All OS: Metal 1 and later.	`float`	Depth value output using the function specified by `depth_argument`.
`sample_mask` All OS: Metal 1 and later.	`uint`	Coverage mask.
`stencil` All OS: Metal 2.1 and later.	`uint`	Stencil reference value to be used in a stencil test.

The color attachment index `m` for fragment output is specified in the same way as it is for `[[color(m)]]` for fragment input (see discussion for Table 5.5). Multiple elements in the fragment function return type that use the same color attachment index for blending needs to be declared with the same data type.

If there is only a single-color attachment in a fragment function, then `[[color(m)]]` is optional. If `[[color(m)]]` is not specified, the attachment index is `0`. If multiple color attachments are specified, `[[color(m)]]` needs to be specified for all color values. See examples of specifying the color attachment in sections 5.5 and 5.8.1.5.

If `index(i)` is not specified in the attribute, the default is an index of `0`. If `index(i)` is specified, the value of `i` needs to be known at compile time.

If a fragment function writes a custom depth value, specify the `depth_argument` parameter as `any`, `greater`, or `less`. The setting controls how the `depth(depth_argument)` attribute on a fragment output interacts with the default depth value that the compiler generates for you. Set `depth_argument` to:

- `any` — Accept any values.
- `greater` — Only accept values that are greater than the default depth.
- `less` — Only accept values that are less than the default depth.

Your app may exhibit unpredictable results if fragment output marked with `depth(depth_argument)` produces a value that conflicts with the `depth_argument` setting.

You cannot use the `[[stencil]]` attribute in fragment-based tile shading functions. The `[[stencil]]` attribute is not compatible with the `[[early_fragment_tests]]` function attribute.

If the fragment function does not output the stencil value, the `setStencilReferenceValue:` or `setStencilFrontReferenceValue:backReferenceValue:` method of `MTLRenderCommandEncoder` can set the stencil reference value.

The following example shows how color attachment indices can be specified. Color values written in `clr_f` write to color attachment index `0`, `clr_i` to color attachment index `1`, and `clr_ui` to color attachment index `2`.

```
struct MyFragmentOutput {
     // Color attachment 0
     float4 clr_f [[color(0)]];

     // Color attachment 1
     int4 clr_i [[color(1)]];

     // Color attachment 2
     uint4 clr_ui [[color(2)]];
}
```

```
fragment MyFragmentOutput
my_fragment(…)
{
     MyFragmentOutput f;
      …
      f.clr_f = …;
      …
      return f;
}
```

If a color attachment index is used as both an input to and an output of a fragment function, the data types associated with the input argument and output declared with this color attachment index must match.

5.2.3.6 Kernel Function Input Attributes

When a kernel function is submitted for execution, it executes over an N-dimensional grid of threads, where N is one, two, or three. A thread is an instance of the kernel function that executes for each point in this grid, and `thread_position_in_grid` identifies its position in the grid.

Within a compute unit, a threadgroup is partitioned into multiple smaller groups for execution. The execution width of the compute unit, referred to as the `threads_per_simdgroup`, determines the recommended size of this smaller group. For best performance, make the total number of threads in the threadgroup a multiple of the `threads_per_simdgroup`.

Threadgroups are assigned a unique position within the grid (referred to as `threadgroup_position_in_grid`). Threads are assigned a unique position within a threadgroup (referred to as `thread_position_in_threadgroup`). The unique scalar index of a thread within a threadgroup is given by `thread_index_in_threadgroup`.

Each thread's position in the grid and position in the threadgroup are N-dimensional tuples. Threadgroups are assigned a position using a similar approach to that used for threads. Threads are assigned to a threadgroup and given a position in the threadgroup with components in the range from zero to the size of the threadgroup size in that dimension minus one.

When a kernel function is submitted for execution, the number of threadgroups and the threadgroup size are specified, or the number of threads in the grid and the threadgroup size are specified. For example, consider a kernel function submitted for execution that uses a 2D grid where the number of threadgroups specified are `(Wx, Wy)` and the threadgroup size is `(Sx, Sy)`. Let `(wx, wy)` be the position of each threadgroup in the grid (`threadgroup_position_in_grid`) and `(lx, ly)` be the position of each thread in the threadgroup (`thread_position_in_threadgroup`).

The thread position in the grid (`thread_position_in_grid`) is:

```
(gx, gy) = (wx * Sx + lx, wy * Sy + ly)
```

The grid size (`threads_per_grid`) is:

```
(Gx, Gy) = (Wx * Sx, Wy * Sy)
```

In cases other than a tile function, the thread index in the threadgroup (`thread_index_in_threadgroup`) is determined by:

```
ly * Sx + lx
```

For a tile function, the thread index is not a linear mapping from the `lx` and `ly` values. Each thread in a tile function is guaranteed to get a unique index in the range `[0, Sx * Sy)`.

Within a threadgroup, threads are divided into SIMD-groups in an implementation-defined fashion. Any given thread in a SIMD-group can query its SIMD lane ID and which SIMD-group it is a member of.

Table 5.8 lists the built-in attributes that can be specified for arguments to a kernel function and the corresponding data types with which they can be used. In Metal 3.1 and later, provide the built-in attributes can be specified on global (program scope) variables to be used in a kernel context.

Table 5.8. Attributes for kernel function input arguments

Attribute	Corresponding data types	Description
`dispatch_quadgroups_per_threadgroup` macOS: Metal 2.1 and later. iOS: Metal 2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The quad-group execution width of a threadgroup specified at dispatch.
`dispatch_simdgroups_per_threadgroup` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The SIMD-group execution width of a threadgroup specified at dispatch.
`dispatch_threads_per_threadgroup` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup for threads specified at dispatch.
`grid_origin` All OS: Metal 1.2 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The origin (offset) of the grid over which compute threads that read per-thread stage-in data are launched.

Attribute	Corresponding data types	Description
`grid_size` All OS: Metal 1.2 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The maximum size of the grid over which compute threads that read per-thread stage-in data are launched.
`quadgroup_index_in_threadgroup` macOS: Metal 2.1 and later iOS: Metal 2 and later iPadOS and visionOS: Always.	`ushort` or `uint`	The scalar index of a quad-group within a threadgroup.
`quadgroups_per_threadgroup` macOS: Metal 2.1 and later. iOS: Metal 2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The quad-group execution width of a threadgroup.
`simdgroup_index_in_threadgroup` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The scalar index of a SIMD-group within a threadgroup.
`simdgroups_per_threadgroup` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The SIMD-group execution width of a threadgroup.
`thread_execution_width` All OS: Metal 1 and later. [[Deprecated as of Metal 3 – use `threads_per_simdgroup.`]]	`ushort` or `uint`	The thread execution width of a SIMD-group (compute unit).
`thread_index_in_quadgroup` macOS: Metal 2.1 and later. iOS: Metal 2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The scalar index of a thread within a quad-group.
`thread_index_in_simdgroup` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The scalar index of a thread within a SIMD-group.
`thread_index_in_threadgroup` All OS: Metal 1 and later.	`ushort` or `uint`	The scalar index of a thread within a threadgroup.

Attribute	Corresponding data types	Description
`thread_position_in_grid` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's position in an N-dimensional grid of threads.
`thread_position_in_threadgroup` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's unique position within a threadgroup
`threadgroup_position_in_grid` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The threadgroup's unique position within a grid.
`threadgroups_per_grid` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The number of threadgroups in a grid.
`threads_per_grid` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The grid size.
`threads_per_simdgroup` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	`ushort` or `uint`	The thread execution width of a SIMD-group (compute unit).
`threads_per_threadgroup` All OS: Metal 1 and later.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup.

All OS: Metal 1.2 and later support `grid_origin` and `grid_size`.
macOS: Metal 2 and later support SIMD-group attributes. Metal 2.1 and later support quad-group attributes. Metal 1 and later support other attributes.
iOS: Metal 2 and later support SIMD-group and quad-group attributes. Metal 1 and later support all other attributes.
iPadOS and visionOS: Metal supports SIMD-group, quad-group, and all other attributes.

All OS: Metal 3.1 and later support global (program scope) variables. You can specify these attributes except when using them in a dynamic library or a separately compiled binary function. In Metal 3.2 and later, you can also use global variables in a dynamic library or a separately compiled binary function for Apple silicon.

For standard Metal compute functions (other than tile functions), SIMD-groups are linear and one-dimensional. (Threadgroups may be multidimensional.) The number of SIMD-groups in a threadgroup (`[[simdgroups_per_threadgroup]]`) is the total number threads in the threadgroup (`[[threads_per_threadgroup]]`) divided by the SIMD-group size (`[[threads_per_simdgroup]]`):

```
simdgroups_per_threadgroup = ceil(threads_per_threadgroup/
threads_per_simdgroup)
```

Similarly, the number of quad-groups in a threadgroup (`quadgroups_per_threadgroup`) is the total number of threads in threadgroup divided by 4, which is the thread execution width of a quad-group:

```
quadgroups_per_threadgroup = ceil(threads_per_threadgroup/4)
```

For tile functions, threads are arranged as 2 x 2 quads. For a 2D grid where the number of threadgroups specified are `(Wx, Wy)`, `simdgroups_per_threadgroup` is computed by:

```
simdgroups_per_threadgroup = ceil(Wx/2) * 2 * ceil(Wy/2) * 2 /
threads_per_simdgroup
```

```
simdgroups_per_threadgroup =
ceil(Wx/2)*ceil(Wy/2)*4/threads_per_simdgroup
```

For tile functions, `quadgroups_per_threadgroup` is computed by:

```
quadgroups_per_threadgroup = ceil(Wx/2) * 2 * ceil(Wy/2) * 2 / 4
```

```
quadgroups_per_threadgroup = ceil(Wx/2) * ceil(Wy/2)
```

`[[dispatch_simdgroups_per_threadgroup]]` and `[[dispatch_quadgroups_per_threadgroup]]` are similarly computed for threads specified at dispatch.

SIMD-groups execute concurrently within a given threadgroup and make independent forward progress with respect to each other, in the absence of threadgroup barrier operations. The thread index in a SIMD-group (given by `[[thread_index_in_simdgroup]]`) is a value between `0` and SIMD-group size `−1`, inclusive. Similarly, the thread index in a quad-group (given by `[[thread_index_in_quadgroup]]`) is a value between `0` and `3`, inclusive.

In Metal 2 and later, the number of threads in the grid does not have to be a multiple of the number of threads in a threadgroup. It is therefore possible that the actual threadgroup size of a specific threadgroup may be smaller than the threadgroup size specified in the dispatch. The `[[threads_per_threadgroup]]` attribute specifies the actual threadgroup size for a given threadgroup executing the kernel. The `[[dispatch_threads_per_threadgroup]]` attribute is the threadgroup size specified at dispatch.

Notes on kernel function attributes:

- The type for declaring `[[thread_position_in_grid]]`, `[[threads_per_grid]]`, `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`,

`[[threadgroup_position_in_grid]]`, `[[dispatch_threads_per_threadgroup]]`, and `[[threadgroups_per_grid]]` needs to be a scalar type or a vector type. If it is a vector type, the number of components for the vector types for declaring these arguments need to match.

- The data types for declaring `[[thread_position_in_grid]]` and `[[threads_per_grid]]` need to match.
- The data types for declaring `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, and `[[dispatch_threads_per_threadgroup]]` need to match.
- If `[[thread_position_in_threadgroup]]` is type `uint`, `uint2`, or `uint3`, `[[thread_index_in_threadgroup]]` needs to be type `uint`.
- The types for declaring `[[thread_index_in_simdgroup]]`, `[[threads_per_simdgroup]]`, `[[simdgroup_index_in_threadgroup]]`, `[[simdgroups_per_threadgroup]]`, `[[dispatch_simdgroups_per_threadgroup]]`, `[[quadgroup_index_in_threadgroup]]`, `[[quadgroups_per_threadgroup]]`, and `[[dispatch_quadgroups_per_threadgroup]]` need to be `ushort` or `uint`. The types for declaring these built-in variables need to match.
- `[[threads_per_simdgroup]]` and `[[thread_execution_width]]` are aliases of one another that reference the same concept.

Table 5.9. Attributes for kernel function tile input arguments

Attribute	Corresponding data types	Description
`render_target_array_index`	`uchar`, `ushort`, or `uint`	The render target array index, which refers to the face of a cubemap, data at a specified depth of a 3D texture, an array slice of a texture array, an array slice, or face of a cubemap array. For a cubemap, the render target array index is the face index, which is a value from `0` to `5`. For a cubemap array the render target array index is computed as: `array slice index * 6 + face index`.

macOS: Metal 2.3 and later support all attributes in Table 5.9.
iOS: Metal 2 and later support all attributes in Table 5.9.
iPadOS and visionOS: Metal supports all attributes in Table 5.9.

5.2.3.7 Intersection Function Input Attributes

Table 5.10 lists the built-in attributes that can be specified for arguments to a custom intersection function (see section 5.1.6). Some built-in attributes can be used when specific values of `primitive_type` and `intersection_tags` are specified on the intersection function.

For example, `instance_id` is available if `intersection_tags` contains `instancing`:

```
[[intersection(triangle, triangle_data, instancing,
world_space_data)]]
bool triangleIntersectionFunction(..., uint id [[instance_id]], …)
{…}
```

Any such restriction is listed in the description of the attribute.

Table 5.10. Attributes for intersection function input arguments

Attribute	Corresponding data types	Description
`origin`	`float3`	Ray origin in object space.
`direction`	`float3`	Ray direction in object space.
`min_distance`	`float`	Ray min distance.
`max_distance`	`float`	Passed by reference. Returns the current closest intersection max distance. The intersector initializes the initial value with the ray's maximum distance and the value decreases as the intersector finds intersections.
`payload`	`User type.` Passed by reference.	User defined payload passed by the calling thread. Needs to be specified to allow matching payload table by `intersect()` (section 6.19.2).
`geometry_id`	`ushort` or `uint`	The per-geometry id.

Attribute	Corresponding data types	Description
`primitive_id`	`ushort` or `uint`	The per-primitive identifier. For curves, this is a curve segment index.
`instance_id`	`ushort`, `uint` or `array_ref<uint>`	The per-instance identifier. Available if `intersection_tags` include `instancing`. In Metal 3.1 and later, if `intersection_tags` include `max_levels<Count>`, the type must be `array_ref<uint>`. Otherwise, it is `ushort` or `uint`.
`world_space_origin`	`float3`	Origin in world space. Available if `intersection_tags` include `world_space_data`.
`world_space_direction`	`float3`	Direction in world space. Available if `intersection_tags` include `world_space_data`.
`barycentric_coord`	`float2`	The barycentric coordinates. Available if the `primitive_type` is `triangle` and intersection tag include `triangle_data`.
`front_facing`	`bool`	This value is true if the triangle front face is visible from the ray origin. Available if `intersection_tags` include `triangle_data`.
`distance`	`float`	Distance along the ray at the triangle intersection. Available if the `primitive_type` is `triangle`.
`opaque`	`bool`	If this primitive should be considered opaque or not. Available if the `primitive_type` is a `bounding_box`.
`instance_intersection_function_table_offset`	`ushort` or `uint`	Offset into the intersection function table used to select the intersection instance.

Attribute	Corresponding data types	Description
`geometry_intersection_function_table_offset`	`ushort` or `uint`	Offset into the geometry object used to select to select the intersection instance.
`time` All OS: Metal 2.4 and later	`float`	Ray intersection time. Available if `intersection_tags` include `primitive_motion`.
`motion_start_time` All OS: Metal 2.4 and later	`float`	Motion start time for this geometry. Available if `intersection_tags` include `primitive_motion`.
`motion_end_time` All OS: Metal 2.4 and later	`float`	Motion end time for this geometry. Available if `intersection_tags` include `primitive_motion`.
`key_frame_count` All OS: Metal 2.4 and later	`ushort` or `uint`	Number of key frames. Available if `intersection_tags` include `primitive_motion`.
`object_to_world_transform` All OS: Metal 2.4 and later	`float4x3`	Object space to world space transformation matrix. Available if `intersection_tags` include `instancing` and `world_space_data`. If `intersection_tags` include `instance_motion`, the matrix is interpolated based on the `time`.
`world_to_object_transform` All OS: Metal 2.4 and later	`float4x3`	World space to object space transformation matrix. Available if `intersection_tags` include `instancing` and `world_space_data`. If `intersection_tags` include `instance_motion`, the matrix is interpolated based on the `time`.
`user_instance_id` All OS: Metal 2.4 and later	`ushort`, `uint` or `array_ref<uint>`	User defined instance id. Available if `intersection_tags` include

Attribute	Corresponding data types	Description
		`instancing`. In Metal 3.1 and later, if `intersection_tags` include `max_levels<Count>`, the type must be `array_ref<uint>`. Otherwise, it is `ushort` or `uint`.

`primitive_data`
All OS: Metal 3 and later
	`const device T*`
	or
	`const device T&`
		Per-primitive data. The data is read-only and passed in the device address space.

`curve_parameter` All OS: Metal 3.1 and later	`float`	The value which you need to pass to the curve basis functions to reconstruct the position corresponding to the intersection along the curve segment. This will be exactly `0.0F` or `1.0F` if, and only if, the ray intersects a curve end cap or elbow. Available if `intersection_tags` include `curve_data`. See section 6.19.7 for a set of curve utility functions.
`function_id` All OS: Metal 4 and later	`ushort` or `uint`	Specifies the index you use to determine the intersection function being invoked by the GPU. Available if `intersection_tags` include `intersection_function_buffer`.

`user_data_buffer`
All OS: Metal 4 and later
	`const device T*`
	or
	`const device T&`
		User data passed. Available if `intersection_tags` include `intersection_function_buffer` and `user_data`.

For vertex attributes `v0`, `v1`, and `v2`, the attribute value at the specified barycentric point is:

```
v1 * barycentric_coord.x +
v2 * barycentric_coord.y +
v0 * (1.0f - (barycentric_coord.x + barycentric_coord.y))
```

The type for a parameter with the `[[payload]]` attribute is of the form `ray_data T &`. It is passed by reference to the intersection functions, and it is allocated in the `ray_data` address space. The type `T` of the payload can be or contain the following types:

- `device` or `constant` pointers or references
- integer types
- enumeration types
- floating-point types
- vector types
- arrays of such types
- structure and union (except for `atomic<T'>` and `imageblock<T'>`).

5.2.3.8 Intersection Function Output Attributes

Table 5.11 lists the built-in attributes that can be specified for a return type of a `[[intersection(primitive_type, intersection_tags…)]]` function (and their corresponding data types).

Table 5.11. Attributes for intersection return types

Attribute	Corresponding data types	Description
`accept_intersection`	`bool`	If `true`, this primitive becomes the next committed hit: if it is the nearest, it will be returned from `intersect()`.
`continue_search`	`bool`	If the hit is accepted (`[[accept_intersection]]` == `true`), `continue_search` indicates if the search should continue. If `continue_search` is `true`, `intersect()` will continue to search for a closer hit. If `false`, no further searching is done. The current nearest hit is returned from `intersect()`. Defaults to `true`. Even if `true` is returned, a committed hit will immediately halt searching if `accept_any_intersection()` is `true`.
`distance`	`float`	This returns the distance along the ray of a hit found within the bounding box. If the hit is rejected (`[[accept_intersection]] == false`), this return value is ignored. Available if the `primitive_type` is a `bounding_box`.

For triangle intersection functions, `[[accept_intersection]]` is the only required return value. If the function returns a `bool` without an attribute, then it is assumed to be `[[accept_intersection]]`.

The value of `[[distance]]` needs to be greater than or equal to the value of `[[min_distance]]` and it needs to be less than or equal to the value of `[[max_distance]]` and within the custom primitive's bounding box (inclusive), or the results are undefined. If the value of `[[distance]]` is the same as the value of `[[max_distance]],` then accepting this hit takes precedence over the previous hit at the same distance.

Any changes made to the ray payload take effect regardless of how the intersection function returns: Rejected primitives can have side effects to memory that are observed by future intersection shader threads.

Writes to device memory also occur even for rejected primitives. Those writes are visible to other threads via the usual memory consistency and coherency rules (at present, only atomics will be coherent, and only relaxed consistency is supported). Intersection functions may be invoked even if the ray does not intersect the primitive's bounding box. For example, implementations may group multiple primitives into one acceleration structure leaf node.

Below is an example of an intersection function of a bounding box:

```
struct IntersectionResult {
    bool continueSearch [[continue_search]];
    bool accept [[accept_intersection]];
    float distance [[distance]];
};
[[intersection(bounding_box)]]
IntersectionResult sphereIntersectionFunction(
                    float3 origin [[origin]],
                    float3 direction [[direction]],
                    uint primitiveIndex [[primitive_id]],
                    ray_data float2& resources [[payload]],
                    float min_distance [[min_distance]],
                    float max_distance [[max_distance]])
{…}
```

5.2.3.9 Object Function Input Attributes

All OS: In Metal 3.1 and later, you can specify these attributes on global variables except when using them in a dynamic library or a separately compiled binary function.

Object functions use the same execution model as a kernel function (see section 5.2.3.6), where it executes over an N-dimensional grid of threads. Object functions arguments can be `samplers`, `textures`, arguments of type `mesh_grid_properties`, and buffers in the `device`, `constant`, and `threadgroup` address space.

Object functions support a subset of the built-in attributes of a kernel function and `[[amplification_count]]` and `[[payload]]`. The semantics of `[[amplification_count]]` is the same as in section 5.2.3.1 Vertex Function Input

Attributes. Table 5.12 lists the built-in attributes that can be specified for arguments to an object function and the corresponding data types with which they can be used. Metal 3.1 and later provide the built-in attributes in Table 5.12, which you can specify on program scope variables, except for `amplification_count` and `payload`.

Table 5.12. Attributes for object function

Attribute	Corresponding data types	Description
`amplification_count`	`ushort` or `uint`	The number of output vertices produced for each vertex instance.
`dispatch_quadgroups_per_threadgroup`	`ushort` or `uint`	The quad-group execution width of a threadgroup specified at dispatch.
`dispatch_simdgroups_per_threadgroup`	`ushort` or `uint`	The SIMD-group execution width of a threadgroup specified at dispatch.
`dispatch_threads_per_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup for threads specified at dispatch.
`payload`	Pointer or l-value reference to user-defined `T` in `object_data` address space	The payload is data passed to the mesh shader from the object shader. The payload pointer or reference is the same for all threads in the threadgroup. The payload memory is assumed uninitialized at the entry of the object function.
`quadgroup_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a quad-group within a threadgroup.
`quadgroups_per_threadgroup`	`ushort` or `uint`	The quad-group execution width of a threadgroup.

Attribute	Corresponding data types	Description
`simdgroup_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a SIMD-group within a threadgroup.
`simdgroups_per_threadgroup`	`ushort` or `uint`	The SIMD-group execution width of a threadgroup.
`thread_index_in_quadgroup`	`ushort` or `uint`	The scalar index of a thread within a quad-group.
`thread_index_in_simdgroup`	`ushort` or `uint`	The scalar index of a thread within a SIMD-group.
`thread_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a thread within a threadgroup.
`thread_position_in_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's position in an N-dimensional grid of threads.
`thread_position_in_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's unique position within a threadgroup
`threadgroup_position_in_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The threadgroup's unique position within a grid.
`threadgroups_per_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The number of threadgroups in a grid.
`threads_per_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The grid size.

Attribute	Corresponding data types	Description
`threads_per_simdgroup`	`ushort` or `uint`	The thread execution width of a SIMD-group.
`threads_per_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup.

Object function attributes have the same restrictions as kernel function attributes:

- The type for declaring `[[thread_position_in_grid]]`, `[[threads_per_grid]]`, `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, `[[threadgroup_position_in_grid]]`, `[[dispatch_threads_per_threadgroup]]`, and `[[threadgroups_per_grid]]` needs to be a scalar type or a vector type. If it's a vector type, the number of components for the vector types for declaring these arguments need to match.
- The data types for declaring `[[thread_position_in_grid]]` and `[[threads_per_grid]]` need to match.
- The data types for declaring `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, and `[[dispatch_threads_per_threadgroup]]` need to match.
- If `[[thread_position_in_threadgroup]]` is type `uint`, `uint2` or `uint3`, `[[thread_index_in_threadgroup]]` needs to be type `uint`.
- The types for declaring `[[thread_index_in_simdgroup]]`, `[[threads_per_simdgroup]]`, `[[simdgroup_index_in_threadgroup]]`, `[[simdgroups_per_threadgroup]]`, `[[dispatch_simdgroups_per_threadgroup]]`, `[[quadgroup_index_in_threadgroup]]`, `[[quadgroups_per_threadgroup]]`, and `[[dispatch_quadgroups_per_threadgroup]]`need to be `ushort` or `uint`. The types for declaring these built-in variables need to match.

5.2.3.10 Mesh Function Input Attributes

All OS: In Metal 3.1 and later, you can specify these attributes on global variables except when using them in a dynamic library or a separately compiled binary function.

Mesh functions use the same execution model as a kernel function (see section 5.2.3.6), where it executes over an N-dimensional grid of threads. Mesh functions arguments can be from `samplers`, `textures`, arguments of type `mesh<V, P, NV, NP, t>`, and buffers of `device` and `constant`. If the mesh function has a `mesh<V, P, NV, NP,t>` argument, it points to an opaque handle for memory representing the mesh to export. The underlying memory referenced by the `mesh<V, P, NV, NP, t>` argument is shared among threads of a given threadgroup.

Mesh functions support a subset of the built-in attributes of a kernel function and also `[[amplification_count]]`, `[[amplification_id]]`, and `[[payload]]` attributes. The semantics of `[[amplification_count]]` and `[[amplification_id]]` is the same as in section 5.2.3.1 Vertex Function Input Attributes. Table 5.13 lists the built-in attributes that can be specified for arguments to a mesh function and the corresponding data types with which they can be used. Metal 3.1 and later provide the built-in attributes in Table 5.13, which you can specify on program scope variables, except for `amplification_count`, `amplification_id`, and `payload`.

Table 5.13. Attributes for mesh function

Attribute	Corresponding data types	Description
`amplification_count`	`ushort` or `uint`	The number of output vertices produced for each primitive instance.
`amplification_id`	`ushort` or `uint`	The array index offset mappings for viewport and render target array indices, which enables routing an amplified vertex to a different viewport and render target.
`dispatch_quadgroups_per_threadgroup`	`ushort` or `uint`	The quad-group execution width of a threadgroup specified at dispatch.
`dispatch_simdgroups_per_threadgroup`	`ushort` or `uint`	The SIMD-group execution width of a threadgroup specified at dispatch.
`dispatch_threads_per_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup for threads specified at dispatch.
`payload`	Pointer or l-value reference to user-defined `T` in `object_data` address space. Needs to be const qualified.	The payload is data passed to the mesh shader from the object shader. The payload pointer or reference is the same for all threads in the `mesh grid`. The payload memory is read-only in the mesh function.

Attribute	Corresponding data types	Description
`quadgroup_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a quad-group within a threadgroup.
`quadgroups_per_threadgroup`	`ushort` or `uint`	The quad-group execution width of a threadgroup.
`simdgroup_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a SIMD-group within a threadgroup.
`simdgroups_per_threadgroup`	`ushort` or `uint`	The SIMD-group execution width of a threadgroup.
`thread_index_in_quadgroup`	`ushort` or `uint`	The scalar index of a thread within a quad-group.
`thread_index_in_simdgroup`	`ushort` or `uint`	The scalar index of a thread within a SIMD-group.
`thread_index_in_threadgroup`	`ushort` or `uint`	The scalar index of a thread within a threadgroup.
`thread_position_in_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's position in an N-dimensional grid of threads.
`thread_position_in_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's unique position within a threadgroup
`threadgroup_position_in_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The threadgroup's unique position within a grid.

Attribute	Corresponding data types	Description
`threadgroups_per_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The number of threadgroups in a grid.
`threads_per_grid`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The grid size.
`threads_per_simdgroup`	`ushort` or `uint`	The thread execution width of a SIMD-group.
`threads_per_threadgroup`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup.

Mesh function attributes have the same restrictions as kernel function attributes:

- The type for declaring `[[thread_position_in_grid]]`, `[[threads_per_grid]]`, `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, `[[threadgroup_position_in_grid]]`, `[[dispatch_threads_per_threadgroup]]`, and `[[threadgroups_per_grid]]` needs to be a scalar type or a vector type. If it's a vector type, the number of components for the vector types for declaring these arguments need to match.
- The data types for declaring `[[thread_position_in_grid]]` and `[[threads_per_grid]]` need to match.
- The data types for declaring `[[thread_position_in_threadgroup]]`, `[[threads_per_threadgroup]]`, and `[[dispatch_threads_per_threadgroup]]` need to match.
- If `[[thread_position_in_threadgroup]]` is type `uint`, `uint2` or `uint3`, `[[thread_index_in_threadgroup]]` needs to be type `uint`.
- The types for declaring `[[thread_index_in_simdgroup]]`, `[[threads_per_simdgroup]]`, `[[simdgroup_index_in_threadgroup]]`, `[[simdgroups_per_threadgroup]]`, `[[dispatch_simdgroups_per_threadgroup]]`, `[[quadgroup_index_in_threadgroup]]`, `[[quadgroups_per_threadgroup]]`, and `[[dispatch_quadgroups_per_threadgroup]]` need to be `ushort` or `uint`. The types for declaring these built-in variables need to match.

5.2.4 Input Assembly Attribute

Vertex function output and the rasterizer-generated fragments become the per-fragment inputs to a fragment function. The `[[stage_in]]` attribute can assemble the per-fragment inputs.

A vertex function can read per-vertex inputs by indexing into buffer(s) passed as arguments to the vertex function using the vertex and instance IDs. To assemble per-vertex inputs and pass them as arguments to a vertex function, declare the inputs with the `[[stage_in]]` attribute.

A kernel function reads per-thread inputs by indexing into buffer(s) or texture(s) passed as arguments to the kernel function using the thread position in grid or thread position in threadgroup IDs. In addition, to pass per-thread inputs as arguments to a kernel function, declaring the inputs with the `[[stage_in]]` attribute.

You can declare only one argument of the vertex, fragment, or kernel function with the `[[stage_in]]` attribute. For a user-defined structure declared with the `[[stage_in]]` attribute, the members of the structure can be:

- A scalar integer or floating-point value.
- A vector of integer or floating-point values.
- An `interpolant<T,P>` value for fragment function input.

You cannot use the `stage_in` attribute to declare members of the structure that are packed vectors, matrices, structures, bitfields, references or pointers to a type, or arrays of scalars, vectors, or matrices.

5.2.4.1 Vertex Function Output Example

The following example shows how to pass per-vertex inputs using the `stage_in` attribute:

```
struct VertexOutput {
    float4 position [[position]];
    float4 color;
    float2 texcoord;
};

struct VertexInput {
    float4 position [[attribute(0)]];
    float3 normal   [[attribute(1)]];
    half4 color     [[attribute(2)]];
    half2 texcoord  [[attribute(3)]];
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
    uint   num_lights;
    float4 light_position[MAX_LIGHTS];
    float4 light_color[MAX_LIGHTS];
    float4 light_attenuation_factors[MAX_LIGHTS];
```

```
};

constexpr sampler s = sampler(coord::normalized,
address::clamp_to_zero,
                              filter::linear);

vertex VertexOutput
render_vertex(VertexInput v_in [[stage_in]],
              constant float4x4& mvp_matrix [[buffer(1)]],
              constant LightDesc& lights [[buffer(2)]],
              uint v_id [[vertex_id]])
{
     VertexOutput v_out;
     v_out.position = v_in.position * mvp_matrix;
     v_out.color = do_lighting(v_in.position, v_in.normal, lights);
     …
     return v_out;
}
```

5.2.4.2 Fragment Function Input Example

An example in section 5.2.3.3 previously introduces the `process_vertex` vertex function, which returns a `VertexOutput` structure per vertex. In the following example, the output from `process_vertex` is pipelined to become input for a fragment function called `render_pixel`, so the first argument of the fragment function uses the `[[stage_in]]` attribute and uses the incoming `VertexOutput` type. (In `render_pixel`, the `imgA` and `imgB` 2D textures call the built-in function `sample`, which is introduced in section 6.13.3).

```
struct VertexOutput2 {
     float4 position [[position]];
     float4 color;
     float2 texcoord;
};

struct VertexInputData {
     float4 position;
     float3 normal;
     float2 texcoord;
};

constexpr constant uint MAX_LIGHTS = 4;

struct LightDesc {
     uint   num_lights;
     float4 light_position[MAX_LIGHTS];
     float4 light_color[MAX_LIGHTS];
     float4 light_attenuation_factors[MAX_LIGHTS];
};
```

```
constexpr sampler s = sampler(coord::normalized,
address::clamp_to_edge,
                              filter::linear);

vertex VertexOutput2
render_vertex(const device VertexInputData *v_in [[buffer(0)]],
              constant float4x4& mvp_matrix [[buffer(1)]],
              constant LightDesc& lights [[buffer(2)]],
              uint v_id [[vertex_id]])
{
     VertexOutput v_out;
     v_out.position = v_in[v_id].position * mvp_matrix;
     v_out.color = do_lighting(v_in[v_id].position,
     v_in[v_id].normal, lights);
     v_out.texcoord = v_in[v_id].texcoord;
     return v_out;
}

fragment float4
render_pixel(VertexOutput2 input [[stage_in]],
             texture2d<float> imgA [[texture(0)]],
             texture2d<float> imgB [[texture(1)]])
{
     float4 tex_clr0 = imgA.sample(s, input.texcoord);
     float4 tex_clr1 = imgB.sample(s, input.texcoord);

     // Compute color.
     float4 clr = compute_color(tex_clr0, tex_clr1, …);
     return clr;
}
```

5.2.4.3 Kernel Function Per-Thread Input Example

The following example shows how to use the `stage_in` attribute to pass per-thread inputs. The `stage_in` attribute in a kernel function allows you to decouple the data type for declaring the per-thread inputs in the function from the actual data type used to store the per-thread inputs.

```
struct PerThreadInput {
     float4 a [[attribute(0)]];
     float3 b [[attribute(1)]];
     half4 c  [[attribute(2)]];
     half2 d  [[attribute(3)]];
};

kernel void
my_kernel(PerThreadInput thread_input [[stage_in]],
          …
          uint t_id [[thread_position_in_grid]])
```

```
{…}
```

5.3 Storage Class Specifiers

Metal supports the `static` and `extern` storage class specifiers. Metal does not support the `thread_local` storage class specifiers.

You can only use the `extern` storage-class specifier for functions and variables declared in program scope or for variables declared inside a function. The `static` storage-class specifier is only for device variables declared in program scope (see section 4.2) and is not for variables declared inside a graphics or kernel function. The following example incorrectly uses the `static` specifier for the variables `b` and `c` declared inside a kernel function:

```
extern constant float4 noise_table[256];
static constant float4 color_table[256] = {…}; //Here, static is OK.

extern void my_foo(texture2d<float> img);
extern void my_bar(device float *a);

[[kernel]] void
my_kernel(texture2d<float> img [[texture(0)]],
          device float *ptr [[buffer(0)]])
{
     extern constant float4 a;
     static constant float4 b; // Here, static is an error.
     static float c; // Here, static is an error.

     …
     my_foo(img);
     …
     my_bar(ptr);
     …
}
```

5.4 Sampling and Interpolation Attributes

Sampling and interpolation attributes are used with inputs to fragment functions declared with the `stage_in` attribute except for members of type `interpolant<T,P>`. The attribute determines what sampling method the fragment function uses and how the interpolation is performed, including whether to use perspective-correct interpolation, linear interpolation, or no interpolation.

The sampling and interpolation attribute can be specified on any `stage_in` structure member whose type is scalar and vector. The sampling and interpolation attributes supported are:

- `center_perspective`
- `center_no_perspective`
- `centroid_perspective`

- `centroid_no_perspective`
- `sample_perspective`
- `sample_no_perspective`
- `flat`

`center_perspective` is the default sampling and interpolation attribute, with the following exceptions:

- For a variable with the `[[position]]` attribute, the only valid sampling and interpolation attribute is `center_no_perspective`.
- For an integer variable, the only valid sampling and interpolation attribute is `flat`.

A perspective attribute (`center_perspective`, `centroid_perspective`, or `sample_perspective`) indicates the values across a primitive are interpolated in a perspective-correct manner. A nonperspective attribute (`center_no_perspective`, `centroid_no_perspective`, or `sample_no_perspective`) indicates the values across a primitive are linearly interpolated in screen coordinates.

The center attribute variants (`center_perspective` and `center_no_perspective`) cause sampling to use the center of each pixel.

The sampling attribute variants (`sample_perspective` and `sample_no_perspective`) cause interpolation at a sample location rather than at the pixel center. With one of these attributes, the fragment function (or code blocks in the fragment function) that use these variables execute per-sample rather than per-fragment.

If a centroid attribute variant is specified (`centroid_perspective` and `centroid_no_perspective`), the interpolation point sampled needs to be within both the primitive and the centroid of the pixel.

The following example demonstrates how to specify the interpolation of data for different members of a user-defined structure:

```
struct FragmentInput {
     float4 pos [[center_no_perspective]];
     float4 color [[center_perspective]];
     float2 texcoord;
     int index [[flat]];
     float f [[sample_perspective]];
     interpolant<float4, interpolation::perspective> icolor;
};
```

In Metal 2.4 and later, the sample and interpolation attribute can also be specified on any `stage_in` structure member whose type is structure. All the members in the structure inherit the specified sampling and interpolation qualifiers. Field declarations in a structure where sampling and interpolation qualifiers have been inherited are valid only if one of the following is true:

- The type of field is compatible with the inherited qualifiers.
- The field declaration does not have a sampling, and interpolation qualifiers attribute.
- The field declaration has the same sampling, and interpolation qualifiers attribute as the inherited one.

The following example demonstrates how to specify the interpolation on structure types.

```
struct VOut {
   float4 pos [[position]];
}

struct POut {
   float4 color0;
   float4 color1;
};

[[mesh]] void mesh_function(mesh<VOut, POut, 3, 1,
                                topology::triangle> m)

struct FragmentInput {
    VOut   vin;
    POut   pin [[center _perspective]];
};
```

5.5 Per-Fragment Function Versus Per-Sample Function

You typically execute the fragment function per-fragment. The sampling attribute identifies if fragment input interpolation is per-sample or per-fragment. Similarly, the `[[sample_id]]` attribute identifies the current sample index, and the `[[color(m)]]` attribute identifies the destination fragment color or sample color (for a multisampled color attachment) value. If you use any of these attributes with arguments to a fragment function, the fragment function may execute per-sample instead of per-pixel. (The implementation may decide to only execute the code that depends on the per-sample values to execute per-sample and the rest of the fragment function may execute per-fragment.)

Only the inputs with `sample` access specified (or declared with the `[[sample_id]]` or `[[color(m)]]` attribute) differ between invocations per-fragment or per-sample, whereas other inputs still interpolate at the pixel center.

The following example uses the `[[color(m)]]` attribute to specify that this fragment function executes on a per-sample basis:

```
[[fragment]] float4
my_fragment(float2 tex_coord [[stage_in]],
            texture2d<float> img [[texture(0)]],
            sampler s [[sampler(0)]],
            float4 framebuffer [[color(0)]])
{
     return c = mix(img.sample(s, tex_coord), framebuffer,
     mix_factor);
}
```

5.6 Imageblock Attributes

iOS: Metal 2 and later support imageblocks.
macOS: Metal 2.3 and later support imageblocks for Apple silicon.
iPadOS and visionOS: Metal supports imageblocks.

This section and its subsections describe several attributes for imageblocks, including the `[[imageblock_data(type)]]` attribute that specifies input and output imageblock with an explicit imageblock layout for a fragment function.

5.6.1 Matching Data Members of Master and View Imageblocks

You can use the `[[user(name)]]` attribute to specify an attribute name for a data member of the imageblock data type for a fragment function. If the imageblock structure specified in a fragment function is a subset of the master explicit imageblock structure, the following rules match data members declared in the imageblock structure used in a fragment function with corresponding data members declared in the master explicit imageblock structure:

- Every attribute name given by `[[user(name)]]` needs to be unique for each data member in the imageblock.
- The attribute name given by `[[user(name)]]` for a data member needs to match with a data member declared in the master explicit imageblock structure, and their associated data types needs to also match.
- If the `[[user(name)]]` attribute is not specified, the data member name and type declared in the imageblock data type for a fragment function and the master imageblock structure needs to match. Additionally, the data member cannot be within a nested structure that is either within the view imageblock structure or within the master imageblock structure.

The following example shows the `[[user(name)]]` attribute in declarations of data members in master and view imageblock structures:

```
// The explicit layout imageblock data master structure.
struct IM {
     rgba8unorm<half4> a [[user(my_a), raster_order_group(0)]];
     rgb9e5<float4> b [[user(my_b), raster_order_group(0)]];
     int c [[user(my_c), raster_order_group(0)]];
     float d [[user(my_d), raster_order_group(0)]];
};

// The explicit layout imageblock data view structure for input.
struct IVIn {
     rgb9e5<float4> x [[user(my_b)]]; // Maps to IM::b
     float y [[user(my_d)]]; // Maps to IM::d
};

// The explicit layout imageblock data view structure for output.
struct IVOut {
     int z [[ user(my_c) ]]; // Maps to IM::c
```

```
};

// The fragment return structure.
struct FragOut {
     // IVOut is a view of the master IM.
     IVOut i [[ imageblock_data(IM) ]];
};

// IVIn is a view of the master IM.
[[fragment]] FragOut
my_fragment(IVIn i [[imageblock_data(IM)]], …) {
     FragOut fragOut;
     … = i.x;
     … = i.y;
     fragOut.i.z = …;
     return fragOut;
}
```

The following example shows the declaration of data members in master and view imageblock structures without the `[[user(name)]]` attribute:

```
struct IM {
     rgba8unorm<half4> a [[raster_order_group(0)]];
     rgb9e5<float4> b [[raster_order_group(0)]];
     int c [[raster_order_group(0)]];
     float d [[raster_order_group(0)]];
};

struct IVIn {
     rgb9e5<float4> b; // Maps to IM::b
     float d; // Maps to IM::d
};

struct IVOut {
     int c; // Maps to IM::c
};

struct FragOut {
     IVOut i [[imageblock_data(IM)]];
};

fragment FragOut
my_fragment(IVIn i [[imageblock_data(IM)]], …) {
     FragOut fragOut;
     … = i.b;
     … = i.d;
     fragOut.i.c = …;
     return fragOut;
}
```

You can declare nested structures in the master imageblock and view imageblock structures. The following example shows how to use nested structures in an imageblock with data members declared with the `[[user(name)]]` attribute:

```
struct A {
     rgba8unorm<half4> a [[user(A_a)]];
     rgb9e5<float4> b [[user(A_b)]];
};

struct B {
     int a [[user(B_a), raster_order_group(1)]];
     float b [[user(B_b), raster_order_group(2)]];
};

struct IM {
     A a [[user(A), raster_order_group(0)]];
     B b [[user(B)]];
};

struct IVIn {
     A x [[user(A)]]; // Maps to IM::a
};

struct IVOut {
     B y [[user(B)]]; // Maps to IM::b
     rgb9e5<float4> z [[user(A_b)]]; // Maps to IM::A::b
};

struct FragOut {
     IVOut i [[imageblock_data(IM)]];
};

fragment FragOut
my_fragment(IVIn i [[imageblock_data(IM)]], …) {
     FragOut fragOut;
     … = i.x;
     fragOut.i.y.a = …;
     fragOut.i.y.b = …;
     fragOut.i.z = …;
     return fragOut;
}
```

Each field of a view structure must correspond to exactly one master structure field. A master structure field can refer to a top-level structure field as well as a field within a nested structure. It is illegal for two or more view structure fields to alias the same master structure field.

Example of illegal use:

```
struct M  {
```

```
    struct A {
        int a [[user(x)]];
    }
    b [[user(y), raster_order_group(0)]];
};

struct V {
    int a [[user(x)]];
    M::A b [[user(y)]]; // Illegal: b aliases with a
};

fragment void
f(V i [[imageblock_data(M)]])
{…}
```

Explicit imageblock types cannot have data members declared with the `[[color(n)]]` attribute.

5.6.2 Imageblocks and Raster Order Groups

In a kernel function, a `[[raster_order_group(index)]]` attribute specified on data members of an imageblock is ignored.

In a fragment function, you must specify the `[[raster_order_group(index)]]` attribute for data members of the master explicit imageblock data structure.

If the master explicit imageblock structure contains data members that are structures, you can specify the `[[raster_order_group(index)]]` attribute for all data members in the nested structure or just the nested structure. If you specify the `[[raster_order_group(index)]]` attribute for the nested structure, then it applies to all data members of the nested structure, and no data member in the nested structure can have the `[[raster_order_group(index)]]` attribute declared.

You optionally may specify the `[[raster_order_group(index)]]` attribute for data members of an imageblock view structure, but the `[[raster_order_group(index)]]` must match the same `[[raster_order_group(index)]]` specified on the data member of the master explicit imageblock structure.

The following example shows how you can specify the `[[raster_order_group(index)]]` attribute for data members of a master imageblock. Because the `[[raster_order_group(index)]]` attribute specifies the `S` structure member of the `gBufferData` structure, you cannot use this attribute on any members of the `S` structure.

```
struct S {
    rgb9e5<half3> normal;
    float factor;
};

struct gBufferData {
```

```
    half3 color [[raster_order_group(0)]];
    S s [[raster_order_group(1)]];
    rgb11b10f<half3> lighting [[raster_order_group(2)]];
};
```

Data members declared as an array have a single raster order group associated with all members of the array. The following example shows how you can specify the `[[raster_order_group(index)]]` attribute for a data member of a master imageblock that is an array of a structure type.

```
struct S {
    rgb9e5<half3> normal;
    float factor;
};

struct IM {
    half3 color [[raster_order_group(0)]];
    S s [[raster_order_group(1)]][2];
    rgb11b10f<half3> lighting [[raster_order_group(2)]];
};
```

The following example shows an incorrect use of the `[[raster_order_group(index)]]` attribute where data member `s` is an array of a structure of type `S` with members that specify raster order groups that result in a compilation error.

```
struct S {
    rgb9e5<half3> normal [[raster_order_group(0)]];
    float factor [[raster_order_group(1)]];
};

struct IM {
    half3 color [[raster_order_group(0)]];
    S s[2]; // This causes a compilation error.
    rgb11b10f<half3> lighting [[raster_order_group(2)]];
};
```

5.6.3 Imageblock Layouts for Fragment Functions

In a fragment function, you can access the imageblock in two ways:

- As a color attachment, where the storage layout of the imageblock is not known in the fragment function. An *implicit imageblock layout* uses the existing color attachment attribute. (For more about the implicit imageblock layout, see section 5.6.3.1.)
- As a structure for declaring the imageblock data where the fragment function explicitly specifies the storage layout of the imageblock. (For more about the *explicit imageblock layout*, see section 5.6.3.2.)

5.6.3.1 Implicit Imageblock Layout for Fragment Functions

You can access the imageblock data (all the data members in the imageblock associated with a pixel) in a fragment function. Metal creates an implicit imageblock that matches the behavior of color attachments (for input to and output from a fragment function). In this mode, the types associated with the color attachments, as described in the fragment function, are the ALU types (that is, the types used to perform computations in the fragment function). The Metal runtime defines the actual pixel storage format.

When accessing the imageblock data as color attachments, you cannot declare the pixel storage types described in section 2.7 in the imageblock slice structure.

For an imageblock data implicit layout of type `T`, `T` is a structure where each member satisfies one of the following:

- Have a color attachment (see the `[[color(m)]]` attribute in Table 5.5 of section 5.2.3.4). The color index `m` needs to be unique for each member (and sub-member) of `T`.
- Be a structure type with members that satisfy the constraint on the list.

5.6.3.2 Explicit Imageblock Layout for Fragment Functions

The imageblock data with *explicit* layout has its layout declared in the shading function, not via the runtime as is done for color attachments. You declare the imageblock data for an explicit layout as a structure. Each data member of the per-fragment imageblock data can be:

- A scalar or vector, integer or floating-point data type.
- One of the pixel data types described in section 2.7.
- An array of these types.
- Or a structure built with these types.

The data members of the imageblock structure use the appropriate alignment rules for each data member type declared in the structure to determine the actual structure layout and size.

A fragment function can read one or more data members in the per-fragment imageblock data and write to one or more data members in the per-fragment imageblock data. You can declare the input and output imageblock data to a fragment function as a structure. The input and output imageblock structures can be the fully explicit imageblock structure (referred to as the master explicit imageblock structure), or be a subset of the master explicit imageblock structure (referred to as the imageblock view structure). For the latter, use the `[[imageblock_data(type)]]` attribute with the input and output imageblock data structure specified on a fragment function, where `type` specifies the fully explicit imageblock data structure.

If you specify the `[[imageblock_data]]` attribute on the input argument or output structure element without type, by default the fragment function uses the master explicit imageblock data structure on the input or output.

Example:

```
struct I {
     float a [[raster_order_group(0)]];
     };
```

```
struct FragOut {
     float c [[color(0)]];
     I i [[imageblock_data]];
};

fragment FragOut
my_fragment(I i [[imageblock_data]])
{
     FragOut fragOut;
     …
     return fragOut;
}
```

Fragment functions can access both an implicit imageblock and an explicit imageblock as separate input arguments, or as fields in a return structure.

Example:

```
struct I {
     float a [[raster_order_group(0)]];
};

struct FragOut {
     float c [[color(0)]];
     I i [[imageblock_data]];
};

[[fragment]] FragOut
my_fragment(I i [[imageblock_data]],
            float c [[color(0)]])
{
     FragOut fragOut;
     …
     return fragOut;
}
```

By default, the explicit imageblock storage is separate from the storage of the implicit imageblock. To share storage between the explicit imageblock and implicit imageblock, see section 5.6.5.

5.6.4 Imageblock Layouts in Kernel Functions

The `imageblock<T>` type (defined in the header `<metal_imageblocks>`) can only be used for arguments declared in a kernel function or in a user function that is called by a kernel function. Only a kernel function can have an argument declared as an `imageblock<T>` type. The data in an imageblock is visible only to threads in a threadgroup.

This imageblock argument to a kernel function is declared as the following templated type:

```
class imageblock_layout_explicit;
class imageblock_layout_implicit;
template<typename T, typename L>
struct imageblock;
```

With the following restrictions:

- `L` is either `imageblock_layout_explicit` or `imageblock_layout_implicit`.
- `T` is a structure; members of `T` can be any of the following:
  - scalars
  - vectors and packed vectors
  - pixel data types
  - an array with elements that are one of the types on this list
  - a structure with members that are one of the types on this list

For an imageblock with implicit layout (`imageblock_layout_implicit`), each member of the structure may have a color attachment (see the `[[color(m)]]` attribute in Table 5.5 of section 5.2.3.4). The color index `m` needs to be unique for each member (and sub-member) of `T`.

If you do not specify an imageblock layout, the compiler deduces the layout based on `T`. If `T` is not compatible with an implicit or explicit imageblock, a compiler error occurs.

Both explicit and implicit imageblocks can be arguments to a kernel function. This also makes it easy to share explicit and implicit imageblock structures between fragment and kernel functions. By default, the explicit imageblock storage is separate from the storage of the implicit imageblock. To share storage between the explicit imageblock and implicit imageblock, see section 5.6.5.

5.6.5 Aliasing Explicit and Implicit Imageblocks

By default, explicit and implicit imageblocks do not alias. To alias the allocation of an explicit imageblock with the implicit imageblock fully or partially, you can use the following attributes to specify an explicit imageblock:

```
[[alias_implicit_imageblock]]
[[alias_implicit_imageblock_color(n)]]
```

The `[[alias_implicit_imageblock]]` attribute specifies that the explicit imageblock allocation completely aliases the implicit imageblock.

The `[[alias_implicit_imageblock_color(n)]]` attribute specifies that the explicit imageblock allocation aliases the implicit imageblock starting at a specific color attachment given by `color(n)`. If `n` is a value that is between the smallest and largest declared attachments, inclusive, but `n` references an undeclared attachment, then a compile-time error occurs. If `n` is a value that exceeds the number of declared attachments, then compilation succeeds, but the attribute is ignored.

The behavior of accessing data members of an aliased implicit imageblock with an explicit imageblock is undefined if the kernel or fragment function modifies the aliased imageblock data members using the explicit imageblock and its associated member functions.

Example:

```
struct I {
     rgba8unorm<half4> a;
     rgb9e5<float4> b;
     int c;
     float d;
};

struct FragOut {
     float4 finalColor [[color(0)]];
     I i [[imageblock_data, alias_implicit_imageblock_color(1)]];
};

[[fragment]] FragOut
my_fragment(I i [[imageblock_data]], …)
{
     FragOut fragOut;
     …
     return fragOut;
}
```

5.6.6 Imageblocks and Function Constants

Do not use `[[function_constant(name)]]` with data members of an imageblock structure either as input to or as returned output from a fragment or kernel function.

5.7 Graphics Function — Signature Matching

A graphics function signature is a list of parameters that are either input to or output from a graphics function.

5.7.1 Vertex — Fragment Signature Matching

You can pass two kinds of data between a vertex and fragment function: user-defined and built-in variables.

You can declare the per-instance input to a fragment function with the `[[stage_in]]` attribute. These are output by an associated vertex function.

You can declare built-in variables with one of the attributes defined in section 5.2.3. Examples of variables that use these attributes are:

- The vertex function output (with the `[[position]]`, `[[point_size]]`, or `[[clip_distance]]` attribute).

- The rasterizer output (with the `[[point_coord]]`, `[[front_facing]]`, `[[sample_id]]`, or `[[sample_mask]]` attribute).
- A fragment function input that refers to a framebuffer color value (with `[[color]]`).

Always return a built-in variable that specifies the `[[position]]` attribute. For built-in variables with either the `[[point_size]]` or `[[clip_distance]]` attribute, that attribute must also specify the corresponding vertex function output. If they are used and read in a fragment function, the shader has undefined behavior.

You may also declare built-in variables that are rasterizer output or refer to a framebuffer color value as the fragment function input with the appropriate attribute.

You can also use the attribute `[[user(name)]]` syntax to specify an attribute name for any user-defined variable.

A vertex function and a fragment function have matching signatures if:

- There is no input argument with the `[[stage_in]]` attribute declared in the fragment function.
- For a fragment function argument declared with `[[stage_in]]`, each element in the type associated with this argument can be one of the following: a built-in variable generated by the rasterizer, a framebuffer color value passed as input to the fragment function, or a user-generated output from a vertex function. For built-in variables generated by the rasterizer or framebuffer color values, there is no requirement to associate a matching type with elements of the vertex return type. For elements that are user-generated outputs, the following rules apply:

If you specify an attribute name for an element using `[[user(name)]]`, the attribute name must match with an element in the return type of the vertex function. If you do not specify the `[[user(name)]]` attribute name, then the argument name and types must match. In either case, their corresponding data types must also match or the fragment function argument type needs to be `interpolant<T,P>`, where `T` is the element's type in the vertex return type.

Below is an example of using compatible signatures together (`my_vertex` and `my_fragment`, or `my_vertex` and `my_fragment2`) to render a primitive:

```
struct VertexOutput {
     float4 position [[position]];
     float3 normal;
     float2 texcoord;
};

vertex VertexOutput
my_vertex(…)
{
     VertexOutput v;
     …
     return v;
}

fragment float4
my_fragment(VertexOutput f [[stage_in]], …)
{
```

```
     float4 clr;
     …
     return clr;
}

fragment float4
my_fragment2(VertexOutput f [[stage_in]],
           bool is_front_face [[front_facing]], …)
{
     float4 clr;
     …
     return clr;
}
```

The following is an example of compatible signatures:

```
struct VertexOutput {
     float4 position [[position]];
     float3 vertex_normal [[user(normal)]];
     float2 texcoord [[user(texturecoord)]];
};

struct FragInput {
     float3 frag_normal [[user(normal)]];
     float4 position [[position]];
     float4 framebuffer_color [[color(0)]];
     bool is_front_face [[front_facing]];
};

vertex VertexOutput
my_vertex(…)
{
     VertexOutput v;
     …
     return v;
}

fragment float4
my_fragment(FragInput f [[stage_in]], …)
{
     float4 clr;
     …
     return clr;
}
```

The following is an example of compatible signatures:

```
struct VertexOutput {
```

```
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

vertex VertexOutput
my_vertex(…)
{
    VertexOutput v;
    …
    return v;
}

fragment float4
my_fragment(float4 p [[position]], …)
{
    float4 clr;
    …
    return clr;
}
```

Below is an example of incompatible signatures. The data type of `normal` in `VertexOutput` (`float3`) does not match the type of `normal` in `FragInput` (`half3`):

```
struct VertexOutput {
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

struct FragInput {
    float4 position [[position]];
    half3 normal;
};

vertex VertexOutput
my_vertex(…)
{
    VertexOutput v;
    …
    return v;
}

fragment float4
my_fragment(FragInput f [[stage_in]], …)
{
    float4 clr;
```

```
    …
    return clr;
}
```

Below is another example of incompatible signatures. The attribute index of `normal` in `VertexOutput` (`normal`) does not match the index of `normal` in `FragInput` (`foo`):

```
struct VertexOutput {
    float4 position [[position]];
    float3 normal [[user(normal)]];
    float2 texcoord [[user(texturecoord)]];
};

struct FragInput {
    float3 normal [[user(foo)]];
    float4 position [[position]];
};

vertex VertexOutput
my_vertex_shader(…)
{
    VertexOutput v;
    …
    return v;
}

fragment float4
my_fragment_shader(FragInput f [[stage_in]], …)
{
    float4 clr;
    …
    return clr;
}
```

5.7.2 Mesh – Fragment Signature Matching

You can pass the two kinds of data from vertex (`V`) and primitive (`P`) of `mesh<V, P, NV, NP, t>` from the mesh function to the fragment function: user-defined and built-in variables. The per-vertex mesh outputs defined in vertex (`V`) are always interpolated, whereas the per-primitive mesh outputs defined in primitive (`P`) are never interpolated. Due to this difference, the rules for signature matching of user-generated output have been adjusted from those described in section 5.7.1 Vertex — Fragment Signature Matching.

A given fragment input *matches* a user-generated mesh output from vertex (`V`) and primitive (`P`) if the following is true:

- If you specify an attribute name for an element using `[[user(name)]]`, the attribute name must match with an element in the return type of the mesh output.
- If you do not specify the `[[user(name)]]` attribute name, then the argument name and types must match.

In either case, their corresponding data types must also match, or the fragment function argument type needs to be `interpolant<T, P>`, where `T` is the element's type in the vertex return type.

A mesh function and a fragment function have matching signatures for user-generated inputs with user-generated mesh outputs if:

- For a given user-generated fragment input with a `flat` interpolation:
    - There is a matching per-primitive mesh output, and the output is propagated to the fragment input without interpolation.
    - There is a matching per-vertex mesh output, and the output for the provoking vertex is propagated to the fragment input without interpolation.
- For a given user-generated fragment input with a non `flat` interpolation:
    - There is a matching per-primitive mesh output, and the output is propagated to the fragment input without interpolation.
    - There is a matching per-vertex mesh output, and the output is interpolated across the primitive in the same method as nonflat vertex outputs are interpolated.

5.8 Program Scope Function Constants

All OS: Metal 1.2 and later support function constants. In Metal 2 and later, you can use a function constant to specify the binding number for a resource (see section 5.8.1.4), to specify the index for the `color()` or `raster_order_group` attributes (section 5.8.1.5), and to identify that a structure element is optional (section 5.8.1.6).

Function constants enable the generation of multiple variants of a function. Without using function constants, you can compile one function many times with different preprocessor macro defines to enable different features (an *ubershader*). Using preprocessor macros for ubershaders with offline compiling can result in many variants and a significant increase in the size of the shading function library assets. Function constants provide the same ease of use as preprocessor macros but moves the generation of the specific variants to the creation of the pipeline state, so you don't have to compile the variants offline.

5.8.1 Specifying Program Scope Function Constants

Program scope variables declared with (or initialized with) the following attribute are *function constants*:

```
[[function_constant(index)]]
```

The value `index` needs to be between 0 and 65535.

In Metal, function constants can:

- Control code paths that get compiled.
- Specify the optional arguments of a function (graphics, kernel, or user functions).
- Specify optional elements of a structure with the `[[stage_in]]` attribute.

You don't initialize function constants in the Metal function source. Instead, you specify their values when creating a specialized function (`MTLFunction`) using an `MTLFunctionDescriptor` in the Metal API. The `index` value specifies a location index that can refer to the function constant variable (instead of by its name) in the runtime.

Examples:

```
constant int a [[function_constant(0)]];
constant bool b [[function_constant(2)]];
```

Function constants can only be a scalar or vector type. Using a user-defined type or an array of a scalar or vector type for a function constant results in a compilation error.

You specify the value of function constants `a` and `b` during the creation of the render or compute pipeline state.

You can also use function constants to initialize variables in program scope declared in the constant address space.

Examples:

```
constant int a [[function_constant(0)]];
constant bool b [[function_constant(2)]];
constant bool c = ((a == 1) && b);
constant int d = (a * 4);
```

You can use the following built-in function to determine if a function constant has been defined and is available. `name` refers to the function constant variable.

```
bool is_function_constant_defined(name)
```

Returns `true` if the function constant variable is defined and `false` otherwise.

If a function constant variable value is not defined during the creation of the pipeline state and if the graphics or kernel function specified with the render or compute pipeline state uses these function constants, `is_function_constant_defined(name)` returns `false`.

5.8.1.1 Function Constants to Control Code Paths to Compile

Consider the following function which uses preprocessor macros for function constants:

```
struct VertexOutput {
     float4 position [[position]];
     float4 color;
};

struct VertexInput {
     float4 position [[attribute(0)]];
     float4 offset   [[attribute(1)]];
     float4 color    [[attribute(2)]];
};
```

```
vertex VertexOutput
myVertex(VertexInput vIn [[stage_in]])
{
     VertexOutput vOut;

     vOut.position = vIn.position;
     #ifdef OFFSET_DEFINED
          vOut.position += vIn.offset;
     #endif

     #ifdef COLOR_DEFINED
          vOut.color = vIn.color;
     #else
          vOut.color = float4(0.0f);
     #endif

     return vOut;
}
```

The corresponding function written using function constant variables is:

```
constant bool offset_defined [[function_constant(0)]];
constant bool color_defined [[function_constant(1)]];

vertex VertexOutput
myVertex(VertexInput vIn [[stage_in]])
{
     VertexOutput vOut;

     vOut.position = vIn.position;
     if (offset_defined)
          vOut.position += vIn.offset;

     if (color_defined)
          vOut.color = vIn.color;
     else
          vOut.color = float4(0.0f);

     return vOut;
}
```

5.8.1.2 Function Constants when Declaring the Arguments of Functions

You can declare an argument to a graphics, kernel, or other user function with the `[[function_constant(name)]]` attribute to identify that the argument is optional. The `name` attribute refers to a function constant variable. If the value of the function constant variable given by `name` is `nonzero` or `true` (determined during creation of the pipeline state), the declaration of the argument is in the function signature. If the value of the function constant

variable given by `name` is `0` or `false`, the argument is *not* declared in the function signature. If `name` refers to a function constant variable that has not been defined (determined during the creation of the pipeline state), the behavior is the same as if the value of `is_function_constant_defined(name)` is `false`.

Consider the following fragment function that uses preprocessor macros in its function declaration:

```
fragment half4
myFragment(
    constant GlobalUniformData *globalUniform [[buffer(0)]],
    constant RenderUniformData_ModelWithLightmap
                           *renderUniform     [[buffer(1)]],
    constant MaterialUniformData
                           *materialUniform [[buffer(2)]],
    texture2d<float> DiffuseTexture [[texture(0)]],
    texture2d<float> LightmapTexture [[texture(1)]],
    texture2d<float> FogTexture [[texture(3)]],
#ifdef MED_QUALITY
    texture2d<float> LookupTexture [[texture(4)]],
#endif
#ifdef REALTIME_SHADOW
    texture2d<float> RealtimeShadowMapTexture [[texture(10)]],
#endif
    sampler DiffuseTextureSampler [[sampler(0)]],
    sampler LightmapTextureSampler [[sampler(1)]],
    sampler FogTextureSampler [[sampler(3)]],
#ifdef MED_QUALITY
    sampler LookupTextureSampler [[sampler(4)]],
#endif
#ifdef REALTIME_SHADOW
    sampler RealtimeShadowMapTextureSampler [[sampler(10)]],
#endif
    VertexOutput fragIn [[stage_in]])
```

Here is the corresponding fragment function, after using function constants instead of `#ifdef` statements to rewrite the previous code:

```
constant bool realtime_shadow [[function_constant(0)]];
constant bool med_quality [[function_constant(1)]];
constant bool med_quality_defined =
is_function_constant_defined(med_quality);
constant bool realtime_shadow_defined =
is_function_constant_defined(realtime_shadow);

fragment half4
myFragment(
    constant GlobalUniformData *globalUniform [[buffer(0)]],
    constant RenderUniformData_ModelWithLightmap
                               *renderUniform [[buffer(1)]],
```

```
    constant MaterialUniformData
                             *materialUniform [[buffer(2)]],
    texture2d<float> DiffuseTexture [[texture(0)]],
    texture2d<float> LightmapTexture [[texture(1)]],
    texture2d<float> FogTexture [[texture(3)]],
    texture2d<float> LookupTexture [[texture(4),
                       function_constant(med_quality_defined)]],
    texture2d<float> RealtimeShadowMapTexture [[texture(10),
                       function_constant(realtime_shadow_defined)]],
    sampler DiffuseTextureSampler [[sampler(0)]],
    sampler LightmapTextureSampler [[sampler(1)]],
    sampler FogTextureSampler [[sampler(3)]],
    sampler LookupTextureSampler [[sampler(4),
                       function_constant(med_quality_defined)]],
    sampler RealtimeShadowMapTextureSampler [[sampler(10),
                       function_constant(realtime_shadow_defined)]],
    VertexOutput fragIn [[stage_in]])
```

Below is another example that shows how to use function constants with arguments to a function:

```
constant bool hasInputBuffer [[function_constant(0)]];

kernel void kernelOptionalBuffer(
  device int *input [[buffer(0),function_constant(hasInputBuffer)]],
  device int *output [[buffer(1)]],
  uint tid [[thread_position_in_grid]])
{
    if (hasInputBuffer)
        output[tid] = inputA[0] * tid;
    else
        output[tid] = tid;
}
```

5.8.1.3 Function Constants for Elements of an Input Assembly Structure

You can use the `[[function_constant(name)]]` attribute to specify elements of an input assembly structure (declared with the `[[stage_in]]` attribute) as optional. If the value of the function constant variable given by `name` is `nonzero` or `true` (determined during the creation of the render or compute pipeline state), the element in the structure is declared in the function signature. If the value of the function constant variable given by `name` is `0` or `false`, the element is not declared in the structure.

Example:

```
constant bool offset_defined [[function_constant(0)]];
constant bool color_defined [[function_constant(1)]];

struct VertexOutput {
     float4 position [[position]];
```

```
    float4 color;
};
struct VertexInput {
    float4 position [[attribute(0)]];
    float4 offset   [[attribute(1),
    function_constant(offset_defined)]];
    float4 color    [[attribute(2),
    function_constant(color_defined)]];
};

vertex VertexOutput
myVertex(VertexInput vIn [[stage_in]])
{
    VertexOutput vOut;

    vOut.position = vIn.position;
    if (offset_defined)
        vOut.position += vIn.offset;

    if (color_defined)
        vOut.color = vIn.color;
    else
        vOut.color = float4(0.0f);

    return vOut;
}
```

5.8.1.4 Function Constants for Resource Bindings

All OS: Metal 2 and later support using a function constant to specify resource bindings.

An argument to a graphics or kernel functions that is a resource (buffer, texture, or sampler) can use a function constant to specify its binding number. The function constant needs to be a scalar integer type.

Example:

```
constant int indexA [[function_constant(0)]];
constant int indexB = indexA + 2;
constant int indexC [[function_constant(1)]];
constant int indexD [[function_constant(2)]];

[[kernel]] void
my_kernel(constant UserParams& params [[buffer(indexA)]],
          device T * p [[buffer(indexB)]],
          texture2d<float> texA [[texture(indexC)]],
          sampler s [[sampler(indexD)]], …)
{…}
```

5.8.1.5 Function Constants for Color Attachments and Raster Order Groups

All OS: Metal 2 and later support using a function constant to specify a color attachment or a raster order group attribute index.

The `[[color(n)]]` or `[[raster_order_group(index)]]` index can also be a function constant. The function constant used needs to be a scalar integer type.

Example:

```
constant int colorAttachment0 [[function_constant(0)]];
constant int colorAttachment1 [[function_constant(1)]];
constant int group0 [[function_constant(2)]];

struct FragmentOutput {
     float4 color0 [[color(colorAttachment0)]];
     float4 color1 [[color(colorAttachment1)]];
};

[[fragment]] FragmentOutput
my_fragment(texture2d<float> texA [[texture(0),
raster_order_group(group0)]], …)
{…}
```

5.8.1.6 Function Constants with Elements of a Structure

All OS: Metal 2 and later support using a function constant to identify that a structure element is optional.

To identify that an element of a structure is optional, you can specify the `[[function_constant(name)]]` attribute with elements of a structure that is the return type of a graphics or user function or is passed by value as an argument to a kernel, graphics, or user function. The behavior is similar to function constants for elements with the `[[stage_in]]` attribute, as described in section 5.8.1.3.

If the value of the function constant variable given by `name` is nonzero or `true` (determined during the render or compute pipeline state creation), the element in the structure is declared in the function signature. If the value of the function constant variable given by `name` is 0 or `false`, the element is not considered to be declared in the structure. If `name` refers to a function constant variable that is undefined, the behavior is the same as if `is_function_constant_defined(name)` returns `false`.

5.9 Program Scope Global Built-ins and Bindings

In Metal 3.1 and later, you can define global variables using attributes defined in Table 5.8 and use them in a kernel (including tile), mesh, or object context. The global variables cannot be used in a dynamic library or a separately compiled binary function. In Metal 3.2 and later, you can use global variables in a dynamic library or a separately compiled binary function for Apple silicon.

Example:

```
uint2 gid [[thread_position_in_grid]];

float4 get_color(texture2d<float> texInput, sampler s) {
 return texInput.sample(s, float2(gid));
}

[[kernel]] void my_kernel(texture2d<float> texInput, sampler s, ...){
 auto color = get_color(texInput, s);
  ...
}
```

In Metal 3.2 and later, you can declare `device`, `constant`, and `threadgroup` buffers, textures, and samplers in the program scope (see section 5.2). Unlike when passing as arguments in a shader, you can't assume different global variables are non-aliased. Instead, specify the binding indexes because the system can't set them automatically.

Example:

```
device void * constant b_d      [[ buffer(0) ]];
constant void * constant b_c    [[ buffer(1) ]];
threadgroup void * constant b_t [[ threadgroup(2) ]];
texture2d<float> constant t     [[ texture(0) ]];
sampler constant s              [[ sampler(0) ]];
constant array<sampler, 4> ss   [[ sampler(1) ]];
```

It's possible to declare global bindings with external linkage, but you need to annotate them with the resource binding and have a complete type. Note that the declaration and the definition binding and type must match.

```
// Declaration
extern constant texture2d<float> t [[ texture(0) ]];

// Definition
constant texture2d<float> t [[ texture(0) ]];
```

You can bind a resource to multiple global variables if they share the same type and binding index.

Example:

```
constant texture2d<float, access::write> t_w_1 [[texture(1)]];
// legal
constant texture2d<float, access::write> t_w_2 [[texture(1)]];
// illegal!
constant texture2d<float, access::read_write> t_w_3 [[texture(1)]];
```

5.10 Per-Primitive Viewport and Scissor Rectangle Index Selection

macOS: Metal 2 and later support the `viewport_array_index` attribute.
iOS: Metal 2.1 and later support the `viewport_array_index` attribute.
iPadOS and visionOS: Metal supports the `viewport_array_index` attribute.

The `[[viewport_array_index]]` attribute supports built-in variables as both vertex output and fragment input. With `[[viewport_array_index]]`, the vertex function output specifies the rasterization viewport and scissor rectangle from the arrays specified by the `setViewports:count:` and `setScissorRects:count:` framework calls, respectively.

The unclamped value of the vertex function output for `[[viewport_array_index]]` is provided as input to the fragment function, even if the value is out of range.

The behavior of the fragment function with an unclamped `[[viewport_array_index]]` value depends upon the implementation. Either Metal can render every primitive to viewport/scissor rectangle 0, regardless of the passed value, or Metal can render to the nth viewport/scissor rectangle, where n is the clamped value. (Hardware that does not support this feature acts as only one viewport and one scissor rectangle are permitted, so the value for `[[viewport_array_index]]` is 0.)

You can specify `[[viewport_array_index]]` in a post-tessellation vertex function. You cannot specify `[[viewport_array_index]]` in the tessellation factor buffer.

Specifying `[[viewport_array_index]]` as fragment function input counts against the number of input assembly components available. (Input assembly components are the fragment function inputs declared with the `stage_in` qualifier.)

You must return the same value of `[[viewport_array_index]]` for every vertex in a primitive. If the values differ, the behavior and the value passed to the fragment function are undefined. The same behavior applies to primitives generated by tessellation.

5.11 Additional Restrictions

MSL functions and arguments have these additional restrictions:

- Writes to a buffer from a vertex function are not guaranteed to be visible to reads from the associated fragment function of a given primitive.
- If a vertex function does writes to one or more buffers or textures, its return type needs to be `void`.
- The return type of a vertex or fragment function cannot include an element that is a packed vector type, matrix type, a structure type, a reference, or a pointer to a type.
- The number of inputs to a fragment function declared with the `stage_in` attribute is limited. The input limits differ for different feature sets. The Metal Feature Set Tables lists the specific limits below “Implementation Limits by GPU Family”. (An input vector counts as `n` input scalars, where `n` is the number of components in the vector.)

- The argument type for arguments to a graphics or kernel function cannot be a derived class. Also, the type of an argument to a graphics function that is declared with the `stage_in` attribute cannot be a derived class.

6 Metal Standard Library

This chapter describes functions in the Metal Standard Library (MSLib).

6.1 Namespace and Header Files

Metal declares all MSLib functions and enumerations in the `metal` namespace. In addition to the header files described in the MSLib functions, the `<metal_stdlib>` header is available and can access all the functions supported by the MSLib.

6.2 Placement New

All OS: Metal 4.1 and later support placement `new`.

The header `<metal_common>` defines placement `new`.

```
thread void *operator new(size_t size, thread void *ptr)
thread void *operator new[](size_t size, thread void *ptr)

threadgroup void *operator new(size_t size, threadgroup void *ptr)
threadgroup void *operator new[](size_t size,
                                 threadgroup void *ptr)

device void *operator new(size_t size, device void *ptr)
device void *operator new[](size_t size, device void *ptr)
```

When calling placement new, you must put the address space qualifier on the type. For example,

```
[[kernel]] void new_example(device void *ptr) {
  device int *iptr = new (ptr) device int;
  *iptr = 7;
}
```

6.3 Common Functions

The header `<metal_common>` defines the functions in Table 6.1. `T` is one of the scalar or vector `half` or `float` floating-point types.

Table 6.1. Common functions in the Metal standard library

Built-in common functions	Description
`T clamp(T x, T minval, T maxval)`	Returns `fmin(fmax(x, minval), maxval)`. Results are undefined if `minval > maxval`.

`T mix(T x, T y, T a)`	Returns the linear blend of `x` and `y` implemented as:
	`x + (y – x) * a`
	
	or:
	
	`(1 –a) * x + a * y`
	
	`a` needs to be a value in the range `0.0` to `1.0`. If `a` is not in the range `0.0` to `1.0`, the return values are undefined.

`T saturate(T x)`	Clamp the specified value within the range of `0.0` to `1.0`.
`T sign(T x)`	Returns `1.0` if `x > 0`, `–0.0` if `x = –0.0`, `+0.0` if `x = +0.0`, or `–1.0` if `x < 0`. Returns `0.0` if `x` is a NaN.

`T smoothstep(T edge0, T edge1, T x)`	Returns `0.0` if `x <= edge0`, and `1.0` if `x >= edge1` and performs a smooth Hermite interpolation between `0` and `1` when `edge0 < x < edge1`. This is useful in cases where you want a threshold function with a smooth transition.
	
	This is equivalent to:
	`t = clamp((x – edge0)/(edge1 – edge0), 0, 1);`
	`return t * t * (3 – 2 * t);`
	
	Results are undefined if `edge0 >= edge1` or if `x`, `edge0`, or `edge1` is a NaN.

`T step(T edge, T x)`	Returns `0.0` if `x < edge`; otherwise, it returns `1.0`.

For single precision floating-point, Metal also supports a precise and fast variant of the following common functions: `clamp` and `saturate`. The difference between the Fast and precise function variants handle NaNs differently. In the fast variant, the behavior of NaNs is undefined, whereas the precise variants follow the IEEE 754 rules for NaN handling. The `ffast-math` compiler option (refer to section 1.6.3) selects the appropriate variant when

compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces provide an explicit way to select the fast or precise variant of these common functions.

6.4 Integer Functions

The header `<metal_integer>` defines the integer functions in Table 6.2. `T` is one of the scalar or vector integer types. `Tu` is the corresponding unsigned scalar or vector integer type. `T32` is one of the scalar or vector 32-bit `int` or `uint` types.

Table 6.2. Integer functions in the Metal standard library

Built-in integer functions	Description
`T abs(T x)`	Returns `|x|`.
`Tu absdiff(T x, T y)`	Returns `|x−y|` without modulo overflow.
`T addsat(T x, T y)`	Returns `x + y` and saturates the result.
`T clamp(T x, T minval, T maxval)`	Returns `min(max(x, minval), maxval)`. Results are undefined if `minval > maxval`.
`T clz(T x)`	Returns the number of leading `0`-bits in `x`, starting at the most significant bit position. If `x` is `0`, returns the size in bits of the type of `x` or component type of `x`, if `x` is a vector
`T ctz(T x)`	Returns the count of trailing `0`-bits in `x`. If `x` is `0`, returns the size in bits of the type of `x` or if `x` is a vector, the component type of `x`.
`vec<`T_{iuN}`, 2> deinterleave(`T_{iu2N}` v)` All OS: Metal 4.1 and later.	Deinterleaves the `2n`-bit value v (where n is the bit width of T_{iuN}) into a vector where the first vector element contains the bits at even positions and the second vector element contains the bits at odd positions. Supported type pairs are: (T_{iu2N}: `ushort`, T_{iuN}: `uchar`) (T_{iu2N}: `uint`, T_{iuN}: `ushort`) (T_{iu2N}: `ulong`, T_{iuN}: `uint`)

Built-in integer functions	Description
`T extract_bits(T x, uint offset, uint bits)` All OS: Metal 1.2 and later.	Extract bits [`offset`, `offset+bits-1`] from `x`, returning them in the least significant bits of the result. For unsigned data types, the most significant bits of the result are set to zero. For signed data types, the most significant bits are set to the value of bit `offset+bits-1`. If `bits` is zero, the result is zero. If the sum of `offset` and `bits` is greater than the number of bits used to store the operand, the result is undefined.
`T hadd(T x, T y)`	Returns `(x + y) >> 1`. The intermediate sum does not modulo overflow.
`T insert_bits(T base, T insert, uint offset, uint bits)` All OS: Metal 1.2 and later.	Returns the insertion of the `bits` least-significant bits of `insert` into `base`. The result has bits [`offset`, `offset+bits-1`] taken from bits [`0`, `bits-1`] of `insert`, and all other bits are taken directly from the corresponding bits of `base`. If `bits` is zero, the result is `base`. If the sum of `offset` and `bits` is greater than the number of bits used to store the operand, the result is undefined.
T_{iu2N} `interleave(`T_{iuN} `even, `T_{iuN} `odd)` T_{iu2N} `interleave(vec<`T_{iuN}`, 2> v)` All OS: Metal 4.1 and later.	Interleaves two `n`-bit values (`even` and `odd`, where `n` is the bit width of T_{iuN}) to form a `2n`-bit result. The `even` bits of the result come from `even` (or the first vector element) and the `odd` bits come from `odd` (or the second vector element). Supported type pairs are: (`ushort`, `uchar`) (`uint`, `ushort`) (`ulong`, `uint`)
`T32 mad24(T32 x, T32 y, T32 z)` All OS: Metal 2.1 and later.	Uses `mul24` to multiply two 24-bit integer values `x` and `y`, adds the 32-bit integer result to the 32-bit integer `z`, and returns that sum.
`T madhi(T a, T b, T c)`	Returns `mulhi(a, b) + c`.
`T madsat(T a, T b, T c)`	Returns `a * b + c` and saturates the result.

Built-in integer functions	Description
`T max(T x, T y)`	Returns `y` if `x < y`, otherwise it returns `x`.
`T max3(T x, T y, T z)` All OS: Metal 2.1 and later.	Returns `max(x, max(y, z))`.
`T median3(T x, T y, T z)` All OS: Metal 2.1 and later.	Return the middle value of `x`, `y`, and `z`.
`T min(T x, T y)`	Returns `y` if `y < x`, otherwise, it returns `x`.
`T min3(T x, T y, T z)` All OS: Metal 2.1 and later.	Returns `min(x, min(y, z))`.
`T32 mul24(T32 x, T32 y)` All OS: Metal 2.1 and later.	Multiplies two 24-bit integer values `x` and `y` and returns the 32-bit integer result. `x` and `y` are 32-bit integers but only the low 24 bits perform the multiplication. (See details following this table.)
`T mulhi(T x, T y)`	Computes `x * y` and returns the high half of the product of `x` and `y`.
`T popcount(T x)`	Returns the number of nonzero bits in `x`.
`T reverse_bits(T x)` All OS: Metal 2.1 and later.	Returns the reversal of the bits of `x`. The bit numbered n of the result is taken from bit (`bits` – 1) – `n` of `x`, where bits is the total number of bits used to represent `x`.
`T rhadd(T x, T y)`	Returns `(x + y + 1) >> 1`. The intermediate sum does not modulo overflow.
`T rotate(T v, T i)`	For each element in `v`, the bits are shifted left by the number of bits given by the corresponding element in `i`. Bits shifted off the left side of the element are shifted back in from the right.
`T subsat(T x, T y)`	Returns `x – y` and saturates the result.

The `mul24` function only operates as described if `x` and `y` are signed integers and `x` and `y` are in the range `[–2^23, 2^23 –1]`, or if `x` and `y` are unsigned integers and `x` and `y` are in the range `[0, 2^24 –1]`. If `x` and `y` are not in this range, the multiplication result is implementation-defined.

6.5 Relational Functions

The header `<metal_relational>` defines the relational functions in Table 6.3. `T` is one of the scalar or vector floating-point types including `bfloat` types. `Ti` is one of the scalar or vector integer or Boolean types. `Tb` only refers to the scalar or vector Boolean types.

Table 6.3. Relational functions in the Metal standard library

Built-in relational functions	Description
`bool all(Tb x)`	Returns true only if all components of `x` are true.
`bool any(Tb x)`	Returns true only if any component of `x` are true.
`Tb isfinite(T x)`	Test for finite value.
`Tb isinf(T x)`	Test for infinity value (positive or negative).
`Tb isnan(T x)`	Test for a NaN.
`Tb isnormal(T x)`	Test for a normal value.
`Tb isordered(T x, T y)`	Test if arguments are ordered. `isordered()` takes arguments `x` and `y` and returns the result `(x == x) && (y == y)`.
`Tb isunordered(T x, T y)`	Test if arguments are unordered. `isunordered()` takes arguments `x` and `y` and returns true if `x` or `y` is NaN; otherwise, returns `false`.
`Tb not(Tb x)`	Returns the componentwise logical complement of x.

`T select(T a, T b, Tb c)`
 	
`Ti select(Ti a,`
`Ti b,`
`Tb c)`
	For each component of a vector type,
	`result[i] = c[i] ? b[i] : a[i]`
	
	For a scalar type,
	`result = c ? b : a`

`Tb signbit(T x)`	Test for sign bit. Returns true if the sign bit is set for the floating-point value in `x`; otherwise, returns `false`.

6.6 Math Functions

The header `<metal_math>` defines the math functions in Table 6.4. `T` is one of the scalar or vector `half` or `float` floating-point types. `Ti` refers only to the scalar or vector integer types.

Table 6.4. Math functions in the Metal standard library

Built-in math functions	Description
`T acos(T x)`	Compute arc cosine of `x`.
`T acosh(T x)`	Compute inverse hyperbolic cosine of `x`.
`T asin(T x)`	Compute arc sine function of `x`.

Built-in math functions	Description
`T asinh(T x)`	Compute inverse hyperbolic sine of `x`.
`T atan(T y_over_x)`	Compute arc tangent of `x`.
`T atan2(T y, T x)`	Compute arc tangent of `y` over `x`.
`T atanh(T x)`	Compute hyperbolic arc tangent of `x`.
`T ceil(T x)`	Round `x` to integral value using the round to positive infinity rounding mode.
`T copysign(T x, T y)`	Return `x` with its sign changed to match the sign of `y`.
`T cos(T x)`	Compute cosine of `x`.
`T cosh(T x)`	Compute hyperbolic cosine of `x`.
`T cospi(T x)`	Compute `cos(πx)`.
`T divide(T x, T y)`	Compute `x / y`.
`T exp(T x)`	Exponential base e function.
`T exp2(T x)`	Exponential base 2 function.
`T exp10(T x)`	Exponential base 10 function.
`T fabs(T x)` `T abs(T x)`	Compute absolute value of a floating-point number.
`T fdim(T x, T y)`	`x − y` if `x > y`; +0 if `x <= y`.
`T floor(T x)`	Round `x` to integral value using the round to negative infinity rounding mode.
`T fma(T a, T b, T c)`	Returns the correctly rounded floating-point representation of the sum of `c` with the infinitely precise product of `a` and `b`. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
`T fmax(T x, T y)` `T max(T x, T y)`	Returns `y` if `x < y`, otherwise returns `x`. If one argument is a NaN, `fmax()` returns the other argument. If both arguments are NaNs, `fmax()` returns a NaN. If `x` and `y` are denormals and the GPU doesn't support denormals, either value may be returned.

Built-in math functions	Description

`T fmax3(T x, T y, T z)`
`T max3(T x, T y, T z)`
All OS: Metal 2.1 and later
	Returns `fmax(x, fmax(y, z))`.

`T fmedian3(T x, T y, T z)`
All OS: Metal 1 and later
`T median3(T x, T y, T z)`
All OS: Metal 2.1 and later
	Returns the middle value of `x`, `y`, and `z`. (If one or more values are NaN, see discussion after this table.)

`T fmin(T x, T y)` `T min(T x, T y)`	Returns `y` if `y < x`, otherwise it returns `x`. If one argument is a NaN, `fmin()` returns the other argument. If both arguments are NaNs, `fmin()` returns a NaN. If `x` and `y` are denormals and the GPU doesn't support denormals, either value may be returned.

`T fmin3(T x, T y, T z)`
`T min3(T x, T y, T z)`
All OS: Metal 2.1 and later
	Returns `fmin(x, fmin(y, z))`.

`T fmod(T x, T y)`	Returns `x - y * trunc(x/y)`.
`T fract(T x)`	Returns the fractional part of `x` that is greater than or equal to 0 or less than 1.
`T frexp(T x, Ti &exponent)`	Extract mantissa and exponent from `x`. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp.
`Ti ilogb(T x)`	Return the exponent as an integer value.
`T ldexp(T x, Ti k)`	Multiply `x` by 2 to the power `k`.
`T log(T x)`	Compute the natural logarithm of `x`.
`T log2(T x)`	Compute the base 2 logarithm of `x`.
`T log10(T x)`	Compute the base 10 logarithm of `x`.
`T modf(T x, T &intval)`	Decompose a floating-point number. The `modf` function breaks the argument `x` into integral and fractional parts, each of which has the same sign as the argument. Returns the fractional value. The integral value is returned in `intval`.
`T nextafter(T x, T y)` All OS: Metal 3.1 and later	Return next representable floating-point value after `x` in the direction of `y`. If `x` equals `y`, return

Built-in math functions	Description
	`y`. Note that if both `x` and `y` represent the floating-point zero values, the result has sign of `y`. If either `x` or `y` is NaN, return NaN.
`T pow(T x, T y)`	Compute `x` to the power `y`.
`T powr(T x, T y)`	Compute `x` to the power `y`, where `x` is `>= 0`.
`T rint(T x)`	Round `x` to integral value using round ties to even rounding mode in floating-point format.
`T round(T x)`	Return the integral value nearest to `x`, rounding halfway cases away from zero.
`T rsqrt(T x)`	Compute inverse square root of `x`.
`T sin(T x)`	Compute sine of `x`.
`T sincos(T x, T &cosval)`	Compute sine and cosine of `x`. Return the computed sine in the function return value, and return the computed cosine in `cosval`.
`T sinh(T x)`	Compute hyperbolic sine of `x`.
`T sinpi(T x)`	Compute `sin(πx)`.
`T sqrt(T x)`	Compute square root of `x`.
`T tan(T x)`	Compute tangent of `x`.
`T tanh(T x)`	Compute hyperbolic tangent of `x`.
`T tanpi(T x)`	Compute `tan(πx)`.
`T trunc(T x)`	Round `x` to integral value using the round toward zero rounding mode.

For `fmedian3`, if all values are NaN, return NaN. Otherwise, treat NaN as missing data and remove it from the set. If two values are NaN, return the non-NaN value. If one of the values is NaN, the function can return either non-NaN value.

For single precision floating-point, Metal supports two variants for most of the math functions listed in Table 6.4: the precise and the fast variants. See Table 8.2 in section 8.4 for the list of fast math functions and their precision. The `ffast-math` compiler option (refer to section 1.6.3) selects the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces provide an explicit way to select the fast or precise variant of these math functions for single precision floating-point.

Examples:

```
float x;
```

```
float a = sin(x); // Use fast or precise version of sin based on
                  // whether you specify –ffast-math as
                  // compile option.
float b = fast::sin(x); // Use fast version of sin().
float c = precise::cos(x); // Use precise version of cos().
```

All OS: Metal 1.2 and later support the constants in Table 6.5 and Table 6.6.

Table 6.5 lists available symbolic constants with values of type `float` that are accurate within the precision of a single-precision floating-point number.

Table 6.5. Constants for single-precision floating-point math functions

Constant name	Description
`MAXFLOAT`	Value of maximum noninfinite single precision floating-point number.
`HUGE_VALF`	A positive float constant expression. `HUGE_VALF` evaluates to +infinity.
`INFINITY`	A constant expression of type float representing positive or unsigned infinity.
`NAN`	A constant expression of type float representing a quiet NaN.
`M_E_F`	Value of e
`M_LOG2E_F`	Value of $\log_2 e$
`M_LOG10E_F`	Value of $\log_{10} e$
`M_LN2_F`	Value of $\log_e 2$
`M_LN10_F`	Value of $\log_e 10$
`M_PI_F`	Value of π
`M_PI_2_F`	Value of π / 2
`M_PI_4_F`	Value of π / 4
`M_1_PI_F`	Value of 1 / π
`M_2_PI_F`	Value of 2 / π
`M_2_SQRTPI_F`	Value of 2 / √π
`M_SQRT2_F`	Value of √2
`M_SQRT1_2_F`	Value of 1 / √2

Table 6.6 lists available symbolic constants with values of type `half` that are accurate within the precision of a half-precision floating-point number.

Table 6.6. Constants for half-precision floating-point math functions

Constant name	Description
`MAXHALF`	Value of maximum noninfinite half precision floating-point number.
`HUGE_VALH`	A positive half constant expression. `HUGE_VALH` evaluates to +infinity.
`M_E_H`	Value of e
`M_LOG2E_H`	Value of $\log_2 e$
`M_LOG10E_H`	Value of $\log_{10} e$
`M_LN2_H`	Value of $\log_e 2$
`M_LN10_H`	Value of $\log_e 10$
`M_PI_H`	Value of π
`M_PI_2_H`	Value of π / 2
`M_PI_4_H`	Value of π / 4
`M_1_PI_H`	Value of 1 / π
`M_2_PI_H`	Value of 2 / π
`M_2_SQRTPI_H`	Value of 2 / √π
`M_SQRT2_H`	Value of √2
`M_SQRT1_2_H`	Value of 1 / √2

Table 6.7 lists available symbolic constants with values of type `bfloat` that are accurate within the precision of a brain floating-point number.

Table 6.7. Constants for brain floating-point math functions

Constant name	Description
`MAXBFLOAT`	Value of maximum noninfinite `bfloat` floating-point number.

Constant name	Description
`HUGE_VALBF`	A positive half constant expression. `HUGE_VALBF` evaluates to +infinity.
`M_E_BF`	Value of e
`M_LOG2E_BF`	Value of $\log_2 e$
`M_LOG10E_BF`	Value of $\log_{10} e$
`M_LN2_BF`	Value of $\log_e 2$
`M_LN10_BF`	Value of $\log_e 10$
`M_PI_BF`	Value of π
`M_PI_2_BF`	Value of π / 2
`M_PI_4_BF`	Value of π / 4
`M_1_PI_BF`	Value of 1 / π
`M_2_PI_BF`	Value of 2 / π
`M_2_SQRTPI_BF`	Value of 2 / √π
`M_SQRT2_BF`	Value of √2
`M_SQRT1_2_BF`	Value of 1 / √2

6.7 Matrix Functions

The header `<metal_matrix>` defines the functions in Table 6.8. `T` is `float` or `half`.

Table 6.8. Matrix functions in the Metal standard library

Built-in matrix functions	Description
`float determinant(floatnxn)` `half determinant(halfnxn)`	Compute the determinant of the matrix. The matrix needs to be a square matrix.
`floatmxn transpose(floatnxm)` `halfmxn transpose(halfnxm)`	Transpose a matrix.

Example:

```
float4x4 mA;
float det = determinant(mA);
```

6.8 SIMD-Group Matrix Functions

The header `<metal_simdgroup_matrix>` defines the SIMD-group Matrix functions.

Instead of using simdgroup matrix multiplication, consider using Tensors (section 2.22) and Metal Performance Primitives (section 7). Together, they provide a performant and portable library supporting more general matrix and tensor operations across a wider variety of dimensions and extents.

6.8.1 Creating, Loading, and Storing Matrix Elements

Metal Shading Library supports the following functions to initialize a SIMD-group matrix with a value, load data from threadgroup or device memory, and store data to threadgroup or device memory.

Table 6.9. SIMD-Group matrix load and stores

Functions	Description
`simdgroup_matrix<T,Cols,Rows>(T dval)`	Creates a diagonal matrix with the given value.
`simdgroup_matrix<T,Cols,Rows> make_filled_simdgroup_matrix(T value)`	Initializes a SIMD-group matrix filled with the given value.
`void simdgroup_load( thread simdgroup_matrix<T,Cols,Rows>& d, const threadgroup T *src, ulong elements_per_row = Cols, ulong2 matrix_origin = 0, bool transpose_matrix = false)`	Loads data from threadgroup memory into a SIMD-group matrix. The `elements_per_row` parameter indicates the number of elements in the source memory layout.
`void simdgroup_load( thread simdgroup_matrix<T,Cols,Rows>& d, const device T *src, ulong elements_per_row = Cols, ulong2 matrix_origin = 0, bool transpose_matrix = false)`	Loads data from device memory into a SIMD-group matrix. The `elements_per_row` parameter indicates the number of elements in the source memory layout.

Functions	Description

`void simdgroup_store(`
`  thread simdgroup_matrix<T,Cols,Rows> a,`
`  threadgroup T *dst,`
`  ulong  elements_per_row = Cols,`
`  ulong2 matrix_origin = 0,`
`  bool   transpose_matrix = false)`
	Stores data from a SIMD-group matrix into threadgroup memory. The `elements_per_row` parameter indicates the number of elements in the destination memory layout.

`void simdgroup_store(`
`  thread simdgroup_matrix<T,Cols,Rows> a,`
`  device T *dst,`
`  ulong  elements_per_row = Cols,`
`  ulong2 matrix_origin = 0,`
`  bool   transpose_matrix = false)`
	Stores data from a SIMD-group matrix into device memory. The `elements_per_row` parameter indicates the number of elements in the destination memory layout.

6.8.2 Matrix Operations

SIMD-group matrices support multiply-accumulate and multiple operations.

Table 6.10. SIMD-Group operations

Operations	Description

`void simdgroup_multiply_accumulate(`
`  thread simdgroup_matrix<T,Cols,Rows>& d,`
`  thread simdgroup_matrix<T,K,Rows>&  a,`
`  thread simdgroup_matrix<T,Cols,K>&  b,`
`  thread simdgroup_matrix<T,Cols,Rows>& c)`
	Returns `d = a * b + c`

`void simdgroup_multiply(`
`  thread simdgroup_matrix<T,Cols,Rows>& d,`
`  thread simdgroup_matrix<T,K,Rows>&  a,`
`  thread simdgroup_matrix<T,Cols,K>&  b)`
	Returns `d = a * b`

`*`	Returns `a * b`

Here is an example of how to use SIMD-group matrices:

```
kernel void float_matmad(device float *pMatA, device float *pMatB
                         device float *pMatC, device float *pMatR)
{
    simdgroup_float8x8 sgMatA;
    simdgroup_float8x8 sgMatB;
    simdgroup_float8x8 sgMatC;
    simdgroup_float8x8 sgMatR;
```

```
    simdgroup_load(sgMatA, pMatA);
    simdgroup_load(sgMatB, pMatB);
    simdgroup_load(sgMatC, pMatC);

    simdgroup_multiply_accumulate(sgMatR, sgMatA, sgMatB, sgMatC);

    simdgroup_store(sgMatR, pMatR);
}
```

6.9 Geometric Functions

The functions in Table 6.11 are defined in the header `<metal_geometric>`. `T` is a vector floating-point type (`floatn` or `halfn`). `Ts` refers to the corresponding scalar type. (If `T` is `floatn` , the scalar type `Ts` is `float`. If `T` is `halfn` , `Ts` is `half`.)

Table 6.11. Geometric functions in the Metal standard library

Built-in geometric functions	Description
`T cross(T x, T y)`	Return the cross product of `x` and `y`. `T` needs to be a 3-component vector type.
`Ts distance(T x, T y)`	Return the distance between `x` and `y`, which is `length(x-y)`
`Ts distance_squared(T x, T y)`	Return the square of the distance between `x` and `y`.
`Ts dot(T x, T y)`	Return the dot product of `x` and `y`, which is `x[0] * y[0] + x[1] * y[1] + …`
`T faceforward(T N, T I, T Nref)`	If `dot(Nref, I) < 0.0` return `N`, otherwise return `-N`.
`Ts length(T x)`	Return the length of vector `x`, which is `sqrt(x[0]2 + x[1]2 + …)`
`Ts length_squared(T x)`	Return the square of the length of vector x, which is `(x[0]2 + x[1]2 + …)`
`T normalize(T x)`	Return a vector in the same direction as `x` but with a length of 1.
`T reflect(T I, T N)`	For the incident vector `I` and surface orientation `N`, compute normalized `N` (`NN`), and return the reflection direction: `I - 2 * dot(NN, I) * NN`.

Built-in geometric functions	Description
`T refract(T I, T N, Ts eta)`	For the incident vector `I` and surface normal `N`, and the ratio of indices of refraction `eta`, return the refraction vector. The input parameters for the incident vector `I` and the surface normal `N` needs to already be normalized to get the desired results.

For single precision floating-point, Metal also supports a precise and fast variant of the following geometric functions: `distance`, `length`, and `normalize`. To select the appropriate variant when compiling the Metal source, use the `ffast-math` compiler option (refer to section 1.6.3). In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide an explicit way to select the fast or precise variant of these geometric functions.

6.10 Synchronization and SIMD-Group Functions

You can use synchronization and SIMD-group functions in:

- `[[kernel]]` functions
- `[[fragment]]` functions
- `[[visible]]` functions that kernel or fragment functions call

6.10.1 Threadgroup and SIMD-Group Synchronization Functions

The `<metal_compute>` header defines the synchronization functions in Table 6.12, which lists threadgroup and SIMD-group synchronization functions it supports.

Table 6.12. Synchronization compute function in the Metal standard library

Built-in threadgroup function	Description
`void threadgroup_barrier(` `  mem_flags flags)`	All threads in a threadgroup executing the kernel, fragment, mesh, or object need to execute this function before any thread can continue execution beyond the `threadgroup_barrier`.

`void threadgroup_barrier(`
`  mem_flags flags,`
`  memory_order_order =`
`  memory_order_seq_cst,`
`  thread_scope scope =`
`  thread_scope_threadgroup)`
All OS: Metal 4.1 and later.	

Built-in threadgroup function	Description

`void simdgroup_barrier(`
`  mem_flags flags)`
macOS: Metal 2 and later
iOS: Metal 1.2 and later
iPadOS and visionOS: Always
	All threads in a SIMD-group executing the kernel, fragment, mesh, or object need to execute this function before any thread can continue execution beyond the `simdgroup_barrier`.

`void simdgroup_barrier(`
`  mem_flags flags,`
`  memory_order_order =`
`  memory_order_seq_cst,`
`  thread_scope scope =`
`  thread_scope_simdgroup)`
All OS: Metal 4.1 and later.	

A *barrier function* (`threadgroup_barrier` or `simdgroup_barrier`) acts as an execution and memory barrier. All threads in a threadgroup (or SIMD-group) executing the kernel must encounter the `threadgroup_barrier` (or `simdgroup_barrier`) function. The `threadgroup_barrier` (or `simdgroup_barrier`) also supports a variant that takes a thread scope parameter. When no thread scope is specified, `threadgroup_barrier` defaults to `thread_scope_threadgroup` and `simdgroup_barrier` defaults to `thread_scope_simdgroup`. On Apple silicon, a thread that has ended no longer participates or blocks remaining threads at a barrier.

If `threadgroup_barrier` (or `simdgroup_barrier`) is inside a conditional statement and if any thread enters the conditional statement and executes the barrier function, then all threads in the threadgroup (or SIMD-group) need to enter the conditional and execute the barrier function.

If `threadgroup_barrier` (or `simdgroup_barrier`) is inside a loop, for each iteration of the loop, if any thread in the threadgroup (or SIMD-group) executes the barrier, then all threads in the threadgroup (or SIMD-group) need to execute the barrier function before any threads continue execution beyond the barrier function.

The `threadgroup_barrier` (or `simdgroup_barrier`) function can also queue a memory fence (for reads and writes) to ensure the correct ordering of memory operations to threadgroup or device memory.

Table 6.13 describes the bit field values for the `mem_flags` argument to `threadgroup_barrier` and `simdgroup_barrier`. The `mem_flags` argument ensures the correct memory is in the correct order between threads in the threadgroup or SIMD-group (for `threadgroup_barrier` or `simdgroup_barrier`), respectively.

Table 6.13. Memory flag enumeration values for barrier functions

Memory flags (mem_flags)	Description
mem_none	The flag sets threadgroup_barrier or simdgroup_barrier to only act as an execution barrier and doesn't apply a Memory fence.
mem_device	The flag ensures the GPU correctly orders the memory operations to device memory for threads in the threadgroup or SIMD-group.
mem_threadgroup	The flag ensures the GPU correctly orders the memory operations to threadgroup memory for threads in a threadgroup or SIMD-group.
mem_texture macOS: Metal 1.2 and later. iOS: Metal 2 and later. iPadOS and visionOS: Always.	The flag ensures the GPU correctly orders the memory operations to texture memory for threads in a threadgroup or SIMD-group for a texture with the read_write access qualifier.
mem_threadgroup_imageblock	The flag ensures the GPU correctly orders the memory operations to threadgroup imageblock memory for threads in a threadgroup or SIMD-group.
mem_object_data	The flag ensures the GPU correctly orders the memory operations to object_data memory for threads in the threadgroup or SIMD-group.

The scope argument (see section 6.16.2) specifies which threads can observe the memory accesses to the address space identified by flags. The accesses become visible within the same threadgroup, within the same SIMD-group, or across all threads on the device.

6.10.2 SIMD-Group Functions

The <metal_simdgroup> header defines the SIMD-group functions for kernel and fragment functions. macOS supports SIMD-group functions in Metal 2 and later, and iOS supports most SIMD-group functions in Metal 2.2 and later. Table 6.14 and Table 6.15 list the SIMD-group functions and their availabilities in iOS and macOS. See the Metal Feature Set Tables to determine which GPUs support each table.

SIMD-group functions allow threads in a SIMD-group (see section 4.4.1) to share data without using threadgroup memory or requiring any synchronization operations, such as a barrier.

An *active* thread is a thread that is executing. An *inactive* thread is a thread that is *not* executing. For example, a thread may not be active due to flow control or when a task has insufficient work to fill the group. A thread needs to only read data from another active thread in the SIMD-group.

Helper threads may also be *active* and *inactive*. For example, if a helper thread finishes executing, it becomes an inactive helper thread. Helper threads for SIMD-group functions can

be active or inactive. Use `simd_is_helper_thread()` (see Table 6.14) to inspect whether a thread is a helper thread.

Table 6.14 uses the placeholder `T` to represent a scalar or vector of any integer or floating-point type, except:

- `bool`
- `bfloat`
- `long`
- `ulong`
- `void`
- `size_t`
- `ptrdiff_t`

For bitwise operations, `Ti` needs to be an integer scalar or vector.

See 6.10.2.1 after the table for examples that use SIMD-group functions.

Table 6.14. SIMD-Group permute functions in the Metal standard library

Built-in SIMD-group functions	Description
`simd_vote simd_active_threads_mask()` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns a `simd_vote` mask that represents the active threads. This function is equivalent to `simd_ballot(true)` and sets the bits that represent active threads to `1`, and inactive Threads to `0`.
`bool simd_all(bool expr)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `true` if all active threads evaluate `expr` to `true`.
`bool simd_any(bool expr)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `true` if at least one active thread evaluates `Expr` to `true`.
`simd_vote simd_ballot (bool expr)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns a wrapper type — see the `simd_vote` example — around a bitmask of the evaluation of the Boolean expression for all active threads in the SIMD-group for which `expr` is `true`. The function sets the bits that correspond to inactive threads to `0`.

Built-in SIMD-group functions	Description
`T simd_broadcast(T data, ushort broadcast_lane_id)` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Broadcasts `data` from the thread whose SIMD lane ID is equal to `broadcast_lane_id`. The specification doesn't define the behavior when `broadcast_lane_id` isn't a valid SIMD lane ID or isn't the same for all threads in a SIMD-group.
`T simd_broadcast_first(T data)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Broadcasts `data` from the first active thread — the active thread with the smallest index — in the SIMD-group to all active threads.
`bool simd_is_first()` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `true` if the current thread is the first active thread — the active thread with the smallest index — in the current SIMD-group; otherwise, `false`.
`T simd_shuffle(T data, ushort simd_lane_id)` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is `simd_lane_id`. The `simd_lane_id` needs to be a valid SIMD lane ID but doesn't have to be the same for all threads in the SIMD-group.
`T simd_shuffle_and_fill_down(T data, T filling_data, ushort delta)` All OS: Metal 2.4 and later.	Returns `data` or `filling_data` from the thread whose SIMD lane ID is the sum of the caller's SIMD lane ID and `delta`. If the sum is greater than the SIMD-group size, the function copies values from the lower `delta` lanes of `filling_data` into the upper `delta` lanes of `data`. The value for `delta` needs to be the same for all threads in a SIMD-group.
`T simd_shuffle_and_fill_down(T data, T filling_data, ushort delta, ushort modulo)` All OS: Metal 2.4 and later.	Returns `data` or `filling_data` for each vector from the thread whose SIMD lane ID is the sum of the caller's SIMD lane ID and `delta`. If the sum is greater than `modulo`, the function copies values from the lower `delta` lanes of `filling_data` into the upper `delta` lanes of `data`. The value of `delta` needs to be the same for all threads in a SIMD-group.

Built-in SIMD-group functions	Description
	The `modulo` parameter defines the vector width that splits the SIMD-group into separate vectors and must be 2, 4, 8, 16, or 32.
`T simd_shuffle_and_fill_up(T data, T filling_data, ushort delta)` All OS: Metal 2.4 and later.	Returns `data` or `filling_data` from the thread whose SIMD lane ID is the difference from the caller's SIMD lane ID minus `delta`. If the difference is negative, the operation copies values from the upper `delta` lanes of `filling_data` to the lower `delta` lanes of `data`. The value of `delta` needs to be the same for all threads in a SIMD-group.
`T simd_shuffle_and_fill_up(T data, T filling_data, ushort delta, ushort modulo)` All OS: Metal 2.4 and later.	Returns `data` or `filling_data` for each vector from the thread whose SIMD lane ID is the difference from the caller's SIMD lane ID minus `delta`. If the difference is negative, the operation copies values from the upper `delta` lanes of `filling_data` to the lower `delta` lanes of `data`. The value of `delta` needs to be the same for all threads in a SIMD-group. The `modulo` parameter defines the vector width that splits the SIMD-group into separate vectors and must be 2, 4, 8, 16, or 32.
`T simd_shuffle_down(T data, ushort delta)` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is the sum of caller's SIMD lane ID and `delta`. The value for `delta` needs to be the same for all threads in the SIMD-group. This function doesn't modify the upper `delta` lanes of `data` because it doesn't wrap values around the SIMD-group.
`T simd_shuffle_rotate_down(T data, ushort delta)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is the sum of caller's SIMD lane ID and `delta`. The value for `delta` needs to be the same for all threads in the SIMD-group. This function wraps values around the SIMD-group.

Built-in SIMD-group functions	Description
`T simd_shuffle_rotate_up(T data, ushort delta)` macOS: Metal 2.1 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is the difference from the caller's SIMD lane ID minus `delta`. The value of `delta` needs to be the same for all threads in a SIMD-group. This function wraps values around the SIMD-group.
`T simd_shuffle_up(T data, ushort delta)` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is the difference from the caller's SIMD lane ID minus `delta`. The value of `delta` needs to be the same for all threads in a SIMD-group. This function doesn't modify the lower `delta` lanes of `data` because it doesn't wrap values around the SIMD-group.
`Ti simd_shuffle_xor(Ti value, ushort mask)` macOS: Metal 2 and later. iOS: Metal 2.2 and later. iPadOS and visionOS: Always.	Returns `data` from the thread whose SIMD lane ID is equal to the bitwise `XOR` (`^`) of the caller's SIMD lane ID and `mask`. The value of `mask` needs to be the same for all threads in a SIMD-group.

Table 6.15. SIMD-Group reduction functions in the Metal standard library

Built-in SIMD-group functions	Description
`Ti simd_and(Ti data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always	Returns the bitwise `AND` (`&`) of `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.
`bool simd_is_helper_thread()` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns `true` if the current thread is a helper thread; otherwise, `false`. You call this function from a fragment function or another function that your fragment function calls; otherwise, it may trigger a compile-time error.
`T simd_max(T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns `data` with the highest value from across all active threads in the SIMD-group and broadcasts that value to all active threads in the SIMD-group.

Built-in SIMD-group functions	Description
`T simd_min(T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns `data` with the lowest value from across all active threads in the SIMD-group and broadcasts that value to all active threads in the SIMD-group.
`Ti simd_or(Ti data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns the bitwise `OR` (`|`) of `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.
`T simd_prefix_exclusive_product (T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	For a given thread, returns the product of the input values in `data` for all active threads with a lower index in the SIMD-group. The first thread in the group, returns `T(1)`.
`T simd_prefix_exclusive_sum (T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	For a given thread, returns the sum of the input values in `data` for all active threads with a lower index in the SIMD-group. The first thread in the group, returns `T(0)`.
`T simd_prefix_inclusive_product (T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	For a given thread, returns the product of the input values in `data` for all active threads with a lower or the same index in the SIMD-group.
`T simd_prefix_inclusive_sum (T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	For a given thread, returns the sum of the input values in `data` for all active threads with a lower or the same index in the SIMD-group.
`T simd_product(T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns the product of the input values in `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.
`T simd_sum(T data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later.	Returns the sum of the input values in `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.

Built-in SIMD-group functions	Description
visionOS: Always.	
`Ti simd_xor(Ti data)` macOS: Metal 2.1 and later. iOS and iPadOS: Metal 2.3 and later. visionOS: Always.	Returns the bitwise `XOR` (`^`) of `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.

6.10.2.1 Examples

To demonstrate the shuffle functions, start with this SIMD-group's initial state:

SIMD Lane ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	a	b	c	d	e	f	g	h	i	j	K	l	m	n	o	p

The `simd_shuffle_up()` function shifts each SIMD-group upward by `delta` threads. For example, with a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs down by two
- Marks the lower two lanes as invalid

Computed SIMD lane ID	–2	–1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
valid	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
data	a	b	a	b	c	d	e	f	g	h	i	j	k	l	m	n

The `simd_shuffle_up()` function is a no-wrapping operation that doesn't affect the lower `delta` lanes.

Similarly, the `simd_shuffle_down()` function shifts each SIMD-group downward by the `delta` threads. Starting with the same initial SIMD-group state, with a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs up by two
- Marks the upper two lanes as invalid

Computed SIMD lane ID	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
valid	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
data	c	d	e	f	g	h	i	j	k	l	m	n	o	p	o	p

The `simd_shuffle_down()` function is a no-wrapping operation that doesn't affect the upper `delta` lanes.

To demonstrate the shuffle-and-fill functions, start this SIMD-group's initial state:

SIMD lane ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	a	b	c	d	e	f	g	h	s	t	u	v	w	x	y	z
filling	fa	fb	fc	fd	fe	ff	fg	fh	fs	ft	fu	fv	fw	fx	fy	fz

The `simd_shuffle_and_fill_up()` function shifts each SIMD-group upward by `delta` threads — similar to `simd_shuffle_up()` — and assigns the values from the upper `filling` lanes to the lower `data` lanes by wrapping the SIMD lane IDs. For example, with a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs down by two
- Assigns the upper two lanes of `filling` to the lower two lanes of `data`

Computed SIMD lane ID	–2	–1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
data	fy	fz	a	b	c	d	e	f	g	h	s	t	u	v	w	x

The `simd_shuffle_and_fill_up()` function with the `modulo` parameter splits the SIMD-group into vectors, each with size `modulo`, and shifts each vector by the `delta` threads. For example, with a `modulo` value of `8` and a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs down by two
- Assigns the upper two lanes of each vector in `filling` to the lower two lanes of each vector in `data`

Computed SIMD lane ID	–2	–1	0	1	2	3	4	5	–2	–1	0	1	2	3	4	5
data	fg	fh	a	b	c	d	e	f	fy	fz	s	t	u	v	w	x

The `simd_shuffle_and_fill_down()` function shifts each SIMD-group downward by `delta` threads — like `simd_shuffle_down()` — and assigns the values from the lower `filling` lanes to the upper `data` lanes by wrapping the SIMD lane IDs. For example, with a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs up by two
- Assigns the lower two lanes of `filling` to the upper two lanes of `data`

Computed SIMD lane ID	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
data	c	d	e	f	g	h	s	t	u	v	w	x	y	z	fa	fb

The `simd_shuffle_and_fill_down()` function with the `modulo` parameter splits the SIMD-group into vectors, each with size `modulo` and shifts each vector by the `delta` threads. For example, with a `modulo` value of `8` and a `delta` value of `2`, the function:

- Shifts the SIMD lane IDs up by two
- Assigns the lower two lanes of each vector in `filling` to the upper two lanes of each vector in `data`

Computed SIMD lane ID	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
data	c	d	e	f	g	h	fa	fb	u	v	w	x	y	z	fs	ft

Below is an example of how to use these SIMD functions to perform a reduction operation:

```
kernel void
reduce(const device int *input [[buffer(0)]],
       device atomic_int *output [[buffer(1)]],
       threadgroup int *ldata [[threadgroup(0)]],
       uint gid [[thread_position_in_grid]],
       uint lid [[thread_position_in_threadgroup]],
       uint lsize [[threads_per_threadgroup]],
       uint simd_size [[threads_per_simdgroup]],
       uint simd_lane_id [[thread_index_in_simdgroup]],
       uint simd_group_id [[simdgroup_index_in_threadgroup]])
{
     // Perform the first level of reduction.
     // Read from device memory, write to threadgroup memory.
     int val = input[gid] + input[gid + lsize];
     for (uint s=lsize/simd_size; s>simd_size; s/=simd_size)
     {
          // Perform per-SIMD partial reduction.
          for (uint offset=simd_size/2; offset>0; offset/=2)
               val += simd_shuffle_down(val, offset);
          // Write per-SIMD partial reduction value to
          // threadgroup memory.
          if (simd_lane_id == 0)
               ldata[simd_group_id] = val;
          // Wait for all partial reductions to complete.
          threadgroup_barrier(mem_flags::mem_threadgroup);

          val = (lid < s) ? ldata[lid] : 0;
     }
     // Perform final per-SIMD partial reduction to calculate
     // the threadgroup partial reduction result.
     for (uint offset=simd_size/2; offset>0; offset/=2)
          val += simd_shuffle_down(val, offset);
     // Atomically update the reduction result.
     if (lid == 0)
          atomic_fetch_add_explicit(output, val,
                                    memory_order_relaxed);
}
```

The `simd_active_threads_mask` and `simd_ballot` function uses the `simd_vote` wrapper type (see below), which can be explicitly cast to its underlying type represented by `vote_t`.

```
class simd_vote {
public:
     explicit constexpr simd_vote(vote_t v = 0);
     explicit constexpr operator vote_t() const;

     // Returns true if all bits corresponding to threads in the
     // SIMD-group are set.
     // You can use all() with the return value of simd_ballot(expr)
     // to determine if all threads are active.
     bool all() const;

     // Returns true if any bit corresponding to a valid thread in
     // the SIMD-group is set.
     // You can use any() with the return value of simd_ballot(expr)
     // to determine if at least one thread is active.
     bool any() const;

      private:
     // bit i in v represents the 'vote' for the thread in the
     // SIMD-group at index i
     uint64_t v;
};
```

Note that `simd_all(expr)` is different from `simd_ballot(expr).all()`:

- `simd_all(expr)` returns `true` if all *active* threads evaluate `expr` to `true`.
- `simd_ballot(expr).all()` returns `true` if all threads *were* active and evaluated the `expr` to `true`. (`simd_vote::all()` does not look at which threads are active.)

The same logic applies to `simd_any`, `simd_vote::any()`, and to the equivalent quad functions listed in section 6.10.3.

On hardware with fewer than 64 threads in a SIMD-group, the value of the top bits in `simd_vote` is undefined. Because you can initialize these bits, do not assume that the top bits are set to `0`.

6.10.3 Quad-Group Functions

macOS: Metal 2.1 and later support quad-group functions.
iOS: Metal 2 and later support some quad-group functions, including `quad_broadcast`, `quad_shuffle`, `quad_shuffle_up`, `quad_shuffle_down`, and `quad_shuffle_xor`.
iPadOS and visionOS: Metal supports quad-group functions.

A quad-group function is a SIMD-group function (see section 6.10.2) with an execution width of 4. The *active* and *inactive* thread terminology is the same as in section 6.10.2.

Helper threads only execute to compute gradients for quad-groups in a fragment shader and then become inactive.

Kernels and fragment functions can call the quad-group functions listed in Table 6.17 and Table 6.18. Threads may only read data from another active thread in a quad-group. See the Metal Feature Set Tables to determine which GPUs support each table.

The placeholder `T` for Table 6.17 and Table 6.18 represents a scalar or vector of any integer or floating-point type, except:

- `bool`
- `void`
- `size_t`
- `ptrdiff_t`

For bitwise operations, `T` needs to be an integer scalar or vector.

Table 6.16. Quad-group function in the Metal standard library

Built-in quad-group functions	Description
`quad_vote quad_ballot (bool expr)` macOS: Metal 2.1 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always	Returns a `quad_vote` bitmask where each bit indicates where the Boolean expression `expr` evaluates to `true` for active threads in the quad-group. The function sets the bits that correspond to inactive threads to `0`. See an example at the end of this section.

Table 6.17. Quad-group permute functions in the Metal standard library

Built-in quad-group functions	Description
`T quad_broadcast(T data, ushort broadcast_lane_id)` macOS: Metal 2 and later iOS: Metal 2 and later iPadOS and visionOS: Always	Broadcasts `data` from the thread whose quad lane ID is `broadcast_lane_id`. The value for `broadcast_lane_id` needs to be a valid quad lane ID that's the same for all threads in a quad-group.
`T quad_broadcast_first(T data)` macOS: Metal 2.1 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always	Broadcasts `data` from the first active thread — the active thread with the smallest index — in the quad-group to all active threads.

Built-in quad-group functions	Description
`T quad_shuffle(T data, ushort quad_lane_id)` macOS: Metal 2 and later iOS: Metal 2 and later iPadOS and visionOS: Always	Returns `data` from the thread whose quad lane ID is the sum of the caller's quad lane ID and `delta`. The value for `quad_lane_id` needs to be a valid land ID and may differ from other threads in the quad-group.
`T quad_shuffle_and_fill_down(T data, T filling_data, ushort delta)` All OS: Metal 2.4 and later	Returns `data` or `filling_data` from the thread whose quad lane ID is the sum of the caller's quad lane ID and `delta`. If the sum is greater than the quad-group size, the function copies values from the lower `delta` lanes of `filling_data` into the upper `delta` lanes of `data`. The value for `delta` needs to be the same for all threads in a quad-group.
`T quad_shuffle_and_fill_down(T data, T filling_data, ushort delta, ushort modulo)` All OS: Metal 2.4 and later	Returns `data` or `filling_data` for each vector, from the thread whose quad lane ID is the sum of caller's quad lane ID and `delta`. If the sum is greater than the quad-group size, the function copies values from the lower `delta` lanes of `filling_data` into the upper `delta` lanes of `data`. The value of `delta` needs to be the same for all threads in a quad-group. The `modulo` parameter defines the vector width that splits the quad-group into separate vectors and must be 2 or 4.
`T quad_shuffle_and_fill_up(T data, T filling_data, ushort delta)` All OS: Metal 2.4 and later	Returns `data` or `filling_data` from the thread whose quad lane ID is the difference from the caller's quad lane ID minus `delta`. If the difference is negative, the operation copies values from the upper `delta` lanes of `filling_data` to the lower `delta` lanes of `data`. If the difference is negative, the function shuffles data from `filling_data` into the lower `delta` lanes. The value of `delta` needs to be the same for all threads in a quad-group.

Built-in quad-group functions	Description
`T quad_shuffle_and_fill_up(T data, T filling_data, ushort delta, ushort modulo)` All OS: Metal 2.4 and later	Returns `data` or `filling_data` for each vector from the thread whose quad lane ID is the difference from the caller's quad lane ID minus `delta`. If the difference is negative, the operation copies values from the upper `delta` lanes of `filling_data` to the lower `delta` lanes of `data`. The value of `delta` needs to be the same for all threads in a quad-group. The `modulo` parameter defines the width that splits the quad-group into separate vectors and must be 2 or 4.
`T quad_shuffle_down(T data, ushort delta)` macOS: Metal 2 and later iOS: Metal 2 and later iPadOS and visionOS: Always	Returns `data` from the thread whose quad lane ID is the sum of the caller's quad lane ID and `delta`. The value for `delta` needs to be the same for all threads in a quad-group. The function doesn't modify the upper `delta` lanes of `data` because it doesn't wrap values around the quad-group.
`T quad_shuffle_rotate_down(T data, ushort delta)` macOS: Metal 2.1 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always	Returns `data` from the thread whose quad lane ID is the sum of the caller's quad lane ID and `delta`. The value for `delta` needs to be the same for all threads in a quad-group. This function wraps values around the quad-group.
`T quad_shuffle_rotate_up(T data, ushort delta)` macOS: Metal 2.1 and later iOS: Metal 2.2 and later iPadOS and visionOS: Always	Returns `data` from the thread whose quad lane ID is the difference from the caller's quad lane ID minus `delta`. The value for `delta` needs to be the same for all threads in a quad-group. This function wraps values around the quad-group.
`T quad_shuffle_up(T data, ushort delta)` macOS: Metal 2 and later iOS: Metal 2 and later iPadOS and visionOS: Always	Returns `data` from thread whose quad lane ID is the difference from the caller's quad lane ID minus `delta`. The value for `delta` needs to be the same for all threads in a quad-group. This function doesn't modify the lower `delta` lanes of `data` because it doesn't wrap values around the quad-group.

Built-in quad-group functions	Description
`T quad_shuffle_xor(T value, ushort mask)` macOS: Metal 2 and later iOS: Metal 2 and later iPadOS and visionOS: Always	Returns `data` from the thread whose quad lane ID is a bitwise `XOR` (`^`) of the caller's quad lane ID and `mask`. The value of `mask` needs to be the same for all threads in a quad-group.

Table 6.18. Quad-group reduction functions in the Metal standard library

Built-in quad-group functions	Description
`quad_vote quad_active_threads_mask()` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns a `quad_vote` mask that represents the active threads. The function is equivalent to `quad_ballot(true)` and sets the bits that represent active threads to `1` and inactive threads to `0`.
`bool quad_all(bool expr)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `true` if all active threads evaluate `expr` to `true`.
`T quad_and(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns the bitwise `AND` (`&`) of `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.
`bool quad_any(bool expr)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `true` if at least one active thread evaluates `expr` to `true`.
`bool quad_is_first()` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `true` if the current thread is the first active thread — the active thread with the smallest index — in the current quad-group; otherwise, `false`.
`bool quad_is_helper_thread()` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `true` if the current thread is a helper thread; otherwise, `false`. You call this function from a fragment function or another function that your fragment function calls; otherwise, it may trigger a compile-time error.

Built-in quad-group functions	Description
`T quad_max(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `data` with the highest value from across all active threads in the quad-group and broadcasts that value to all active threads in the quad-group.
`T quad_min(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns `data` with the lowest value from across all active threads in the quad-group and broadcasts that value to all active threads in the quad-group.
`T quad_or(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns the bitwise `OR` (`|`) of `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.
`T quad_prefix_exclusive_product (T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	For a given thread, returns the product of the input values in `data` for all active threads with a lower index in the quad-group. For the first thread in the group, return `T(1)`.
`T quad_prefix_exclusive_sum (T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	For a given thread, returns the sum of the input values in `data` for all active threads with a lower index in the quad-group. For the first thread in the group, return `T(0)`.
`T quad_prefix_inclusive_product (T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	For a given thread, returns the product of the input values in `data` for all active threads with a lower or the same index in the quad-group.
`T quad_prefix_inclusive_sum (T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	For a given thread, returns the sum of the input values in `data` for all active threads with a lower or the same index in the quad-group.
`T quad_product(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns the product of the input values in `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.

Built-in quad-group functions	Description
`T quad_sum(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns the sum of the input values in `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.
`T quad_xor(T data)` macOS: Metal 2.1 and later iOS and iPadOS: Metal 2.3 and later visionOS: Always	Returns the bitwise `XOR` (`^`) of `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.

In a kernel function, quads divide across the SIMD-group. In a fragment function, the lane ID represents the fragment location in a 2 x 2 quad:

- Lane ID 0 is the upper-left pixel
- Lane ID 1 is the upper-right pixel
- Lane ID 2 is the lower-left pixel
- Lane ID 3 is the lower-right pixel

To demonstrate the shuffle functions, start with this quad-group's initial state:

Quad lane ID	0	1	2	3
data	a	b	c	d

The `quad_shuffle_up()` function shifts each quad-group upward by `delta` threads. For example, with a `delta` value of `2`, the function:

- Shifts the quad lane IDs down by two
- Marks the lower two lanes as invalid

Computed quad lane ID	–2	–1	0	1
valid	0	0	1	1
data	a	b	a	b

The `quad_shuffle_up()` function is a no wrapping operation that doesn't affect the lower `delta` lanes.

Similarly, `quad_shuffle_down()` function shifts each quad-group downward by `delta` threads. Starting with the same initial quad-group state, with a `delta` of `2`, the function:

- Shifts the quad lane IDs up by two
- Marks the upper two lanes as invalid

Computed quad lane ID	2	3	4	5
valid	1	1	0	0
data	c	d	c	d

The `quad_shuffle_down()` function is a no wrapping operation that doesn't affect the upper `delta` lanes.

To demonstrate the shuffle-and-fill functions, start this quad-group's initial state:

Quad lane ID	0	1	2	3
data	a	b	c	d
filling	fa	fb	fc	fd

The `quad_shuffle_and_fill_up()` function shifts each quad-group upward by the `delta` threads — similar to `quad_shuffle_up()` — and assigns the values from the upper `filling` lanes to the lower `data` lanes by wrapping the quad lane IDs. For example, with a `delta` value of `2`, the function:

- Shifts the quad lane IDs down by two
- Assigns the upper two lanes of `filling` to the lower two lanes of `data`

Computed quad lane ID	–2	–1	0	1
data	fc	fd	a	b

The `quad_shuffle_and_fill_up()` function with the `modulo` parameter splits the quad-group into vectors, each with size `modulo` and shifts each vector by the `delta` threads. For example, with a `modulo` value of `2` and a `delta` value of `1`, the function:

- Shifts the quad lane IDs down by one
- Assigns the upper lane of each vector in `filling` to the lower lane of each vector in `data`

Computed quad lane ID	–1	0	–1	0
data	fb	a	fd	c

The `quad_shuffle_and_fill_down()` function shifts each quad-group downward by `delta` threads — similar to `quad_shuffle_down()` — and assigns the values from the lower `filling` lanes to the upper `data` lanes by wrapping the quad lane IDs. For example, with a `delta` value of `2`, the function:

- Shifts the quad lane IDs up by two
- Assigns the lower two lanes of `filling` to the upper two lanes of `data`

Computed quad lane ID	2	3	4	5
data	c	d	fa	fb

The `quad_shuffle_and_fill_down()` function with the `modulo` parameter splits the quad-group into vectors, each with size `modulo` and shifts each vector by the `delta` threads. For example, with a `modulo` value of `2` and a `delta` value of `1`, the function:

- Shifts the quad lane IDs up by one
- Assigns the lower lane of each vector in `filling` to the upper lane of each vector in `data`

Computed quad lane ID	1	2	1	2
data	b	fa	d	fc

The `quad_ballot` function uses the `quad_vote` wrapper type, which can be explicitly cast to its underlying type. (In the following example, note use of `vote_t` to represent an underlying type, `XXX`.)

```
class quad_vote {
public:
     typedef XXX vote_t;
     explicit constexpr quad_vote(vote_t v = 0);
     explicit constexpr operator vote_t() const;

     // Returns true if all bits corresponding to threads in the
     // quad-group (the four bottom bits) are set.
     bool all() const;

     // Returns true if any bit corresponding to a thread in the
     // quad-Group is set.
     bool any() const;
};
```

The `quad_vote` constructor masks out the top bits (that is, other than the four bottom bits). Therefore, Metal clears the upper bits, and the bottom four bits don't change when you cast to `vote_t`.

6.11 Graphics Functions

The graphics functions in this section and its subsections are defined in the header `<metal_graphics>`. You can only call these graphics functions from a `fragment` function.

6.11.1 Fragment Functions

You can only call the functions in this section (listed in Table 6.19, Table 6.20, and Table 6.21) inside a fragment function (see section 5.1.2) or inside a function called from a fragment function. Otherwise, the behavior is undefined and may result in a compile-time error.

Fragment function helper threads may be created to help evaluate derivatives (explicit or implicit) for use with a fragment thread(s). Fragment function helper threads execute the same code as the other fragment threads, but do not have side effects that modify the render targets or any other memory that can be accessed by the fragment function. In particular:

- Fragments corresponding to helper threads are discarded when the fragment function execution is complete without any updates to the render targets.
- Stores and atomic operations to buffers and textures performed by helper threads have no effect on the underlying memory associated with the buffer or texture.

6.11.1.1 Fragment Functions – Derivatives

Metal includes the functions in Table 6.19 to compute derivatives. `T` is one of `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3`, or `half4`.

Derivatives are undefined within nonuniform control flow.

Note: In Metal 2.2 and earlier, `discard_fragment` could make the control flow nonuniform. In Metal 2.3 and later, `discard_fragment` does not affect whether the control flow is considered nonuniform or not. See Section 6.11.1.3 for more information.

Table 6.19. Derivatives fragment functions in the Metal standard library

Built-in fragment functions	Description
`T dfdx(T p)`	Returns a high precision partial derivative of the specified value with respect to the screen space `x` coordinate.
`T dfdy(T p)`	Returns a high precision partial derivative of the specified value with respect to the screen space `y` coordinate.
`T fwidth(T p)`	Returns the sum of the absolute derivatives in `x` and `y` using local differencing for `p`; that is, `fabs(dfdx(p)) + fabs(dfdy(p))`

6.11.1.2 Fragment Functions — Samples

Metal includes the per-sample functions listed in Table 6.20. `get_num_samples` and `get_sample_position` return the number of samples for the color attachment and the sample offsets for a given sample index. For example, for transparency super-sampling, these functions can be used to shade per-fragment but do the alpha test per-sample.

Table 6.20. Samples fragment functions in the Metal standard library

Built-in fragment functions	Description
`uint get_num_samples()`	Returns the number of samples for the multisampled color attachment.
`float2 get_sample_position(uint index)`	Returns the normalized sample offset (`x`, `y`) for a given sample index `index`. Values of `x` and `y` are in `[0.0 … 1.0]`.

If you have customized sample positions (set with the `setSamplePositions:count:` method of `MTLRenderPassDescriptor`), `get_sample_position(index)` returns the position programmed for the specified index.

6.11.1.3 Fragment Functions — Flow Control

The Metal function in Table 6.21 terminates a fragment.

Table 6.21. Fragment flow control function in the Metal standard library

Built-in fragment functions	Description
`void discard_fragment(void)`	Marks the current fragment as terminated and discards this fragment's output of the fragment function.

Writes to a buffer or texture from a fragment thread made *before* calling `discard_fragment` are not discarded.

Multiple fragment threads or helper threads associated with a fragment thread execute together to compute derivatives. In Metal 2.2 and earlier, if any (but not all) of these threads executes the `discard_fragment` function, the thread is terminated and the behavior of any derivative computations (explicit or implicit) is undefined. In Metal 2.3 and later, `discard_fragment` marks the fragment as terminated while continuing to execute in parallel and has no effect on whether derivatives are defined. Even though execution continues, the write behavior remains the same as before. The fragment will discard the fragment output and discard all writes to buffer or texture after `discard_fragment`.

6.12 Pull-Model Interpolation

All OS: Metal 2.3 and later support pull-model interpolation.

The interpolant type `interpolant<T,P>` (section 2.18) and associated methods are defined in `<metal_interpolate>`. In a fragment function, you explicitly interpolate the values of a `interpolant<T,P>` type by invoking its methods, as shown below. The interpolant may be

sampled and interpolated multiple times, in different modes, and may be passed to other functions to be sampled and interpolated there. Perspective correctness is fixed across all interpolations of the argument by the value of `P` in its type.

Table 6.22. Pull-Model interpolant methods

Interpolant method	Description
`T interpolate_at_center()`	Sample shader input at the center of a pixel, returning the same value as if the input had type `T` with `[[center_perspective]]` or `[[center_no_perspective]]`.
`T interpolate_at_centroid()`	Sample shader input within the covered area of the pixel, returning the same value as if the input had type `T` with `[[centroid_perspective]]` or `[[centroid_no_perspective]]`.
`T interpolate_at_offset(float2 offset)`	Sample shader input at a specified window-coordinate offset from a pixel's top-left corner. Allowable offset components are in the range [0.0, 1.0) along a 1/16 pixel grid.
`T interpolate_at_sample(uint sample)`	Sample shader input at the location of the specified sample index, returning the same value as if the input had type `T` with `[[sample_perspective]]` or `[[sample_no_perspective]]` and was in the specified per-sample evaluation of the shader. If a sample of the given index does not exist, the position of interpolation is undefined.

6.13 Texture Functions

The texture member functions, defined in the header `<metal_texture>`, listed in this section and its subsections for different texture types include:

- `sample` — Sample from a texture.
- `sample_compare` — Sample compare from a texture.
- `gather` — Gather from a texture.
- `gather_compare` — Gather compare from a texture.
- `read` — Sampler-less read from a texture; in Metal 4.1 and later, also supports a clamp-to-edge read.
- `write` — Write to a texture.

- `get_width` — Return the width of a texture.
- `get_height` — Return the height of a texture.
- `get_num_mip_levels` — Return the number of mip levels of a texture.
- `get_array_size` — Return the size of the array if the texture is a texture array.
- `fence` — Make a write visible to a read on the same texture.

In Metal 3.1 and later, new atomic texture member functions are supported on 1D texture, 1D texture array, 2D texture, 2D texture array, 3D texture, and texture buffer for `int`, `uint`, and `ulong` color type:

- `atomic_load` — Atomic load from a texture for `int` and `uint` color type.
- `atomic_store` — Atomic store to a texture for `int` and `uint` color type.
- `atomic_exchange` — Atomic exchange a value in a texture for `int` and `uint` color type.
- `atomic_compare_exchange_weak` — Atomic compare and exchange in a texture for `int` and `uint` color type.
- `atomic_fetch_`*op*`_explicit` — Atomic fetch and modify in a texture for `int` and `uint` color type where *op* can be `add`, `and`, `max`, `min`, `or`, `sub`, or `xor`.
- `atomic_max` — Atomic max in a texture for `ulong` color type.
- `atomic_min` — Atomic min in a texture for `ulong` color type.

Metal 4 adds support for the atomic texture functions for cube texture and cube texture array.

See the Metal Feature Set Tables to determine which GPUs support texture atomics.

Metal 4.1 adds support for clamp-to-edge reads, integer-coordinate reads with offsets, and multipixel reads.

Metal 3.2 introduces coherence (see section 2.9).

The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions take an `offset` argument for a 2D texture, 2D texture array, and 3D texture. The `offset` is an integer value applied to the texture coordinate before looking up each pixel. This integer value must be in the range `-8` to `+7`; the default value is `0`.

The texture `sample`, `sample_compare`, `gather`, and `gather_compare` functions require that you declare the texture with the `sample` access attribute. The texture `read` functions require that you declare the texture with the `sample`, `read`, or `read_write` access attribute. The texture `write` functions require that you declare the texture with the `write` or `read_write` access attribute. (For more about access attributes, see section 2.9.)

The texture `sample_compare` and `gather_compare` functions are only available for depth texture types.

`compare_func` sets the comparison test for the `sample_compare` and `gather_compare` functions. For more about `compare_func`, see section 2.10.

Overloaded variants of the texture `sample` and `sample_compare` functions with an `lod_options` argument are available for a 2D texture, 2D texture array, 2D depth texture, 2D depth texture array, 3D texture, cube texture, cube texture array, cube depth texture, and cube depth texture array. (LOD/lod is short for level-of-detail.) The values for `lod_options` are:

- `level(float lod)` — Sample from the specified mipmap level.

- `bias(float value)` — Apply the specified bias to a mipmap level before sampling.
- `gradient*(T dPdx, T dPdy)` — Apply the specified gradients with respect to the x and y directions. The texture type changes the name and the arguments; for example, for 3D textures, the name is `gradient3d` and the arguments are `float3` type.
- `min_lod_clamp(float lod)` — Specify lowest mipmap level for sampler access, which restricts sampler access to a range of mipmap levels. (All OS: Support since Metal 2.2.)

In macOS, Metal 2.2 and earlier don't support `sample_compare`, `bias` and `gradient*` functions, and `lod` needs to be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

In Metal 2.2 and later, you can specify a LOD range for a sampler. You can either specify a minimum and maximum mipmap level or use `min_lod_clamp` to specify just the minimum mipmap level of an open range. When the sampler determines which mipmaps to sample, the selection is clamped to the specified range.

Clamping the LOD is useful where some of the texture data is not available all the time (for example, texture streaming). You can create a texture with all the necessary mipmaps and then can stream image data starting from the smallest mipmaps. When the GPU samples the texture, it clamps to the mipmaps that already have valid data. When you copy larger mipmaps into the texture, you reduce the minimum LOD level. As new data becomes ready, you can change the LOD clamp, which changes the sampling resolution.

The texture `sample` and `sample_compare` functions that don't take an explicit LOD or gradients when you don't call them in a fragment function, have a default LOD of `0`. In a fragment function, the texture `sample` and `sample_compare` functions that don't take an explicit LOD or gradients calculate an implicit LOD by taking the derivative of the texture coordinate passed to the function. The `gather` and `gather_compare` functions you don't call in a fragment function also have a default LOD of `0`.

For the `gather` and `gather_compare` functions, place the four samples that contribute to filtering into `xyzw` components in counter-clockwise order, starting with the sample to the lower-left of the queried location. This is the same as nearest sampling with unnormalized texture coordinate deltas at the following locations: `(-,+)`, `(+,+)`, `(+,-)`, `(-,-)`, where the magnitude of the deltas is always half a pixel.

A `read` from or `write` to a texture is out-of-bounds if and only if any of these conditions is met:

- the coordinates accessed are out-of-bounds
- the level of detail argument is out-of-bounds
- the texture is a texture array (`texture?d_array` type), and the `array` slice argument is out-of-bounds
- the texture is a `texturecube` or `texturecube_array` type, and the `face` argument is out-of-bounds
- the texture is a multisampled texture, and the `sample` argument is out-of-bounds

For all texture types, an out-of-bounds `write` to a texture is ignored.

For all texture types:

- For components specified in a pixel format, an out-of-bounds `read` returns a color with components with the value zero.

- For components unspecified in a pixel format, an out-of-bounds `read` returns the default value.

For unspecified color components in a pixel format, the default values are:

- `0`, for components other than alpha.
- `1`, for the alpha component.

In a pixel format with integer components, the alpha default value is represented as the integral value `0x1`. For a pixel format with floating-point or normalized components, the alpha default value is represented as the floating-point value `1.0`.

For example, for a texture with the `MTLPixelFormatR8Uint` pixel format, the default values for unspecified integer components are G = `0`, B = `0`, and A = `1`. For a texture with the `MTLPixelFormatR8Unorm` pixel format, the default values for unspecified normalized components are G = `0.0`, B = `0.0`, and A = `1.0`. For a texture with depth or stencil pixel format (such as `MTLPixelFormatDepth24Unorm_Stencil8` or `MTLPixelFormatStencil8`), the default value for an unspecified component is undefined.

In macOS, for Metal 2.2 and earlier, `lod` needs to be 0 for texture `write` functions. Metal 2.3 and later lift this restriction for Apple silicon.

In Metal 4.1 and later, the texture `read`, `sparse_read`, `block_read`, and `sparse_block_read` functions (see below) accept a sampler argument to support clamp-to-edge addressing and an `offset` for selected texture types. All texture types support clamp-to-edge. Clamp-to-edge applies only when the texture coordinate (after adding the offset) falls out of bounds. It doesn't apply to other parameters such as `face`, `array`, or `lod`. The `offset` argument is supported for 2D textures, 2D texture arrays, 3D textures, 2D depth textures, and 2D depth texture arrays. The `offset` is a per-component integer value applied to the texture coordinate before looking up each pixel. This integer value must be in the range `-8` to `+7`, with a default value of `0`. The offset argument also applies to atomic operations.

The following texture member functions are available to support sparse textures:

macOS: Metal 2.3 and later support sparse texture functions.
iOS: Metal 2.2 and later support sparse texture functions.
iPadOS and visionOS: Metal supports sparse texture functions.

- `sparse_sample` — sample from a sparse texture
- `sparse_sample_compare` — sample compare from a sparse texture
- `sparse_gather` — gather from a sparse texture
- `sparse_gather_compare` — gather compare from a sparse texture

These sparse texture member functions return a `sparse_color` structure that contains one or more color values and a residency flag. If any of the accessed pixels is not mapped, `resident` is set to `false`.

```
template <typename T>
struct sparse_color {
public:
  constexpr sparse_color(T value, bool resident) thread;

  // Indicates whether all memory addressed to retrieve
```

```
  // the value was mapped.
  constexpr bool resident() const thread;

  // Retrieve the color value.
  constexpr T const value() const thread;
};
```

For a sparse texture, to specify the minimum LOD level that the sampler can access, use `min_lod_clamp`.

For sections 6.13.1 through 6.13.16, the following abbreviations are used for the data types of function arguments and return values:

`Tv` denotes a 4-component vector type based on the templated type `<T>` for declaring the texture type:

- If `T` is `float`, `Tv` is `float4`.
- If `T` is `half`, `Tv` is `half4`.
- If `T` is `int`, `Tv` is `int4`.
- If `T` is `uint`, `Tv` is `uint4`.
- If `T` is `short`, `Tv` is `short4`.
- If `T` is `ushort`, `Tv` is `ushort4`.
- If `T` is `ulong`, `Tv` is `ulong4` (since Metal 3.1)

Metal doesn't support sampling of textures when T is `ulong`. Not all operations are supported on all types.

In Metal 3.1 and later, texture support atomic functions for element `T` where `T` is `int`, `uint`, or `ulong`:

- When the element T is `int` or `uint`, the texture on the Metal needs to be either `MTLPixelFormatR32Uint` or MTLPixelFormatR`32Sint`.
- When the element `T` is `ulong`, the texture on the Metal needs to be `MTLPixelFormatRG32Uint`.

The semantics of the atomic texture functions are the same as the atomic functions defined in Sec 6.16.

`sparse_color<Tv>` denotes a `sparse_color` structure that contains a four-component vector of color values, based on the templated type `<T>`, and a residency flag. These represent the return values of many sparse texture member functions.

`sparse_color<T>` denotes a `sparse_color` structure that contains a single value, based on the templated type `<T>`, and a residency flag. `T` typically represents a depth value that a sparse texture member function returns.

The following functions can be used to return the LOD (mip level) computation result for a simulated texture fetch:

macOS: Metal 2.2 and later support sparse texture functions.
iOS and iPadOS: Metal 2.3 and later support sparse texture functions.
visionOS: Metal supports sparse texture functions.

`calculate_unclamped_lod` — Calculates the level of detail that would be sampled for the given coordinates, ignoring any sampler parameter. The fractional part of this value contains the mip level blending weights, even if the sampler indicates a nearest mip selection.

`calculate_clamped_lod` — Similar to `calculate_unclamped_lod`, but additionally clamps the LOD to stay:

- within the texture mip count limits
- within the sampler's `lod_clamp` min and max values
- less than or equal to the sampler's `max_anisotropy` value

Only call the `calculate_unclamped_lod` and `calculate_clamped_lod` functions from a fragment function or a function you call with a fragment function; otherwise, the behavior is undefined.

All OS: Metal 4.1 and later support multipixel reads.

Use the following types and methods for multipixel reads:

- `block_read` — multipixel read from a texture
- `sparse_block_read` — multipixel read from a sparse texture

`block_read` and `sparse_block_read` are template member functions that take a width `W` and a height `H`. See Table 6.22 for which texture types support these functions and the permitted values of `W` and `H`.

The `block_read` member function returns a `block_color` structure that contains `W`×`H` color values.

```
template <typename T, int W, int H>
struct block_color {
  template <int X, int Y>
  constexpr T value() const thread;
};
```

The `sparse_block_read` member function returns a `sparse_block_color` structure that contains `W`×`H` color values and per-pixel residency flags. Each `resident<X, Y>()` returns false if the corresponding pixel is not mapped.

```
template <typename T, int W, int H>
struct sparse_block_color {
  template <int X, int Y>
  constexpr bool resident() const thread;

  template <int X, int Y>
```

```
  constexpr T value() const thread;
};
```

Table 6.22. Supported block read dimensions

Texture type	Dimensions
1D/1D Array	(W=2, H=1)
2D/2D Array	(W=2, H=1) and (W=1, H=2)
Cube/Cube Array	(W=2, H=1) and (W=1, H=2)
2D Depth/ 2D Depth Array	(W=2, H=1) and (W=1, H=2)
Cube Depth/ Cube Depth Array	(W=2, H=1) and (W=1, H=2)
Texture Buffer	(W=2, H=1)

6.13.1 1D Texture

This member function samples from a 1D texture.

```
Tv sample(sampler s, float coord) const
```

These member functions perform sampler-less reads from a 1D texture. Because mipmaps are not supported for 1D textures, `lod` needs to be `0`:

```
Tv read(uint coord, uint lod = 0) const

Tv read(ushort coord,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 1D texture:

```
Tv read(sampler s, int coord, uint lod = 0) const
```

These member functions can write to a 1D texture. Because mipmaps are not supported for 1D textures, `lod` needs to be `0`:

```
void write(Tv color, uint coord, uint lod = 0)

void write(Tv color, ushort coord,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

These member functions query a 1D texture. Since mipmaps are not supported for 1D textures, `get_num_mip_levels()` always return `0`, and `lod` needs to be `0` for `get_width()`:

```
uint get_width(uint lod = 0) const
uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function samples from a sparse 1D texture:

```
sparse_color<Tv> sparse_sample(sampler s, float coord) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 1D texture. Because mipmaps aren't supported for 1D textures, `lod` needs to be `0`:

```
sparse_color<Tv> sparse_read(ushort coord, ushort lod = 0) const
sparse_color<Tv> sparse_read(uint coord, uint lod = 0) const
```

Note: Sparse 1D textures aren't supported on Apple silicon.

In Metal 3.1 and later, these member functions execute an atomic load from a 1D texture:

```
Tv atomic_load(uint coord) const
Tv atomic_load(ushort coord) const
```

In Metal 3.1 and later, these member functions execute an atomic store to a 1D texture:

```
void atomic_store(Tv color, uint coord) const
void atomic_store (Tv color, ushort coord) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a 1D texture:

```
bool atomic_compare_exchange_weak(uint coord, thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort coord, thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a 1D texture:

```
Tv atomic_exchange(uint coord, Tv desired) const
Tv atomic_exchange(ushort coord, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a 1D texture, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint coord, Tv operand)
Tv atomic_fetch_op(ushort coord, Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a 1D texture:

```
void atomic_min(uint coord, ulong4 operand)
void atomic_min(ushort coord, ulong4 operand)
void atomic_max(uint coord, ulong4 operand)
void atomic_max(ushort coord, ulong4 operand)
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a 1D texture:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
                                 int coord,
                                 uint lod = 0) const
```

6.13.2 1D Texture Array

This member function samples from a 1D texture array:

```
Tv sample(sampler s, float coord, uint array) const
```

These member functions perform sampler-less reads from a 1D texture array. Because mipmaps are not supported for 1D textures, `lod` must be a zero constant:

```
Tv read(uint coord, uint array, uint lod = 0) const
Tv read(ushort coord, ushort array,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 1D texture array:

```
Tv read(sampler s, int coord, uint array, uint lod = 0) const
```

These member functions write to a 1D texture array. Because mipmaps are not supported for 1D textures, `lod` must be a zero constant:

```
void write(Tv color, uint coord, uint array, uint lod = 0)
void write(Tv color, ushort coord, ushort array,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

These member functions query a 1D texture array. Because mipmaps are not supported for 1D textures, `get_num_mip_levels()` always return `0`, and `lod` must be a zero constant for `get_width()`:

```
uint get_width(uint lod = 0) const
uint get_array_size() const
uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, this function samples from a sparse 1D texture array:

```
sparse_color<Tv sparse_sample(sampler s, float coord , uint array)
const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these functions perform a sampler-less read from a sparse 1D texture array. Because mipmaps are not supported for 1D texture arrays, `lod` must be a zero constant.

```
sparse_color<Tv sparse_read(ushort coord, ushort array,
                                          ushort lod = 0) const
sparse_color<Tv sparse_read(uint coord, uint array,
                                         uint lod = 0) const
```

Note: Sparse 1D texture arrays aren't supported on Apple silicon.

In Metal 3.1 and later, these member functions execute an atomic load from a 1D texture array:

```
Tv atomic_load(uint coord, uint array) const
Tv atomic_load(ushort coord, ushort array) const
```

In Metal 3.1 and later, these member functions execute an atomic store to a 1D texture array:

```
void atomic_store(Tv color, uint coord, uint array) const
void atomic_store (Tv color, ushort coord, ushort array) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a 1D texture array:

```
bool atomic_compare_exchange_weak(uint coord, uint array,
                                  thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort coord, ushort array,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a 1D texture array:

```
Tv atomic_exchange(uint coord, uint array, Tv desired) const
Tv atomic_exchange(ushort coord, ushort array, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a 1D texture array, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor`:

```
Tv atomic_fetch_op(uint coord, uint array,Tv operand)
Tv atomic_fetch_op(ushort coord, ushort array,Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a 1D texture array:

```
void atomic_min(uint coord, uint array, ulong4 operand)
void atomic_min(ushort coord, ushort array, ulong4 operand)
void atomic_max(uint coord, uint array, ulong4 operand)
void atomic_max(ushort coord, ushort array, ulong4 operand)
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a 1D texture array:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
```

```
                                int coord,
                                uint array,
                                uint lod = 0) const
```

6.13.3 2D Texture

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)

level(float lod)

gradient2d(float2 dPdx, float2 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a 2D texture:

```
Tv sample(sampler s, float2 coord, int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, lod_options options,
                   int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, bias bias_options,
          min_lod_clamp min_lod_clamp_options,
          int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, gradient2d grad_options,
          min_lod_clamp min_lod_clamp_options,
          int2 offset = int2(0)) const
```

These member functions perform sampler-less reads from a 2D texture:

```
Tv read(uint2 coord, uint lod = 0) const

Tv read(ushort2 coord,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D texture:

```
Tv read(sampler s, int2 coord, int2 offset = 0, uint lod = 0) const
```

These member functions write to a 2D texture. In macOS, for Metal 2.2 and earlier, `lod` must be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

```
void write(Tv color, uint2 coord, uint lod = 0)

void write(Tv color, ushort2 coord,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

This member functions gathers four samples for bilinear interpolation when sampling a 2D texture:

```
enum class component {x, y, z, w};

Tv gather(sampler s, float2 coord, int2 offset = int2(0),
          component c = component::x) const
```

These member functions query a 2D texture query:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions sample from a sparse 2D texture:

```
sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, bias options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               level options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               bias bias_options,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               gradient2d grad_options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord,
                               gradient2d grad_options,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 2D texture:

```
sparse_color-Tv sparse_read(ushort2 coord, ushort lod = 0) const
sparse_color-Tv sparse_read(uint2 coord, uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a sparse 2D texture:

```
sparse_color-Tv sparse_read(sampler s, int2 coord, int2 offset = 0,
                            uint lod = 0) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse 2D texture:

```
sparse_color-Tv sparse_gather(sampler s, float2 coord,
                              int2 offset = int2(0),
                              component c = component::x) const
```

In Metal 2.3 and later in iOS, and in Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float2 coord);
float calculate_unclamped_lod(sampler s, float2 coord);
```

In Metal 3.1 and later, these member functions execute an atomic load from a 2D texture:

```
Tv atomic_load(uint2 coord) const
Tv atomic_load(ushort2 coord) const
```

In Metal 4.1 and later, the following member function performs an atomic load with offset from a 2D texture:

```
Tv atomic_load(int2 coord, int2 offset) const
```

In Metal 3.1 and later, these member functions execute an atomic store to a 2D texture:

```
void atomic_store(Tv color, uint2 coord) const
void atomic_store (Tv color, ushort2 coord) const
```

In Metal 4.1 and later, the following member function performs an atomic store with offset to a 2D texture:

```
void atomic_store(Tv color, int2 coord, int2 offset) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a 2D texture:

```
bool atomic_compare_exchange_weak(uint2 coord, thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort2 coord,thread Tv *expected,
                                  Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic compare and exchange with offset to a 2D texture:

```
bool atomic_compare_exchange_weak(int2 coord, int2 offset,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a 2D texture:

```
Tv atomic_exchange(uint2 coord, Tv desired) const
Tv atomic_exchange(ushort2 coord, Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic exchange with offset to a 2D texture:

```
Tv atomic_exchange(int2 coord, int2 offset, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a 2D texture, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint2 coord, Tv operand)
Tv atomic_fetch_op(ushort2 coord, Tv operand) const
```

In Metal 4.1 and later, the following member functions perform an atomic fetch and modify with offset to a 2D texture:

```
Tv atomic_fetch_op(int2 coord, int2 offset, Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a 2D texture:

```
void atomic_min(uint2 coord, ulong4 operand)
void atomic_min(ushort2 coord, ulong4 operand)
void atomic_max(uint2 coord, ulong4 operand)
```

```
void atomic_max(ushort2 coord, ulong4 operand)
```

In Metal 4.1 and later, the following member functions perform an atomic `min` or `max` with `offset` to a 2D texture:

```
void atomic_min(int2 coord, int2 offset, ulong4 operand) const
void atomic_max(int2 coord, int2 offset, ulong4 operand) const
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a 2D texture and a sparse 2D texture:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
                                 int2 coord,
                                 int2 offset = 0,
                                 uint lod = 0) const


template <int W, int H>
sparse_block_color<Tv, W, H> sparse_block_read(sampler s,
                                               int2 coord,
                                               int2 offset = 0,
                                               uint lod = 0) const
```

6.13.3.1 2D Texture Sampling Example

The following code shows several uses of the 2D texture sample function, depending upon its arguments:

```
texture2d<float> tex;
sampler s;
float2 coord;
int2 offset;
float lod;

// No optional arguments.
float4 clr = tex.sample(s, coord);

// Sample using a mipmap level.
clr = tex.sample(s, coord, level(lod));

// Sample with an offset.
clr = tex.sample(s, coord, offset);

// Sample using a mipmap level and an offset.
clr = tex.sample(s, coord, level(lod), offset);
```

6.13.4 2D Texture Array

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)

level(float lod)

gradient2d(float2 dPdx, float2 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a 2D texture array:

```
Tv sample(sampler s, float2 coord, uint array,
          int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, uint array, lod_options options,
          int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, uint array, bias bias_options,
          min_lod_clamp min_lod_clamp_options,
          int2 offset = int2(0)) const

Tv sample(sampler s, float2 coord, uint array,
          gradient2d grad_options,
          min_lod_clamp min_lod_clamp_options,
          int2 offset = int2(0)) const
```

These member functions perform sampler-less reads from a 2D texture array:

```
Tv read(uint2 coord, uint array, uint lod = 0) const

Tv read(ushort2 coord, ushort array,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D texture array:

```
Tv read(sampler s, int2 coord, uint array,
        int2 offset = 0, uint lod = 0) const
```

These member functions write to a 2D texture array. In macOS, for Metal 2.2 and earlier, `lod` must be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

```
void write(Tv color, uint2 coord, uint array, uint lod = 0)

void write(Tv color, ushort2 coord, ushort array,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

This member functions gathers four samples for bilinear interpolation when sampling a 2D texture array:

```
Tv gather(sampler s, float2 coord, uint array,
          int2 offset = int2(0),
          component c = component::x) const
```

These member functions query a 2D texture array:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_array_size() const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions sample from a sparse 2D texture array:

```
sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               bias options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               level options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               bias bias_options,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               gradient2d options,
                               int2 offset = int2(0)) const

sparse_color<Tv> sparse_sample(sampler s, float2 coord, uint array,
                               gradient2d grad_options,
                               min_lod_clamp min_lod_clamp_options,
                               int2 offset = int2(0)) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these functions perform a sampler-less read from a sparse 2D texture array:

```
sparse_color-Tv sparse_read(ushort2 coord, ushort array,
                            ushort lod = 0) const

sparse_color-Tv sparse_read(uint2 coord, uint array,
                            uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multipixel read from a sparse 2D texture array:

```
sparse_color-Tv sparse_read(sampler s, int2 coord, uint array,
                            int2 offset = 0, uint lod = 0) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, this function gathers four samples for bilinear interpolation from a sparse 2D texture array:

```
sparse_color-Tv sparse_gather(sampler s, float2 coord, uint array,
                              int2 offset = int2(0),
                              component c = component::x) const
```

In Metal 2.3 and later in iOS, and in Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float2 coord);

float calculate_unclamped_lod(sampler s, float2 coord);
```

These member functions execute an atomic load from a 2D texture array in Metal 3.1 and later:

```
Tv atomic_load(uint2 coord, uint array) const

Tv atomic_load(ushort2 coord, ushort array) const
```

In Metal 4.1 and later, the following member function performs an atomic load with offset from a 2D texture array:

```
Tv atomic_load(int2 coord, uint array, int2 offset) const
```

In Metal 3.1 and later, these member functions execute an atomic store to a 2D texture array:

```
void atomic_store(Tv color, uint2 coord, uint array) const

void atomic_store (Tv color, ushort2 coord, ushort array) const
```

In Metal 4.1 and later, the following member function performs an atomic store with offset to a 2D texture array:

```
void atomic_store(Tv color, int2 coord, uint array,
                  int2 offset) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a 2D texture array:

```
bool atomic_compare_exchange_weak(uint2 coord, uint array,
                                  thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort2 coord, ushort array,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic compare and exchange with offset to a 2D texture array:

```
bool atomic_compare_exchange_weak(int2 coord, uint array,
                                  int2 offset, thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a 2D texture array:

```
Tv atomic_exchange(uint2 coord, uint array, Tv desired) const
Tv atomic_exchange(ushort2 coord, ushort array, Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic exchange with offset to a 2D texture array:

```
Tv atomic_exchange(int2 coord, uint array,
                   int2 offset, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a 2D texture array, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint2 coord, uint array,Tv operand)
Tv atomic_fetch_op(ushort2 coord, ushort array,Tv operand) const
```

In Metal 4.1 and later, the following member functions perform an atomic fetch and modify with offset to a 2D texture array:

```
Tv atomic_fetch_op(int2 coord, uint array,
                   int2 offset, Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a 2D texture array:

```
void atomic_min(uint2 coord, uint array, ulong4 operand)
void atomic_min(ushort2 coord, ushort array, ulong4 operand)
void atomic_max(uint2 coord, uint array, ulong4 operand)
void atomic_max(ushort2 coord, ushort array, ulong4 operand)
```

In Metal 4.1 and later, the following member functions perform an atomic `min` or `max` with `offset` to a 2D texture array:

```
void atomic_min(int2 coord, uint array,
                int2 offset, ulong4 operand) const
void atomic_max(int2 coord, uint array,
                int2 offset, ulong4 operand) const
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a 2D texture array and a sparse 2D texture array:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
                                 int2 coord,
                                 uint array,
                                 int2 offset = 0,
                                 uint lod = 0) const


template <int W, int H>
sparse_block_color<Tv, W, H> sparse_block_read(sampler s,
                                               int2 coord,
                                               uint array,
                                               int2 offset = 0,
                                               uint lod = 0) const
```

6.13.5 3D Texture

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)
level(float lod)
gradient3d(float3 dPdx, float3 dPdy)
min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a 3D texture:

```
Tv sample(sampler s, float3 coord, int3 offset = int3(0)) const
```

```
Tv sample(sampler s, float3 coord, lod_options options,
          int3 offset = int3(0)) const

Tv sample(sampler s, float3 coord, bias bias_options,
          min_lod_clamp min_lod_clamp_options,
          int3 offset = int3(0)) const

Tv sample(sampler s, float3 coord, gradient3d grad_options,
          min_lod_clamp min_lod_clamp_options,
          int3 offset = int3(0)) const
```

These member functions perform sampler-less reads from a 3D texture:

```
Tv read(uint3 coord, uint lod = 0) const

Tv read(ushort3 coord,
        ushort lod = 0) const // All OS: Metal 1.2 and later
```

In Metal 4.1 and later, the following member function performs a sampler read from a 3D texture:

```
Tv read(sampler s, int3 coord, int3 offset = 0, uint lod = 0) const
```

These member functions write to a 3D texture. In macOS, in Metal 2.2 and earlier, `lod` must be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

```
void write(Tv color, uint3 coord, uint lod = 0)

void write(Tv color, ushort3 coord,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

These member functions query a 3D texture:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_depth(uint lod = 0) const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these functions sample from a sparse 3D texture:

```
sparse_color<Tv> sparse_sample(sampler s, float3 coord,
                               int3 offset = int3(0)) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, bias options,
                               int3 offset = int3(0)) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord,
```

```
                              level options,
                              int3 offset = int3(0)) const
sparse_color-Tv sparse_sample(sampler s, float3 coord,

min_lod_clamp min_lod_clamp_options, int3 offset = int3(0)) const
sparse_color-Tv sparse_sample(sampler s, float3 coord,
                              bias bias_options,
                              min_lod_clamp min_lod_clamp_options,
                              int3 offset = int3(0)) const
sparse_color-Tv sparse_sample(sampler s, float3 coord,
                              gradient3d grad_options,
                              int3 offset = int3(0)) const
sparse_color-Tv sparse_sample(sampler s, float3 coord,
                              gradient3d grad_options,
                              min_lod_clamp min_lod_clamp_options,
                              int3 offset = int3(0)) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 3D texture:

```
sparse_color-Tv sparse_read(uint3 coord, uint lod = 0) const
sparse_color-Tv sparse_read(ushort3 coord, ushort lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a sparse 3D texture:

```
sparse_color-Tv sparse_read(sampler s, int3 coord, int3 offset = 0,
                            uint lod = 0) const
```

In Metal 2.3 and later in iOS, and in Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float3 coord)
float calculate_unclamped_lod(sampler s, float3 coord)
```

These member functions execute an atomic load from a 3D texture in Metal 3.1 and later:

```
Tv atomic_load(uint3 coord) const
Tv atomic_load(ushort3 coord) const
```

This member function executes an atomic load with offset from a 3D texture in Metal 4.1 and later:

```
Tv atomic_load(int3 coord, int3 offset) const
```

These member functions execute an atomic store to a 3D texture in Metal 3.1 and later:

```
void atomic_store(Tv color, uint3 coord) const
void atomic_store (Tv color, ushort3 coord) const
```

In Metal 4.1 and later, the following member function performs an atomic store with offset to a 3D texture:

```
void atomic_store(Tv color, int3 coord, int3 offset) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a 3D texture:

```
bool atomic_compare_exchange_weak(uint3 coord, thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort3 coord,thread Tv *expected,
                                  Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic compare and exchange with offset to a 3D texture:

```
bool atomic_compare_exchange_weak(int3 coord, int3 offset,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a 3D texture:

```
Tv atomic_exchange(uint3 coord, Tv desired) const
Tv atomic_exchange(ushort3 coord, Tv desired) const
```

In Metal 4.1 and later, the following member function performs an atomic exchange with offset to a 3D texture:

```
Tv atomic_exchange(int3 coord, int3 offset, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a 3D texture, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint3 coord, Tv operand)
Tv atomic_fetch_op(ushort3 coord, Tv operand) const
```

In Metal 4.1 and later, the following member functions perform an atomic fetch and modify with offset to a 3D texture:

```
Tv atomic_fetch_op(int3 coord, int3 offset, Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a 3D texture:

```
void atomic_min(uint3 coord, ulong4 operand)
void atomic_min(ushort3 coord, ulong4 operand)
void atomic_max(uint3 coord, ulong4 operand)
void atomic_max(ushort3 coord, ulong4 operand)
```

In Metal 4.1 and later, the following member functions perform an atomic `min` or `max` with `offset` to a 3D texture:

```
void atomic_min(int3 coord, int3 offset, ulong4 operand) const
void atomic_max(int3 coord, int3 offset, ulong4 operand) const
```

6.13.6 Cube Texture

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a cube texture:

```
Tv sample(sampler s, float3 coord) const
Tv sample(sampler s, float3 coord, lod_options options) const
Tv sample(sampler s, float3 coord, bias bias_options,
          min_lod_clamp min_lod_clamp_options) const
Tv sample(sampler s, float3 coord, gradientcube grad_options,
          min_lod_clamp min_lod_clamp_options) const
```

Table 6.23 describes a cube face and the number used to identify the face.

Table 6.23. Cube face number

Face number	Cube face
0	Positive X

Face number	Cube face
1	Negative X
2	Positive Y
3	Negative Y
4	Positive Z
5	Negative Z

This member function gathers four samples for bilinear interpolation when sampling a cube texture:

```
Tv gather(sampler s, float3 coord, component c = component::x) const
```

These member functions perform sampler-less reads from a cube texture:

```
Tv read(uint2 coord, uint face, uint lod = 0) const

Tv read(ushort2 coord, ushort face,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a cube texture:

```
Tv read(sampler s, int2 coord, uint face, uint lod = 0) const
```

These member functions write to a cube texture. In macOS, for Metal 2.2 and earlier, `lod` must be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

```
void write(Tv color, uint2 coord, uint face, uint lod = 0)

void write(Tv color, ushort2 coord, ushort face,
                     ushort lod = 0) // All OS: Metal 1.2 and later.
```

These member functions query a cube texture:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse cube texture:

```
sparse_color-Tv sparse_sample(sampler s, float3 coord) const

sparse_color-Tv sparse_sample(sampler s, float3 coord, bias options)
const

sparse_color-Tv sparse_sample(sampler s, float3 coord,
                              level options) const

sparse_color-Tv sparse_sample(sampler s, float3 coord,

min_lod_clamp min_lod_clamp_options) const

sparse_color-Tv sparse_sample(sampler s, float3 coord,
                         bias bias_options,
                         min_lod_clamp min_lod_clamp_options) const

sparse_color-Tv sparse_sample(sampler s, float3 coord,
                            gradientcube grad_options) const

sparse_color-Tv sparse_sample(sampler s, float3 coord,
                         gradientcube grad_options,
                         min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse cube texture:

```
sparse_color-Tv sparse_read(ushort2 coord, ushort face, ushort lod =
0) const

sparse_color-Tv sparse_read(uint2 coord, uint face, uint lod = 0)
const
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a sparse cube texture:

```
sparse_color-Tv sparse_read(sampler s, int2 coord, uint face,
                            uint lod = 0) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse cube texture:

```
sparse_color-Tv sparse_gather(sampler s, float3 coord,
                              component c = component::x) const
```

In Metal 2.3 and later in iOS, and Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float3 coord);

float calculate_unclamped_lod(sampler s, float3 coord);
```

In Metal 4 and later, these member functions execute an atomic load from a cube texture:

```
Tv atomic_load(uint2 coord, uint face) const
Tv atomic_load(ushort2 coord, ushort face) const
```

In Metal 4 and later, these member functions execute an atomic store to a cube texture:

```
void atomic_store(Tv color, uint2 coord, uint face) const
void atomic_store (Tv color, ushort2 coord, ushort face) const
```

In Metal 4 and later, these member functions execute an atomic compare and exchange to a cube texture:

```
bool atomic_compare_exchange_weak(uint2 coord, uint face,
                                  thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort2 coord, ushort face,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 4 and later, these member functions execute an atomic exchange to a cube texture:

```
Tv atomic_exchange(uint2 coord, uint face, Tv desired) const
Tv atomic_exchange(ushort2 coord, ushort face, Tv desired) const
```

In Metal 4 and later, these member functions execute an atomic fetch and modify to a cube texture, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint2 coord, uint face, Tv operand)
Tv atomic_fetch_op(ushort2 coord, ushort face, Tv operand) const
```

In Metal 4 and later, these member functions execute an atomic `min` or `max` to a cube texture:

```
void atomic_min(uint2 coord, uint face, ulong4 operand)
void atomic_min(ushort2 coord, ushort face, ulong4 operand)
void atomic_max(uint2 coord, uint face, ulong4 operand)
void atomic_max(ushort2 coord, ushort face, ulong4 operand)
```

In Metal 4.1 and later, the following member functions performs a multi-pixel read from a cube texture and a sparse cube texture:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
                                 int2 coord,
                                 uint face,
                                 uint lod = 0) const


template <int W, int H>
sparse_block_color<Tv, W, H> sparse_block_read(sampler s,
                                               int2 coord,
                                               uint face,
                                               uint lod = 0) const
```

6.13.7 Cube Texture Array

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)

level(float lod)

gradientcube(float3 dPdx, float3 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a cube texture array:

```
Tv sample(sampler s, float3 coord, uint array) const

Tv sample(sampler s, float3 coord, uint array,
          lod_options options) const

Tv sample(sampler s, float3 coord, uint array, bias bias_options,
          min_lod_clamp min_lod_clamp_options) const

Tv sample(sampler s, float3 coord, uint array,
          gradientcube grad_options,
          min_lod_clamp min_lod_clamp_options) const
```

This member function gathers four samples for bilinear interpolation when sampling a cube texture array:

```
Tv gather(sampler s, float3 coord, uint array,
                  component c = component::x) const
```

These member functions perform sampler-less reads from a cube texture array:

```
Tv read(uint2 coord, uint face, uint array, uint lod = 0) const

Tv read(ushort2 coord, ushort face, ushort array,
        ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a cube texture array:

```
Tv read(sampler s, int2 coord, uint face, uint array,
        uint lod = 0) const
```

These member functions write to a cube texture array. In macOS, for Metal 2.2 and earlier, `lod` must be a zero constant. Metal 2.3 and later lift this restriction for Apple silicon.

```
void write(Tv color, uint2 coord, uint face, uint array,
           uint lod = 0)

void write(Tv color, ushort2 coord, ushort face, ushort array,
           ushort lod = 0) // All OS: Metal 1.2 and later.
```

These member functions query a cube texture array:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_array_size() const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions sample from a sparse cube texture array:

```
sparse_color<Tv> sparse_sample(sampler s, float3 coord,
                               uint array) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                               bias options) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                               level options) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                           min_lod_clamp min_lod_clamp_options) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                           bias bias_options,
                           min_lod_clamp min_lod_clamp_options) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                               gradientcube options) const

sparse_color<Tv> sparse_sample(sampler s, float3 coord, uint array,
                           gradientcube grad_options,
                           min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse cube texture array:

```
sparse_color-Tv sparse_read(ushort2 coord, ushort face,
                            ushort array, ushort lod = 0) const
sparse_color-Tv sparse_read(uint2 coord, uint face,
                            uint array, uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multi-pixel read from a sparse cube texture array:

```
sparse_color-Tv sparse_read(sampler s, int2 coord, uint face,
                            uint array, uint lod = 0) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse cube texture array:

```
sparse_color-Tv sparse_gather(sampler s, float3 coord, uint array,
                              component c = component::x) const
```

In Metal 2.3 and later in iOS, and in Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float3 coord);
float calculate_unclamped_lod(sampler s, float3 coord);
```

In Metal 4 and later, these member functions execute an atomic load from a cube texture array:

```
Tv atomic_load(uint2 coord, uint face, uint array) const
Tv atomic_load(ushort2 coord, ushort face, ushort array) const
```

In Metal 4 and later, these member functions execute an atomic store to a cube texture array:

```
void atomic_store(Tv color, uint2 coord, uint face,
                  uint array) const
void atomic_store (Tv color, ushort2 coord, ushort face,
                   ushort array) const
```

In Metal 4 and later, these member functions execute an atomic compare and exchange to a cube texture array:

```
bool atomic_compare_exchange_weak(uint2 coord, uint face,
                                  uint array,
                                  thread Tv *expected,
                                  Tv desired) const

bool atomic_compare_exchange_weak(ushort2 coord, ushort face,
                                  ushort array,
                                  thread Tv *expected,
                                  Tv desired) const
```

In Metal 4 and later, these member functions execute an atomic exchange to a cube texture array:

```
Tv atomic_exchange(uint2 coord, uint face, uint array,
                   Tv desired) const

Tv atomic_exchange(ushort2 coord, ushort face, ushort array,
                   Tv desired) const
```

In Metal 4 and later, these member functions execute an atomic fetch and modify to a cube texture array, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint2 coord, uint face, uint array, Tv operand)

Tv atomic_fetch_op(ushort2 coord, ushort face, ushort array,
                   Tv operand) const
```

In Metal 4 and later, these member functions execute an atomic `min` or `max` to a cube texture array:

```
void atomic_min(uint2 coord, uint face, uint array, ulong4 operand)

void atomic_min(ushort2 coord, ushort face, ushort array,
                ulong4 operand)

void atomic_max(uint2 coord, uint face, uint array, ulong4 operand)

void atomic_max(ushort2 coord, ushort face, ushort array,
                ulong4 operand)
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a cube texture array and a sparse cube texture array:

```
template <int W, int H>
block_color<Tv, W, H> block_read(sampler s,
```

```
                                    int2 coord,
                                    uint face,
                                    uint array,
                                    uint lod = 0) const

template <int W, int H>
sparse_block_color<Tv, W, H> sparse_block_read(sampler s,
                                               int2 coord,
                                               uint face,
                                               uint array,
                                               uint lod = 0) const
```

6.13.8 2D Multisampled Texture

These member functions perform sampler-less reads from a 2D multisampled texture:

```
Tv read(uint2 coord, uint sample) const

Tv read(ushort2 coord,
        ushort sample) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D multisampled texture:

```
Tv read(sampler s, int2 coord, uint sample) const
```

If you have customized sample positions (set with the `setSamplePositions:count:` method of `MTLRenderPassDescriptor`), then `read(coord, sample)` returns the data for the sample at the programmed sample position.

These member functions query a 2D multisampled texture:

```
uint get_width() const

uint get_height() const

uint get_num_samples() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 2D multisampled texture:

```
sparse_color<Tv> sparse_read(ushort2 coord, ushort sample) const

sparse_color<Tv> sparse_read(uint2 coord, uint sample) const
```

In Metal 4.1 and later, the following member function performs a sampler read from a sparse 2D multisampled texture:

```
sparse_color<Tv> sparse_read(sampler s, int2 coord,
                             uint sample) const
```

6.13.9 2D Multisampled Texture Array

macOS: Metal 2 and later support 2D multisampled texture array.

iOS and iPadOS: Metal 2.3 and later support 2D multisampled texture array.
visionOS: Metal supports 2D multisampled texture array.

The following member functions can perform sampler-less reads from a 2D multisampled texture array:

```
Tv read(uint2 coord, uint array, uint sample) const
Tv read(ushort2 coord, ushort array, ushort sample) const
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D multisampled texture array:

```
Tv read(sampler s, int2 coord, uint array, uint sample) const
```

These member functions query a 2D multisampled texture array:

```
uint get_width() const
uint get_height() const
uint get_num_samples() const
uint get_array_size() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these functions perform a sampler-less read from a sparse 2D multisampled texture array:

```
sparse_color<Tv> sparse_read(ushort2 coord, ushort array,
                                            ushort sample) const
sparse_color<Tv> sparse_read(uint2 coord, uint array,
                                          uint sample) const
```

In Metal 4.1 and later, this member function performs a sampler read from a sparse 2D multisampled texture array:

```
sparse_color<Tv> sparse_read(sampler s, int2 coord, uint array,
                             uint sample) const
```

6.13.10 2D Depth Texture

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)
level(float lod)
```

```
gradient2d(float2 dPdx, float2 dPdy)
min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a 2D depth texture:

```
T sample(sampler s, float2 coord, int2 offset = int2(0)) const
T sample(sampler s, float2 coord, lod_options options,
         int2 offset = int2(0)) const
T sample(sampler s, float2 coord, bias bias_options,
         min_lod_clamp min_lod_clamp_options,
         int2 offset = int2(0)) const
T sample(sampler s, float2 coord, gradient2d grad_options,
         min_lod_clamp min_lod_clamp_options,
         int2 offset = int2(0)) const
```

These member functions sample from a 2D depth texture and compare a single component against the comparison value:

```
T sample_compare(sampler s, float2 coord, float compare_value,
                 int2 offset = int2(0)) const
T sample_compare(sampler s, float2 coord, float compare_value,
                 lod_options options, int2 offset = int2(0)) const
```

`T` must be a `float` type.

`sample_compare` performs a comparison of the `compare_value` value against the pixel value (`1.0` if the comparison passes, and `0.0` if it fails). These comparison result values per-pixel are then blended together as in normal texture filtering and the resulting value between `0.0` and `1.0` is returned. In macOS, Metal 2.2 and earlier don't support `lod_options` values `level` and `min_lod_clamp` (the latter, in Metal 2.2 and later); `lod` must be a zero constant. Metal 2.3 and later lift this restriction for `lod_options` for Apple silicon.

These member functions perform sampler-less reads from a 2D depth texture:

```
T read(uint2 coord, uint lod = 0) const
T read(ushort2 coord,
       ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D depth texture:

```
T read(sampler s, int2 coord, int2 offset = 0, uint lod = 0) const
```

This built-in function gathers four samples for bilinear interpolation when sampling a 2D depth texture:

```
Tv gather(sampler s, float2 coord, int2 offset = int2(0)) const
```

This member function gathers four samples for bilinear interpolation when sampling a 2D depth texture and comparing these samples with a specified comparison value (`1.0` if the comparison passes, and `0.0` if it fails):

```
Tv gather_compare(sampler s, float2 coord, float compare_value,
                  int2 offset = int2(0)) const
```

`T` must be a `float` type.

The following member functions query a 2D depth texture:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions sample from a sparse 2D depth texture:

```
sparse_color<T> sparse_sample(sampler s, float2 coord,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, bias options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, level options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord,
                              bias bias_options,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord
                              gradient2d grad_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord,
                              gradient2d grad_options,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const
```

In Metal 2.2 and later in iOS, and in Metal 2.3 and later in macOS, these member functions sample from a sparse 2D depth texture and compare a single component against a comparison value:

```
sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                                     float compare_value,
                                     int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                                     float compare_value,
                                     bias options,
                                     int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                                     float compare_value,
                                     level options,
                                     int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    float compare_value,
                    min_lod_clamp min_lod_clamp_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord
                    float compare_value, bias bias_options,
                    min_lod_clamp min_lod_clamp_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    float compare_value, gradient2d grad_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    float compare_value, gradient2d grad_options,
                    min_lod_clamp min_lod_clamp_options,
                    int2 offset = int2(0)) const
```

In Metal 2.2 and later, in iOS and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 2D depth texture:

```
sparse_color-T sparse_read(ushort2 coord, ushort lod = 0) const

sparse_color-T sparse_read(uint2 coord, uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multipixel read from a sparse 2D depth texture:

```
sparse_color-T sparse_read(sampler s, int2 coord, int2 offset = 0,
                           uint lod = 0) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse 2D depth texture:

```
sparse_color-Tv sparse_gather(sampler s, float2 coord,
                              int2 offset = int2(0),
                              component c = component::x) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers those samples and compares them against a comparison value from a sparse 2D depth texture:

```
sparse_color-Tv sparse_gather_compare(sampler s, float2 coord,
                                      float compare_value,
                                      int2 offset = int2(0)) const
```

In Metal 2.3 and later in iOS, and Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float2 coord);

float calculate_unclamped_lod(sampler s, float2 coord);
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a 2D depth texture and a sparse 2D depth texture:

```
template <int W, int H>
block_color<T, W, H> block_read(sampler s,
                                int2 coord,
                                int2 offset = 0,
                                uint lod = 0) const


template <int W, int H>
sparse_block_color<T, W, H> sparse_block_read(sampler s,
                                              int2 coord,
                                              int2 offset = 0,
                                              uint lod = 0) const
```

6.13.11 2D Depth Texture Array

The member functions in this section use the following data types and constructor functions to set the sampling option fields of their `lod_options` parameter:

```
bias(float value)

level(float lod)

gradient2d(float2 dPdx, float2 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a 2D depth texture array:

```
T sample(sampler s, float2 coord, uint array,
         int2 offset = int2(0)) const

T sample(sampler s, float2 coord, uint array, lod_options options,
         int2 offset = int2(0)) const

T sample(sampler s, float2 coord, uint array, bias bias_options,
         min_lod_clamp min_lod_clamp_options,
```

```
        int2 offset = int2(0)) const

T sample(sampler s, float2 coord, uint array,
        gradient2d grad_options,
        min_lod_clamp min_lod_clamp_options,
        int2 offset = int2(0)) const
```

These member functions sample from a 2D depth texture array and compare a single component to a value where `T` is a `float` type:

```
T sample_compare(sampler s, float2 coord, uint array,
                 float compare_value,int2 offset = int2(0)) const

T sample_compare(sampler s, float2 coord, uint array,
                 float compare_value, lod_options options,
                 int2 offset = int2(0)) const
```

The `lod_options` fields support are:

- `level`
- `bias` for all iOS Metal versions and macOS Metal 2.3 and later for Apple silicon
- `gradient` for iOS Metal versions and macOS Metal 2.3 and later for Apple silicon
- `min_lod_clamp` for Metal 2.2 and later
  - Must be `0` for Metal 2.2 and later
  - Can be any value for all iOS Metal versions and macOS Metal 2.3 and later for Apple silicon

These member functions read from a 2D depth texture array without using a sampler:

```
T read(uint2 coord, uint array, uint lod = 0) const

T read(ushort2 coord, ushort array,
       ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D depth texture array:

```
T read(sampler s, int2 coord, uint array,
       int2 offset = 0, uint lod = 0) const
```

This member function gathers four samples for bilinear interpolation when sampling a 2D depth texture array:

```
Tv gather(sampler s, float2 coord, uint array,
          int2 offset = int2(0)) const
```

This member function gathers four samples for bilinear interpolation when sampling a 2D depth texture array and compares them to a value where `Tv` is a float vector type:

```
Tv gather_compare(sampler s, float2 coord, uint array,
```

```
                  float compare_value, int2 offset = int2(0)) const
```

The following member functions query a 2D depth texture array:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_array_size() const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse 2D depth texture array:

```
sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              bias options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              level options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              bias bias_options,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              gradient2d grad_options,
                              int2 offset = int2(0)) const

sparse_color<T> sparse_sample(sampler s, float2 coord, uint array,
                              gradient2d grad_options,
                              min_lod_clamp min_lod_clamp_options,
                              int2 offset = int2(0)) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these functions sample from a sparse 2D depth texture array and compare a single component to a comparison value:

```
sparse_color<T> sparse_sample_compare(sampler s, float2 coord,
                    uint array, float compare_value,
                    int2 offset = int2(0)) const

sparse_color<T> sparse_sample_compare(sampler s, float2 coord,
```

```
                    uint array, float compare_value,
                    bias options, int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    uint array, float compare_value,
                    level options, int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    uint array,float compare_value,
                    min_lod_clamp min_lod_clamp_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    uint array, float compare_value,
                    bias bias_options,
                    min_lod_clamp min_lod_clamp_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                    uint array,
                    float compare_value, gradient2d grad_options,
                    int2 offset = int2(0)) const

sparse_color-T sparse_sample_compare(sampler s, float2 coord,
                   uint array,float compare_value,
                   gradient2d grad_options,
                   min_lod_clamp min_lod_clamp_options,
                   int2 offset = int2(0)) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these functions read from a sparse 2D depth texture array without a sampler:

```
sparse_color-T sparse_read(ushort2 coord, uint array,
                           ushort lod = 0) const

sparse_color-T sparse_read(uint2 coord, uint array,
                           uint lod = 0) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this function gathers four samples for bilinear interpolation from a sparse 2D depth texture array:

```
sparse_color-Tv sparse_gather(sampler s, float2 coord, uint array,
                              int2 offset = int2(0),
                              component c = component::x) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this function gathers those samples and compares them against a value from a sparse 2D depth texture array:

```
sparse_color-Tv sparse_gather_compare(sampler s, float2 coord, uint
array,
```

```
float compare_value, int2 offset = int2(0)) const
```

In Metal 2.3 and later in iOS, and Metal 2.2 and later in macOS, these functions simulate a texture fetch and return a LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float2 coord);

float calculate_unclamped_lod(sampler s, float2 coord);
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a 2D depth texture array and a sparse 2D depth texture array:

```
template <int W, int H>
block_color<T, W, H> block_read(sampler s,
                                int2 coord,
                                uint array,
                                int2 offset = 0,
                                uint lod = 0) const


template <int W, int H>
sparse_block_color<T, W, H> sparse_block_read(sampler s,
                                              int2 coord,
                                              uint array,
                                              int2 offset = 0,
                                              uint lod = 0) const
```

6.13.12 2D Multisampled Depth Texture

The following member functions can perform sampler-less reads from a 2D multisampled depth texture:

```
T read(uint2 coord, uint sample) const

T read(ushort2 coord,
       ushort sample) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D multisampled depth texture:

```
T read(sampler s, int2 coord, uint sample) const
```

The following member functions query a 2D multisampled depth texture:

```
uint get_width() const

uint get_height() const

uint get_num_samples() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 2D multisampled depth texture:

```
sparse_color-T sparse_read(ushort2 coord, ushort sample) const
sparse_color-T sparse_read(uint2 coord, uint sample) const
```

In Metal 4.1 and later, the following member function performs a sampler read from a sparse 2D multisampled depth texture:

```
sparse_color-T sparse_read(sampler s, int2 coord, uint sample) const
```

6.13.13 2D Multisampled Depth Texture Array

macOS: Metal 2 and later support 2D multisampled depth texture array.
iOS and iPadOS: Metal 2.3 and later support 2D multisampled depth texture array.
visionOS: Metal supports 2D multisampled depth texture array.

The following member functions perform sampler-less reads from a 2D multisampled depth texture array:

```
T read(uint2 coord, uint array, uint sample) const
T read(ushort2 coord, ushort array, ushort sample) const
```

In Metal 4.1 and later, the following member function performs a sampler read from a 2D multisampled depth texture array:

```
T read(sampler s, int2 coord, uint array, uint sample) const
```

The following member functions query a 2D multisampled depth texture array:

```
uint get_width(uint lod = 0) const
uint get_height(uint lod = 0) const
uint get_array_size() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse 2D multisampled depth texture array:

```
sparse_color-T sparse_read(ushort2 coord, ushort array,
                           ushort sample) const
sparse_color-T sparse_read(uint2 coord, uint array,
                           uint sample) const
```

In Metal 4.1 and later, the following member function performs a sampler read from a sparse 2D multisampled depth texture array:

```
sparse_color-T sparse_read(sampler s, int2 coord,
                           uint array, uint sample) const
```

6.13.14 Cube Depth Texture

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)

level(float lod)

gradientcube(float3 dPdx, float3 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

The following member functions sample from a cube depth texture:

```
T sample(sampler s, float3 coord) const

T sample(sampler s, float3 coord, lod_options options) const

T sample(sampler s, float3 coord, bias bias_options,
         min_lod_clamp min_lod_clamp_options) const

T sample(sampler s, float3 coord, gradientcube grad_options,
         min_lod_clamp min_lod_clamp_options) const
```

The following member functions sample from a cube depth texture and compare a single component against the specified comparison value:

```
T sample_compare(sampler s, float3 coord, float compare_value) const

T sample_compare(sampler s, float3 coord, float compare_value,
                 lod_options options) const
```

`T` must be a `float` type. In macOS, Metal 2.2 and earlier support `lod_options` values `level` and `min_lod_clamp` (the latter, in Metal 2.2 and later), and `lod` must be a zero constant. Metal 2.3 and later lift this restriction for `lod_options` for Apple silicon.

The following member functions perform sampler-less reads from a cube depth texture:

```
T read(uint2 coord, uint face, uint lod = 0) const

T read(ushort2 coord, ushort face,
       ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a cube depth texture:

```
T read(sampler s, int2 coord, uint face, uint lod = 0) const
```

This member function gathers four samples for bilinear interpolation when sampling a cube depth texture:

```
Tv gather(sampler s, float3 coord) const
```

This member function gathers four samples for bilinear interpolation when sampling a cube texture and comparing these samples with a specified comparison value:

```
Tv gather_compare(sampler s, float3 coord, float compare_value)
const
```

`T` must be a `float` type.

The following member functions query a cube depth texture:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse cube depth texture:

```
sparse_color<T> sparse_sample(sampler s, float3 coord) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  bias options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  level options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  min_lod_clamp min_lod_clamp_options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  bias bias_options,
                  min_lod_clamp min_lod_clamp_options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  gradientcube grad_options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                  gradientcube grad_options,
                  min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse cube depth texture and compare a single component against a comparison value:

```
sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                                      float compare_value) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value, bias options) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value, level options) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value,
```

```
                  min_lod_clamp min_lod_clamp_options) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value, bias bias_options,
                  min_lod_clamp min_lod_clamp_options) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value,
                  gradient2d grad_options) const

sparse_color<T> sparse_sample_compare(sampler s, float3 coord,
                  float compare_value, gradient2d grad_options,
                  min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse cube depth texture:

```
sparse_color<T> sparse_read(ushort2 coord, ushort face
                          ushort lod = 0) const

sparse_color<T> sparse_read(uint2 coord, uint face,
                          uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multipixel read from a sparse cube depth texture:

```
sparse_color<T> sparse_read(sampler s, int2 coord, uint face,
                          uint lod = 0) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse cube depth texture:

```
sparse_color<Tv> sparse_gather(sampler s, float3 coord) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers those samples and compare them against a comparison value from a sparse cube depth texture:

```
sparse_color<Tv> sparse_gather_compare(sampler s, float3 coord,
                                       float compare_value) const
```

In Metal 2.3 and later in iOS, and Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float3 coord);

float calculate_unclamped_lod(sampler s, float3 coord);
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a cube depth texture and a sparse cube depth texture:

```
template <int W, int H>
block_color<T, W, H> block_read(sampler s,
                                int2 coord,
                                uint face,
                                uint lod = 0) const


template <int W, int H>
sparse_block_color<T, W, H> sparse_block_read(sampler s,
                                              int2 coord,
                                              uint face,
                                              uint lod = 0) const
```

6.13.15 Cube Depth Texture Array

For the functions in this section, the following data types and corresponding constructor functions can specify sampling options (`lod_options`):

```
bias(float value)

level(float lod)

gradientcube(float3 dPdx, float3 dPdy)

min_lod_clamp(float lod) // All OS: Metal 2.2 and later.
```

These member functions sample from a cube depth texture array:

```
T sample(sampler s, float3 coord, uint array) const

T sample(sampler s, float3 coord, uint array,
         lod_options options) const

T sample(sampler s, float3 coord, uint array, bias bias_options,
         min_lod_clamp min_lod_clamp_options) const

T sample(sampler s, float3 coord, uint array,
         gradientcube grad_options,
         min_lod_clamp min_lod_clamp_options) const
```

These member functions sample from a cube depth texture array and compare a single component against the specified comparison value:

```
T sample_compare(sampler s, float3 coord, uint array,
                 float compare_value) const

T sample_compare(sampler s, float3 coord, uint array,
                 float compare_value, lod_options options) const
```

`T` must be a `float` type. In macOS, Metal 2.2 and earlier support `lod_options` values `level` and `min_lod_clamp` (the latter, in Metal 2.2 and later), and `lod` must be a zero constant. Metal 2.3 and later lift this restriction for `lod_options` for Apple silicon.

These member functions perform sampler-less reads from a cube depth texture array:

```
T read(uint2 coord, uint face, uint array, uint lod = 0) const

T read(ushort2 coord, ushort face, ushort array,
       ushort lod = 0) const // All OS: Metal 1.2 and later.
```

In Metal 4.1 and later, the following member function performs a sampler read from a cube depth texture array:

```
T read(sampler s, int2 coord, uint face, uint array,
       uint lod = 0) const
```

This member function gathers four samples for bilinear interpolation when sampling a cube depth texture:

```
Tv gather(sampler s, float3 coord, uint array) const
```

This member function gathers four samples for bilinear interpolation when sampling a cube depth texture and comparing these samples with a specified comparison value:

```
Tv gather_compare(sampler s, float3 coord, uint array,
                  float compare_value) const
```

`T` must be a `float` type.

These member functions query a cube depth texture:

```
uint get_width(uint lod = 0) const

uint get_height(uint lod = 0) const

uint get_array_size() const

uint get_num_mip_levels() const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse cube depth texture array:

```
sparse_color<T> sparse_sample(sampler s, float3 coord,
                              uint array) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                              uint array, bias options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                              uint array, level options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                    uint array,
                    min_lod_clamp min_lod_clamp_options) const

sparse_color<T> sparse_sample(sampler s, float3 coord,
                    uint array, bias bias_options,
                    min_lod_clamp min_lod_clamp_options) const
```

```
sparse_color-T sparse_sample(sampler s, float3 coord,
                             uint array,
                             gradientcube grad_options) const

sparse_color-T sparse_sample(sampler s, float3 coord,
                    uint array,
                    gradientcube grad_options,
                    min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions sample from a sparse cube depth texture array and compare a single component against a comparison value:

```
sparse_color-T sparse_sample_compare(sampler s, float3 coord,
                    uint array, float compare_value) const

sparse_color-T sparse_sample_compare(sampler s,float3 coord,
                    uint array, float compare_value,
                    bias options) const

sparse_color-T sparse_sample_compare(sampler s,float3 coord,
                    uint array, float compare_value,
                    level options) const

sparse_color-T sparse_sample_compare(sampler s, float3 coord,
                    uint array, float compare_value,
                    min_lod_clamp min_lod_clamp_options) const

sparse_color-T sparse_sample_compare(sampler s, float3 coord,
                    uint array, float compare_value,
                    bias bias_options,
                    min_lod_clamp min_lod_clamp_options) const

sparse_color-T sparse_sample_compare(sampler s, float3 coord,
                    uint array,float compare_value,
                    gradient2d grad_options) const

sparse_color-T sparse_sample_compare(sampler s, float3 coord,
                    uint array, float compare_value,
                    gradient2d grad_options,
                    min_lod_clamp min_lod_clamp_options) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, these member functions perform a sampler-less read from a sparse cube depth texture array:

```
sparse_color-T sparse_read(ushort2 coord, ushort face, ushort array,
                           ushort lod = 0) const

sparse_color-T sparse_read(uint2 coord, uint face, uint array,
                           uint lod = 0) const
```

In Metal 4.1 and later, the following member function performs a multipixel read from a sparse cube depth texture array:

```
sparse_color<T> sparse_read(sampler s, int2 coord, uint face,
                            uint array, uint lod = 0) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers four samples for bilinear interpolation from a sparse cube depth texture array:

```
sparse_color<vec<T, 4>> sparse_gather(sampler s, float3 coord,
                                      uint array) const
```

In Metal 2.2 and later in iOS, and Metal 2.3 and later in macOS, this member function gathers those samples and compare them against a comparison value from a sparse 2D depth texture:

```
sparse_color<vec<T, 4>> sparse_gather_compare(sampler s, float3 coord,
                                              uint array,
                                              float compare_value) const
```

In Metal 2.3 and later in iOS, and Metal 2.2 and later in macOS, these member functions simulate a texture fetch and return the LOD (mip level) computation result:

```
float calculate_clamped_lod(sampler s, float3 coord);

float calculate_unclamped_lod(sampler s, float3 coord);
```

In Metal 4.1 and later, the following member functions performs a multipixel read from a cube depth texture array and a sparse cube depth texture array:

```
template <int W, int H>
block_color<T, W, H> block_read(sampler s,
                                int2 coord,
                                uint face,
                                uint array,
                                uint lod = 0) const


template <int W, int H>
sparse_block_color<T, W, H> sparse_block_read(sampler s,
                                              int2 coord,
                                              uint face,
                                              uint array,
                                              uint lod = 0) const
```

6.13.16 Texture Buffer Functions

All OS: Metal 2.1 and later support texture buffers and these functions.

The following member functions can read from and write to an element in a texture buffer (also see section 2.9.1):

```
Tv read(uint coord) const;
void write(Tv color, uint coord);
```

In Metal 4.1 and later, the following member function performs a sampler read from a texture buffer:

```
Tv read(sampler s, int coord) const
```

In Metal 3.1 and later, these member functions execute an atomic load from a texture buffer:

```
Tv atomic_load(uint coord) const
Tv atomic_load(ushort coord) const
```

In Metal 3.1 and later, these member functions execute an atomic store to a texture buffer:

```
void atomic_store(Tv color, uint coord) const
void atomic_store (Tv color, ushort coord) const
```

In Metal 3.1 and later, these member functions execute an atomic compare and exchange to a texture buffer:

```
bool atomic_compare_exchange_weak(uint coord, thread Tv *expected,
                                  Tv desired) const
bool atomic_compare_exchange_weak(ushort coord, thread Tv *expected,
                                  Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic exchange to a texture buffer:

```
Tv atomic_exchange(uint coord, Tv desired) const
Tv atomic_exchange(ushort coord, Tv desired) const
```

In Metal 3.1 and later, these member functions execute an atomic fetch and modify to a texture buffer, where *op* is `add`, `and`, `max`, `min`, `or`, `sub`, or `xor` for `int`, and `uint` color type:

```
Tv atomic_fetch_op(uint coord, Tv operand)
Tv atomic_fetch_op(ushort coord, Tv operand) const
```

In Metal 3.1 and later, these member functions execute an atomic min or max to a texture buffer:

```
void atomic_min(uint coord, ulong4 operand)
void atomic_min(ushort coord, ulong4 operand)
void atomic_max(uint coord, ulong4 operand)
void atomic_max(ushort coord, ulong4 operand)
```

The following example uses the `read` method to access a texture buffer:

```
kernel void
myKernel(texture_buffer<float, access::read> myBuffer)
{
     uint index = …;
     float4 value = myBuffer.read(index);
}
```

Use the following method to query the number of elements in a texture buffer:

```
uint get_width() const;
```

6.13.17 Texture Synchronization Functions

All OS: Metal 1.2 and later support texture synchronization functions.

The texture `fence()` member function ensures that writes to the texture by a thread become visible to subsequent reads from that texture by the same thread (the thread that is performing the write). Texture types (including texture buffers) that you can declare with the `access::read_write` attribute support the `fence` function.

```
void fence()
```

The following example shows how to use a texture `fence` function to make sure that writes to a texture by a thread are visible to later reads to the same location by the same thread:

```
kernel void
my_kernel(texture2d<float, access::read_write> texA,
          …,
          ushort2 gid [[thread_position_in_grid]])
{
     float4 clr = …;
     texA.write(clr, gid);
     …
     // Use fence to ensure that writes by thread are
     // visible to later reads by the thread.
     texA.fence();

     clr_new = texA.read(gid);
     …
}
```

6.13.18 Null Texture Functions

All OS: Metal 1.2 and later support null texture functions.

macOS: Metal 2 and later support null texture functions for `texture2d_ms_array` and `depth2d_ms_array`.

Use the following functions to determine if a texture is a null texture. If the texture is a null texture, `is_null_texture` returns `true`; otherwise, return `false`:

```
bool is_null_texture(texture1d<T, access>);
bool is_null_texture(texture1d_array<T, access>);
bool is_null_texture(texture2d<T, access>);
bool is_null_texture(texture2d_array<T, access>);
bool is_null_texture(texture3d<T, access>);
bool is_null_texture(texturecube<T, access>);
bool is_null_texture(texturecube_array<T, access>);
bool is_null_texture(texture2d_ms<T, access>);
// Metal 2 and later support texture2d_ms_array in macOS, and
// Metal 2.3 and later in iOS.
bool is_null_texture(texture2d_ms_array<T, access>);
bool is_null_texture(depth2d<T, access>);
bool is_null_texture(depth2d_array<T, access>);
bool is_null_texture(depthcube<T, access>);
bool is_null_texture(depthcube_array<T, access>);
bool is_null_texture(depth2d_ms<T, access>);
// depth2d_ms_array is macOS only, in Metal 2 and later.
bool is_null_texture(depth2d_ms_array<T, access>);
```

The behavior of calling any texture member function with a null texture is undefined.

6.14 Imageblock Functions

macOS: Metal 2.3 and later support imageblocks for Apple silicon.
iOS: Metal 2 and later support imageblocks.
iPadOS and visionOS: Metal supports imageblocks.

This section lists the Metal member functions for imageblocks. (For more about the imageblock data type, see sections 2.11 and 5.6.)

The following member functions query information about the imageblock:

```
ushort get_width() const;
ushort get_height() const;
ushort get_num_samples() const;
```

Use the following member function to query the number of unique color entries for a specific location given by an (x, y) coordinate inside the imageblock:

```
ushort get_num_colors(ushort2 coord) const;
```

The following member function returns the color coverage mask (that is, whether a given color covers one or more samples in the imageblock). Each sample is identified by its bit position in the return value. If a bit is set, then this indicates that this sample uses the color index.

```
ushort get_color_coverage_mask(ushort2 coord, ushort color_index)
const;
```

`color_index` is a value from `0` to `get_num_colors() - 1`.

6.14.1 Functions for Imageblocks with Implicit Layout

Use the following functions to read or write an imageblock at pixel rate for a given (x, y) coordinate inside the imageblock:

```
T read(ushort2 coord) const;
```

```
void write(T data, ushort2 coord);
```

Use the following member function to read or write an imageblock at sample or color rate. `coord` specifies the (x, y) coordinate inside the imageblock, and `index` is the sample or color index.

```
enum class imageblock_data_rate { color, sample };
```

```
T read(ushort2 coord, ushort index,
       imageblock_data_rate data_rate) const;
```

```
void write(T data, ushort2 coord, ushort index,
           imageblock_data_rate data_rate);
```

Example:

```
struct Foo {
     float4 a [[color(0)]];
     int4 b [[color(1)]];
};

kernel void
my_kernel(imageblock<Foo, imageblock_layout_implicit> img_blk,
          ushort2 lid [[thread_position_in_threadgroup]] …)
{
     …
     Foo f = img_blk.read(lid); float4 r = f.a;
     …
     f.a = r;
     …
```

```
    img_blk.write(f, lid);
}
```

Use the following member function to write an imageblock with a color coverage mask. You must use this member function when writing to an imageblock at color rate:

```
void write(T data, ushort2 coord, ushort color_coverage_mask);
```

Use the following member functions to get a region of a slice for a given data member in the imageblock. You use these functions to write data associated with a specific data member described in the imageblock for all threads in the threadgroup to a specified region in a texture. `color_index` refers to the data member declared in the structure type specified in `imageblock<T>` with the `[[color(n)]]` attribute where `n` Is `color_index`. `size` is the actual size of the copied slice.

```
const imageblock_slice<E, imageblock_layout_implicit> slice(ushort
color_index) const;

const imageblock_slice<E, imageblock_layout_implicit> slice(ushort
color_index, ushort2 size) const;
```

The region to copy has an origin of (0,0). The `slice(…)` member function that does not have the argument `size` copies the entire width and height of the imageblock.

6.14.2 Functions for Imageblocks with Explicit Layout

Use the following member functions to get a reference to the imageblock data for a specific location given by an (x, y) coordinate inside the imageblock. Use these member functions when reading or writing data members in an imageblock at pixel rate.

```
threadgroup_imageblock T* data(ushort2 coord);

const threadgroup_imageblock T* data(ushort2 coord) const;
```

Use the following member functions to get a reference to the imageblock data for a specific location given by an (x, y) coordinate inside the imageblock and a sample or color index. Use these member functions when reading or writing data members in an imageblock at sample or color rate. `T` is the type specific in the `imageblock<T>` templated declaration. `coord` is the coordinate in the imageblock, and `index` is the sample or color index for a multisampled imageblock. `data_rate` specifies whether the index is a color or sample index. If `coord` refers to a location outside the imageblock dimensions or if `index` is an invalid index, the behavior of `data()` is undefined.

```
enum class imageblock_data_rate { color, sample };

threadgroup_imageblock T* data(ushort2 coord, ushort index,
imageblock_data_rate data_rate);

const threadgroup_imageblock T* data(ushort2 coord, ushort index,
imageblock_data_rate data_rate) const;
```

Calling the `data(coord)` member function for an imageblock that stores pixels at sample or color rate is equivalent to calling `data(coord, 0, imageblock_data_rate::sample)`.

Example:

```
struct Foo {
     rgba8unorm<half4> a;
     int b;
};

kernel void
my_kernel(imageblock<Foo> img_blk,
          ushort2 lid [[thread_position_in_threadgroup]] …)
{
     …
     threadgroup_imageblock Foo* f = img_blk.data(lid);
     half4 r = f->a;
     f->a = r;
     …
}
```

Use the following `write` member function to write an imageblock with a color coverage mask. You must use this member function when writing to an imageblock at color rate.

```
void write(T data, ushort2 coord, ushort color_coverage_mask);
```

Use the following `slice` member functions to get a region of a slice for a given data member in the imageblock structure. You use this function to write data associated with a specific data member described in the imageblock structure for all threads in the threadgroup to a specified region in a texture.

`data_member` is a data member declared in the structure type specified in `imageblock<T>`. `size` is the actual size of the copied slice.

```
const imageblock_slice<E, imageblock_layout_explicit>
slice(const threadgroup_imageblock E& data_member) const;
const imageblock_slice<E, imageblock_layout_explicit>
slice(const threadgroup_imageblock E& data_member, ushort2 size)
const;
```

The region to copy has an origin of (0,0). The `slice(…)` member function that doesn't have the argument `size` copies the entire width and height of the imageblock.

6.14.3 Writing an Imageblock Slice to a Region in a Texture

Use the following `write(…)` member function in these texture types to write pixels associated with a slice in the imageblock to a texture starting at a location that `coord` provides.

A `write` to a texture from an imageblock is out-of-bounds if, and only if, it meets any of these conditions:

- The accessed coordinates are out-of-bounds.

- The level of detail argument is out-of-bounds.
- Any part of the `imageblock_slice` accesses outside the texture.

An out-of-bounds `write` to a texture is undefined. Note that the write from `imageblock_slice` to a texture must have matching MSAA modes or the result is undefined.

For a 1D texture:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint coord, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort coord, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint coord, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort coord, ushort lod = 0);
```

For a 1D texture array:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint coord, uint array, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort coord, ushort array, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint coord, uint array, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort coord, ushort array, ushort lod = 0);
```

For a 2D texture:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint2 coord, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort2 coord, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint2 coord, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort2 coord, ushort lod = 0);
```

For a 2D MSAA texture:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint2 coord, uint lod = 0);
```

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort2 coord, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint2 coord, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort2 coord, ushort lod = 0);
```

For a 2D texture array:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                    uint2 coord, uint array, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort2 coord, ushort array, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint2 coord, uint array, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort2 coord, ushort array, ushort lod = 0);
```

For a cube texture:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint2 coord, uint face, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort2 coord, ushort face, ushort lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint2 coord, uint face, uint lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort2 coord, ushort face, ushort lod = 0);
```

For a cube texture array:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint2 coord, uint face, uint array, uint lod =
0);
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort2 coord, ushort face, ushort array, ushort
lod = 0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint2 coord, uint face, uint array, uint lod =
0);
void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort2 coord, ushort face, ushort array, ushort
lod = 0);
```

For a 3D texture:

```
void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   uint3 coord, uint lod = 0);

void write(imageblock_slice<E, imageblock_layout_explicit> slice,
                   ushort3 coord, ushort lod = 0);

void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   uint3 coord, uint lod = 0);

void write(imageblock_slice<E, imageblock_layout_implicit> slice,
                   ushort3 coord, ushort lod = 0);
```

Example:

```
struct Foo {
    half4 a;
    int b;
    float c;
};

kernel void
my_kernel(texture2d<half> src [[ texture(0) ]],
          texture2d<half, access::write> dst [[ texture(1) ]],
          imageblock<Foo> img_blk,
          ushort2 lid [[ thread_position_in_threadgroup ]],
          ushort2 gid [[ thread_position_in_grid ]])
{
    // Read the pixel from the input image using the thread ID.
    half4 clr = src.read(gid);

    // Get the image slice.
    threadgroup_imageblock Foo* f = img_blk.data(lid);
    // Write the pixel in the imageblock using the thread ID in
    // threadgroup.
    f->a = clr;

    // A barrier to make sure all threads finish writing to the
    // imageblock.
    //
    // In this case, each thread writes to its location in the
    // imageblock so a barrier isn't necessary.
    threadgroup_barrier(mem_flags::mem_threadgroup_imageblock);

    // Process the pixels in imageblock, and update the elements in
    // slice.
    process_pixels_in_imageblock(img_blk, gid, lid);

    // A barrier to make sure all threads finish writing to the
    // elements in the imageblock.
```

```
    threadgroup_barrier(mem_flags::mem_threadgroup_imageblock);

    // Write a specific element in an imageblock to the output
    // image. Only one thread in the threadgroup performs the
    // imageblock write.
    if (lid.x == 0 && lid.y == 0)
        dst.write(img_blk.slice(f->a), gid);
}
```

6.15 Pack and Unpack Functions

This section lists the Metal functions, defined in the header `<metal_pack>`, for converting a vector floating-point data to and from a packed integer value. Refer to subsections of section 8.7 for details on how to convert from an 8-, 10-, or 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value and vice-versa.

6.15.1 Unpack and Convert Integers to a Floating-Point Vector

Table 6.24 lists functions that unpack multiple values from a single unsigned integer and then converts them into floating-point values that are stored in a vector.

Table 6.24. Unpack functions

Built-in unpack functions	Description

`float4 unpack_unorm4x8_to_float(uint x)`
`float4 unpack_snorm4x8_to_float(uint x)`
`half4 unpack_unorm4x8_to_half(uint x)`
`half4 unpack_snorm4x8_to_half(uint x)`
	Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector.

`float4 unpack_unorm4x8_srgb_to_float(uint x)` `half4 unpack_unorm4x8_srgb_to_half(uint x)`	Unpack a 32-bit unsigned integer into four 8-bit signed or unsigned integers and then convert each 8-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 4-component vector. The RGB color values are converted from sRGB to linear RGB.

Built-in unpack functions	Description

`float2 unpack_unorm2x16_to_float(uint x)`
`float2 unpack_snorm2x16_to_float(uint x)`
`half2 unpack_unorm2x16_to_half(uint x)`
`half2 unpack_snorm2x16_to_half(uint x)`
	Unpack a 32-bit unsigned integer into two 16-bit signed or unsigned integers and then convert each 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value to generate a 2-component vector.

`float4 unpack_unorm10a2_to_float(uint x)`
`float3 unpack_unorm565_to_float(ushort x)`
`half4 unpack_unorm10a2_to_half(uint x)`
`half3 unpack_unorm565_to_half(ushort x)`
	Convert a `10a2` (1010102) or `565` color value to the corresponding normalized single- or half-precision floating-point vector.

`float4 unpack_snorm10a2_to_float(uint x)`
`half4 unpack_snorm10a2_to_half(uint x)`
 	
All OS: Metal 4 and later
	Convert a `10a2` (1010102) signed color value to the corresponding normalized single- or half-precision floating-point vector.

When converting from a 16-bit unsigned normalized or signed normalized value to a half-precision floating-point, the `unpack_unorm2x16_to_half` and `unpack_snorm2x16_to_half` functions may lose precision.

6.15.2 Convert Floating-Point Vector to Integers, then Pack the Integers

Table 6.25 lists functions that start with a floating-point vector, converts the components into integer values, and then packs the multiple values into a single unsigned integer.

Table 6.25. Pack functions

Built-in pack functions	Description

`uint pack_float_to_unorm4x8(float4 x)`
`uint pack_float_to_snorm4x8(float4 x)`
`uint pack_half_to_unorm4x8(half4 x)`
`uint pack_half_to_snorm4x8(half4 x)`
	Convert a four-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer.

Built-in pack functions	Description
`uint pack_float_to_srgb_unorm4x8(float4 x)` `uint pack_half_to_srgb_unorm4x8(half4 x)`	Convert a four-component vector normalized single- or half-precision floating-point value to four 8-bit integer values and pack these 8-bit integer values into a 32-bit unsigned integer. The color values are converted from linear RGB to sRGB.

`uint pack_float_to_unorm2x16(float2 x)`
`uint pack_float_to_snorm2x16(float2 x)`
`uint pack_half_to_unorm2x16(half2 x)`
`uint pack_half_to_snorm2x16(half2 x)`
	Convert a two-component vector of normalized single- or half-precision floating-point values to two 16-bit integer values and pack these 16-bit integer values into a 32-bit unsigned integer.

`uint pack_float_to_unorm10a2(float4)`
`ushort pack_float_to_unorm565(float3)`
`uint pack_half_to_unorm10a2(half4)`
`ushort pack_half_to_unorm565(half3)`
	Convert a three- or four-component vector of normalized single- or half-precision floating-point values to a packed, `10a2` (1010102) or `565` color integer value.

`uint pack_float_to_snorm10a2(float4)`
`uint pack_half_to_snorm10a2(half4)`
 	
All OS: Metal 4 and later.
	Convert a four-component vector of normalized single- or half-precision floating-point values to a packed `10a2` (1010102) signed color integer value.

6.16 Atomic Functions

The Metal programming language implements a subset of the C++17 atomics and synchronization operations. Metal atomic functions must operate on Metal atomic data, as described in section 2.6.

Atomic operations play a special role in making assignments in one thread visible to another thread. A synchronization operation on one or more memory locations is either an acquire operation, a release operation, or both. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both. In addition, there are relaxed atomic operations that are not synchronization operations.

There are only a few kinds of operations on atomic types, although there are many instances of those kinds. This section specifies each general kind.

Atomic functions are defined in the header `<metal_atomic>`.

6.16.1 Memory Order

All OS: Metal 4.1 and later support the additional memory order values, `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`.

The enumeration `memory_order` specifies the detailed regular (nonatomic) memory synchronization operations (see section 29.3 of the C++17 specification) and may provide for operation ordering (see Table 6.26):

```
enum memory_order {
     memory_order_relaxed,
     memory_order_acquire,
     memory_order_release,
     memory_order_acq_rel,
     memory_order_seq_cst
};
```

Table 6.26. Memory order enumeration values

Memory Description	Description
`memory_order_relaxed`	There are no synchronization or ordering constraints; only atomicity is required of this operation.
`memory_order_acquire` All OS: Metal 4.1 and later.	A load operation with this memory order performs the acquire operation on the affected memory location: prior writes made to other memory locations by the thread that performed the release become visible to this thread.
`memory_order_release` All OS: Metal 4.1 and later.	A store operation with this memory order performs the release operation: prior writes to other memory locations become visible to the threads that perform an acquire on the same memory location.
`memory_order_acq_rel` All OS: Metal 4.1 and later.	A load operation with this memory order performs the acquire operation on the affected memory location, and a store operation with this memory order performs the release operation.
`memory_order_seq_cst` All OS: Metal 4 and later.	Same as `memory_order_acq_rel`, plus a single total order exists in which all threads observe all modifications in the same order.

For atomic operations other than `atomic_thread_fence`, `memory_order_relaxed` is the only enumeration value. With `memory_order_relaxed`, there are no synchronization or ordering constraints; the operation only requires atomicity. These operations do not order memory, but they guarantee atomicity and modification order consistency. A typical use for relaxed memory ordering is updating counters, such as reference counters because this only requires atomicity, but neither ordering nor synchronization.

In Metal 3.2 and later, you can use `memory_order_seq_cst` on `atomic_thread_fence` to indicate that everything that happens before a store operation in one thread becomes a visible side effect in the thread that performs the load, and establishes a single total modification order of all tagged atomic operations.

In Metal 4.1 and later, you can specify `memory_order` values for atomic operations, `atomic_thread_fence`, `threadgroup_barrier`, and `simdgroup_barrier`. Sections 6.10 and 6.16.3 describe these functions and the valid orderings for each.

6.16.1.1 Relaxed Ordering

Atomic operations tagged with `memory_order_relaxed` are not synchronization operations. These operations don't order memory, but they guarantee atomicity and modification order consistency.

A typical use for relaxed memory ordering is updating counters, such as reference counters, because this requires only atomicity not ordering or synchronization.

6.16.1.2 Release-Acquire Ordering

If an atomic store in thread A uses `memory_order_release`, and an atomic load in thread B from the same variable uses `memory_order_acquire`, all memory writes (nonatomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A become *visible* to thread B. When the atomic load completes, thread B is guaranteed to see everything thread A wrote to memory.

The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable. Other threads can see a different order of memory accesses than either or both synchronized threads.

6.16.1.3 Sequentially Consistent Ordering

Atomic operations that use `memory_order_seq_cst` order memory the same way as release-acquire ordering — everything that happened before a store operation in one thread becomes visible to the thread that performed the load. They also establish a single total modification order of all atomic operations. Sequential ordering may be necessary for multiple producer–multiple consumer situations, where all consumers must observe all producer's actions occurring in the same order.

Note: Sequential consistency is lost when an atomic operation encounters an operation that doesn't use memory_order_seq_cst.

6.16.2 Thread Scope

All OS: Metal 3.2 and later support `thread_scope` for Apple silicon.

The enumeration `thread_scope` denotes a set of threads for the memory order constraint that the `memory_order` provides:

```
enum thread_scope {
  thread_scope_thread,
  thread_scope_simdgroup,
  thread_scope_threadgroup,
  thread_scope_device
}
```

Informally, the thread scope on a synchronization operation defines the set of threads with which this operation may synchronize, or which may synchronize with the operation. You use it with `atomic_thread_fence`.

6.16.3 Fence Functions

All OS: Metal 3.2 and later support `atomic_thread_fence` for Apple silicon.

The `atomic_thread_fence` establishes memory synchronization ordering of nonatomic and relaxed atomic accesses, according to the memory order and thread scope, without an associated atomic function:

```
void atomic_thread_fence(mem_flags flags, memory_order order,
                         thread_scope scope = thread_scope_device)
```

A fence operates on the following address space scopes:

- `threadgroup`, if `mem_flags` include `mem_threadgroup`
- `threadgroup_imageblock`, if `mem_flags` include `mem_threadgroup_imageblock`
- `object_data`, if `mem_flags` include `mem_object_data`
- `device`, if `mem_flags` include `mem_device`
- `texture`, if `mem_flags` include `mem_texture`

A fence accepts a scope parameter (see section 6.16.2) that denotes the set of threads affected by the fence's `order`. Depending on the value of `order` (see section 6.16.1), this operation:

- has no effects, if order == `memory_order_relaxed`
- is an acquire fence, if `order` == `memory_order_acquire`
- is a release fence, if `order` == `memory_order_release`
- is both an acquire fence and a release fence, if `order` == `memory_order_acq_rel`
- is a sequentially consistent acquire and release fence, if `order` == `memory_order_seq_cst`

An `atomic_thread_fence` imposes different synchronization constraints than an atomic store operation with the same `memory_order`. An atomic store-release operation prevents all preceding writes from moving past the store-release, and an `atomic_thread_fence` with `memory_order_seq_cst` ordering prevents all preceding writes from moving past all subsequent stores within that scope.

6.16.4 Atomic Functions

In addition, accesses to atomic objects may establish interthread synchronization and order nonatomic memory accesses as specified by `memory_order`.

In the atomic functions described in the subsections of this section:

- `A` refers to one of the atomic types.
- `C` refers to its corresponding nonatomic type.
- `M` refers to the type of the other argument for arithmetic operations. For atomic integer types, `M` is `C`.

Note that each atomic function may support only some types. The following sections indicate which type `A` Metal supports.

All OS: Metal 1 and later support functions with names that end with `_explicit` (such as `atomic_store_explicit` or `atomic_load_explicit`) unless otherwise indicated. Metal 3 supports the `atomic_float` for device memory only.

iOS: Metal 2 and later support the `atomic_store`, `atomic_load`, `atomic_exchange`, `atomic_compare_exchange_weak`, and `atomic_fetch_key` functions.

iPadOS and visionOS: Metal supports the `atomic_store`, `atomic_load`, `atomic_exchange`, `atomic_compare_exchange_weak`, and `atomic_fetch_key` functions.

6.16.4.1 Atomic Store Functions

These functions atomically replace the value pointed to by `object` with `desired`. These functions support atomic types `A` of `atomic_int`, `atomic_uint`, `atomic_bool`, and `atomic_float`. Atomic store supports `atomic_float` only for device memory.

All OS: Support for the `atomic_store_explicit` function with `memory_order_relaxed` supported, as indicated.

```
void atomic_store_explicit(threadgroup A* object, C desired,
                           memory_order order) // All OS: Since Metal 2.

void atomic_store_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 1.

void atomic_store_explicit(device A* object, C desired,
                           memory_order order) // All OS: Since Metal 2.

void atomic_store_explicit(volatile device A* object, C desired,
                           memory_order order) // All OS: Since Metal 1.
```

Metal 4.1 and later add support for `mem_flags` in `atomic_store_explicit`:

```
void atomic_store_explicit(threadgroup A* object, C desired,
                           memory_order order, mem_flags flags)

void atomic_store_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order, mem_flags flags)

void atomic_store_explicit(device A* object, C desired,
                           memory_order order, mem_flags flags)

void atomic_store_explicit(volatile device A* object, C desired,
                           memory_order order, mem_flags flags)
```

6.16.4.2 Atomic Load Functions

These functions atomically obtain the value pointed to by `object`. These functions support atomic types `A` of `atomic_int`, `atomic_uint`, `atomic_bool`, and `atomic_float`. Atomic load supports `atomic_float` only for device memory.

All OS: Support for the `atomic_load_explicit` function with `memory_order_relaxed` supported, as indicated.

```
C atomic_load_explicit(const threadgroup A* object,
                       memory_order order) // All OS: Since Metal 2.

C atomic_load_explicit(const volatile threadgroup A* object,
                       memory_order order) // All OS: Since Metal 1.

C atomic_load_explicit(const device A* object,
                       memory_order order) // All OS: Since Metal 2.

C atomic_load_explicit(const volatile device A* object,
                       memory_order order) // All OS: Since Metal 1.
```

Metal 4.1 and later add support for `mem_flags` in `atomic_load_explicit`:

```
C atomic_load_explicit(const threadgroup A* object,
                       memory_order order, mem_flags flags)

C atomic_load_explicit(const volatile threadgroup A* object,
                       memory_order order, mem_flags flags)

C atomic_load_explicit(const device A* object,
                       memory_order order, mem_flags flags)

C atomic_load_explicit(const volatile device A* object,
                       memory_order order, mem_flags flags)
```

6.16.4.3 Atomic Exchange Functions

These functions atomically replace the value pointed to by `object` with `desired` and return the value `object` previously held. These functions support atomic types `A` of `atomic_int`, `atomic_uint`, `atomic_bool`, and `atomic_float`.

All OS: Support for the `atomic_exchange_explicit` function with `memory_order_relaxed` supported, as indicated.

```
C atomic_exchange_explicit(threadgroup A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 2.

C atomic_exchange_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 1.

C atomic_exchange_explicit(device A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 2.

C atomic_exchange_explicit(volatile device A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 1.
```

Metal 4.1 and later add support for `mem_flags` in `atomic_exchange_explicit`:

```
C atomic_exchange_explicit(threadgroup A* object,
                           C desired,
                           memory_order order, mem_flags flags)

C atomic_exchange_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order, mem_flags flags)

C atomic_exchange_explicit(device A* object,
                           C desired,
                           memory_order order, mem_flags flags)

C atomic_exchange_explicit(volatile device A* object,
                           C desired,
                           memory_order order, mem_flags flags)
```

6.16.4.4 Atomic Compare and Exchange Functions

These compare-and-exchange functions atomically compare the value in `*object` with the value in `*expected`. If those values are equal, the compare-and-exchange function performs a read-modify-write operation to replace `*object` with `desired`. Otherwise if those values are not equal, the compare-and-exchange function loads the actual value from `*object` into `*expected`. If the underlying atomic value in `*object` was successfully changed, the compare-and-exchange function returns `true`; otherwise it returns `false`. These functions support atomic types `A` of `atomic_int`, `atomic_uint`, `atomic_bool`, and `atomic_float`.

Copying is performed in a manner like `std::memcpy`. The effect of a compare-and- exchange function is:

```
if (memcmp(object, expected, sizeof(*object)) == 0) {
    memcpy(object, &desired, sizeof(*object));
} else {
    memcpy(expected, object, sizeof(*object));
}
```

All OS: Support for the `atomic_compare_exchange_weak_explicit` function supported as indicated; support for `memory_order_relaxed` for indicating success and failure. If the comparison is `true`, the value of `success` affects memory access, and if the comparison is `false`, the value of `failure` affects memory access.

```
bool atomic_compare_exchange_weak_explicit(threadgroup A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure) // All OS: Since Metal 2.

bool atomic_compare_exchange_weak_explicit(
                    volatile threadgroup A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure)  // All OS: Since Metal 1.
bool atomic_compare_exchange_weak_explicit(device A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure)  // All OS: Since Metal 2.

bool atomic_compare_exchange_weak_explicit(
                    volatile device A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure)  // All OS: Since Metal 1.
```

Metal 4.1 and later add support for `mem_flags` in `atomic_compare_exchange_weak_explicit`:

```
bool atomic_compare_exchange_weak_explicit(threadgroup A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure, mem_flags flags)

bool atomic_compare_exchange_weak_explicit(
                    volatile threadgroup A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure, mem_flags flags)
bool atomic_compare_exchange_weak_explicit(device A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure, mem_flags flags)
```

```
bool atomic_compare_exchange_weak_explicit(
                    volatile device A* object,
                    thread C *expected, C desired,
                    memory_order success,
                    memory_order failure, mem_flags flags)
```

6.16.4.5 Atomic Fetch and Modify Functions

All OS: The following atomic fetch and modify functions are supported, as indicated.

The only supported value for `order` is `memory_order_relaxed`.

```
C atomic_fetch_key_explicit(threadgroup A* object,
                        M operand,
                        memory_order order) // All OS: Since Metal 2.

C atomic_fetch_key_explicit(volatile threadgroup A* object,
                        M operand,
                        memory_order order) // All OS: Since Metal 1.

C atomic_fetch_key_explicit(device A* object,
                        M operand,
                        memory_order order) // All OS: Since Metal 2.

C atomic_fetch_key_explicit(volatile device A* object,
                        M operand,
                        memory_order order) // All OS: Since Metal 1.
```

Metal 4.1 and later add support for `mem_flags` in `atomic_fetch_`*key*`_explicit`:

```
C atomic_fetch_key_explicit(threadgroup A* object,
                            M operand,
                            memory_order order, mem_flags flags)

C atomic_fetch_key_explicit(volatile threadgroup A* object,
                            M operand,
                            memory_order order, mem_flags flags)

C atomic_fetch_key_explicit(device A* object,
                            M operand,
                            memory_order order, mem_flags flags)

C atomic_fetch_key_explicit(volatile device A* object,
                            M operand,
                            memory_order order, mem_flags flags)
```

The `key` in the function name is a placeholder for an operation name listed in the first column of Table 6.27, such as `atomic_fetch_add_explicit`. The operations detailed in Table 6.27 are arithmetic and bitwise computations. The function atomically replaces the value pointed to by `object` with the result of the specified computation (third column of Table 6.27). The function returns the value that `object` held previously. There are no undefined results.

These functions are applicable to any atomic object of type `atomic_int`, and `atomic_uint`. Atomic `add` and `sub` support `atomic_float` only in `device` memory. Metal 4.1 and later add `atomic_float` support for atomic `add` and `sub` in `threadgroup` memory.

Table 6.27. Atomic operations

Key	Operator	Computation
`add`	`+`	Addition
`and`	`&`	Bitwise and
`max`	`max`	Compute max
`min`	`min`	Compute min
`or`	`|`	Bitwise inclusive or
`sub`	`-`	Subtraction
`xor`	`^`	Bitwise exclusive or

These operations are atomic read-modify-write operations. For signed integer types, the arithmetic operation uses two's complement representation with silent wrap-around on overflow.

6.16.4.6 Atomic Modify Functions (64 Bits)

All OS: Metal 2.4 and later support the following atomic modify functions for Apple silicon. See the Metal Feature Set Tables to determine which GPUs support this feature.

These functions are applicable to any atomic object of type `atomic_ulong`. The only supported value for `order` is `memory_order_relaxed`.

```
void atomic_key_explicit(device A* object,
                         M operand,
                         memory_order order)
void atomic_key_explicit(volatile device A* object,
                         M operand,
                         memory_order order)
```

The `key` in the function name is a placeholder for an operation name listed in the first column of Table 6.28, such as `atomic_max_explicit`. The operations detailed in Table 6.28 are arithmetic. The function atomically replaces the value pointed to by `object` with the result of the specified computation (third column of Table 6.28). The function returns void. There are no undefined results.

Table 6.28. Atomic modify operations

Key	Operator	Computation
`max`	`max`	Compute max

Key	Operator	Computation
`min`	`min`	Compute min

These operations are atomic read-modify-write operations.

6.17 Encoding Commands for Indirect Command Buffers

Indirect Command Buffers (ICBs) support the encoding of Metal commands into a Metal buffer for repeated use. Later, you can submit these encoded commands to the CPU or GPU for execution. ICBs for both render and compute commands use the `command_buffer` type to encode commands into an ICB object (represented in the Metal framework by `MTLIndirectCommandBuffer`):

```
struct command_buffer {
    size_t size() const;
};
```

An ICB can contain either render or compute commands but not both. Execution of compute commands from a render encoder is illegal. So is execution of render commands from a compute encoder.

6.17.1 Encoding Render Commands in Indirect Command Buffers

All OS: Metal 2.1 and later support indirect command buffers for render commands.

ICBs allow the encoding of draw commands into a Metal buffer for subsequent execution on the GPU.

In a shading language function, use the `command_buffer` type to encode commands for ICBs into a Metal buffer object that provides indexed access to a `render_command` structure.

```
struct arguments {
    command_buffer cmd_buffer;
};
kernel void producer(device arguments &args,
                     ushort cmd_idx [[thread_position_in_grid]])
{
    render_command cmd(args.cmd_buffer, cmd_idx);
    ...
}
```

`render_command` can encode any draw command type. The following public interface for `render_command` is defined in the header `<metal_command_buffer>`. To pass

`render_pipeline_state` objects to your shader, use argument buffers. Within an argument buffer, the pipeline state can be passed as scalars or in an array.

In iOS in Metal 2.2 and later, and macOS in Metal 2.1 and later, `set_render_pipeline_state(…)` and render pipeline states are available:

```
enum class primitive_type { point, line, line_strip, triangle,
                            triangle_strip };
```

Metal 4 defines the following structures and enumerations:

```
  enum class cull_mode { none, front, back };
  enum class depth_clip_mode { clip, clamp };
  enum class triangle_fill_mode { fill, lines };

  struct depth_stencil_state {
   public:
     depth_stencil_state();
     depth_stencil_state(const depth_stencil_state &);
     depth_stencil_state &operator=(const depth_stencil_state);
  };


struct render_command {
public:
    explicit render_command(command_buffer icb, unsigned cmd_index);
    void set_render_pipeline_state(
            render_pipeline_state pipeline_state);

    template <typename T …>
    void set_vertex_buffer(device T *buffer, uint index);
    template <typename T …>
    void set_vertex_buffer(constant T *buffer, uint index);

    // Metal 3.1: Supported passing vertex strides.
    template <typename T …>
    void set_vertex_buffer(device T *buffer, size_t stride,
                           uint index);
    template <typename T …>
    void set_vertex_buffer(constant T *buffer, size_t stride,
                           uint index);

    // Metal 4: Support setting raster states.
    void set_cull_mode(cull_mode mode);
    void set_front_facing_winding(winding w);
    void set_triangle_fill_mode(triangle_fill_mode mode);

    // Metal 4: Set depth stencil states.
```

```
void set_depth_bias(float bias, float slope_scale, float clamp);
void set_depth_clip_mode(depth_clip_mode mode);
void set_depth_stencil_state(depth_stencil_state state);

template <typename T …>
void set_fragment_buffer(device T *buffer, uint index);
template <typename T …>
void set_fragment_buffer(constant T *buffer, uint index);

void draw_primitives(primitive_type type, uint vertex_start,
                     uint vertex_count, uint instance_count,
                     uint base_instance);

// Overloaded draw_indexed_primitives based on index_buffer.
void draw_indexed_primitives(primitive_type type,
                             uint index_count,
                             device ushort *index_buffer,
                             uint instance_count,
                             uint base_vertex,
                             uint base_instance);

void draw_indexed_primitives(primitive_type type,
                             uint index_count,
                             device uint *index_buffer,
                             uint instance_count,
                             uint base_vertex,
                             uint base_instance);

void draw_indexed_primitives(primitive_type type,
                             uint index_count,
                             constant ushort *index_buffer,
                             uint instance_count,
                             uint base_vertex,
                             uint base_instance);

void draw_indexed_primitives(primitive_type type,
                             uint index_count,
                             constant uint *index_buffer,
                             uint instance_count,
                             uint base_vertex,
                             uint base_instance);



// Overloaded draw_patches based on patch_index_buffer and
// tessellation_factor_buffer.
void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  const device uint *patch_index_buffer,
```

```
                  uint instance_count, uint base_instance,
                  const device MTLQuadTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  const device uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  const device
                    MTLTriangleTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  const device uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  constant MTLQuadTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  const device uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  constant MTLTriangleTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  constant uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  const device MTLQuadTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  constant uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  const device
                    MTLTriangleTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
```

```
                  constant uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  constant MTLQuadTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);

void draw_patches(uint number_of_patch_control_points,
                  uint patch_start, uint patch_count,
                  constant uint *patch_index_buffer,
                  uint instance_count, uint base_instance,
                  constant MTLTriangleTessellationFactorsHalf
                      *tessellation_factor_buffer,
                  uint instance_stride = 0);



// Overloaded draw_indexed_patches based on patch_index_buffer,
// control_point_index_buffer and tessellation_factor_buffer.

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
```

```
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);


void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                const device uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLQuadTessellationFactorsHalf
```

```
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);


void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                const device void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                constant MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLQuadTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);

void draw_indexed_patches(uint number_of_patch_control_points,
                uint patch_start, uint patch_count,
                constant uint *patch_index_buffer,
                constant void *control_point_index_buffer,
                uint instance_count, uint base_instance,
                const device MTLTriangleTessellationFactorsHalf
                    *tessellation_factor_buffer,
                uint instance_stride = 0);
```

```
    void draw_indexed_patches(uint number_of_patch_control_points,
                              uint patch_start, uint patch_count,
                              constant uint *patch_index_buffer,
                              constant void *control_point_index_buffer,
                              uint instance_count, uint base_instance,
                              constant MTLQuadTessellationFactorsHalf
                                  *tessellation_factor_buffer,
                              uint instance_stride = 0);

    void draw_indexed_patches(uint number_of_patch_control_points,
                              uint patch_start, uint patch_count,
                              constant uint *patch_index_buffer,
                              constant void *control_point_index_buffer,
                              uint instance_count, uint base_instance,
                              constant MTLTriangleTessellationFactorsHalf
                                  *tessellation_factor_buffer,
                              uint instance_stride = 0);


    // Reset the entire command. After reset(), without further
    // modifications, execution of this command doesn't perform
    // any action.
    void reset();

    // Copy the content of the `source` command into this command.
    void copy_command(render_command source);
};
```

When accessing `command_buffer`, Metal does not check whether the access is within bounds. If an access is beyond the capacity of the buffer, the behavior is undefined.

The exposed methods in `render_command` mirror the interface of `MTLIndirectRenderCommand` and are similar to `MTLRenderCommandEncoder`. Notable differences with `MTLRenderCommandEncoder` are:

- Calls to `draw*` methods in `render_command` encode the actions taken by the command. If multiple calls are made, only the last one takes effect.
- The tessellation arguments are passed directly in `render_command::draw_patches` and `render_command::draw_indexed_patches`. Other calls do not set up the tessellation arguments.

6.17.2 Encoding Compute Commands in Indirect Command Buffers

iOS: Metal 2.2 and later support indirect command buffers for compute commands.
macOS: Metal 2.3 and later support indirect command buffers for compute commands.
iPadOS and visionOS: Metal supports indirect command buffers for compute commands.

ICBs allow the encoding of dispatch commands into a Metal buffer for subsequent execution on the GPU.

In a shading language function, use the `command_buffer` type to encode commands for ICBs into a Metal buffer object that provides indexed access to a `compute_command` structure:

```
struct arguments {
    command_buffer cmd_buffer;
};
[[kernel]] void producer(device arguments &args,
                         ushort cmd_idx [[thread_position_in_grid]])
{
    compute_command cmd(args.cmd_buffer, cmd_idx);
    ...
}
```

`compute_command` can encode any dispatch command type. The following public interface for `compute_command` is defined in the header `<metal_command_buffer>`. The `compute_pipeline_state` type represents compute pipeline states, which can only be passed to shaders through argument buffers. Within an argument buffer, the pipeline state can be passed as scalars or in an array.

```
struct compute_command {
public:
    explicit compute_command(command_buffer icb,
                             unsigned cmd_index);

    void set_compute_pipeline_state(
             compute_pipeline_state pipeline);

    template <typename T …>
    void set_kernel_buffer(device T *buffer, uint index);
    template <typename T …>
    void set_kernel_buffer(constant T *buffer, uint index);

    // Metal 3.1: Supports passing kernel strides.
    template <typename T …>
    void set_kernel_buffer(device T *buffer, size_t stride,
                           uint index);
    template <typename T …>
    void set_kernel_buffer(constant T *buffer, size_t stride,
                           uint index);


    void set_barrier();
    void clear_barrier();

    void concurrent_dispatch_threadgroups(
             uint3 threadgroups_per_grid,
             uint3 threads_per_threadgroup);
```

```
    void concurrent_dispatch_threads(uint3 threads_per_grid,
                                     uint3 threads_per_threadgroup);

    void set_threadgroup_memory_length(uint length, uint index);
    void set_stage_in_region(uint3 origin, uint3 size);

    // Reset the entire command. After reset(), without further
    // modifications. Execution of this command doesn't perform
    // any action.
    void reset();

    // Copy the content of the `source` command into this command.
    void copy_command(compute_command source);
};
```

When accessing `command_buffer`, Metal does not check whether the access is within bounds. If an access is beyond the capacity of the buffer, the behavior is undefined.

The exposed methods in `compute_command` mirror the interface of `MTLIndirectComputeCommand` and are similar to `MTLComputeCommandEncoder`.

In an ICB, dispatches are always concurrent. Calls to the `concurrent_dispatch*` methods in `compute_command` encode the actions taken by the command. If multiple calls are made, only the last one takes effect.

The application is responsible for putting barriers where they are needed. Barriers encoded in an ICB do not affect the parent encoder.

The CPU may have initialized individual commands within a `command_buffer` before the `command_buffer` is passed as an argument to a shader. If the CPU has not already initialized a command, you must reset that command before using it.

6.17.3 Copying Commands of an Indirect Command Buffer

Copying a `command` structure (either `render_command` or `compute_command`) via `operator=` does not copy the content of the command, it only makes the destination command point to the same buffer and index as the source command. To copy the content of the command, call the `copy_command` functions listed in sections 6.17.1 and 6.17.2.

Copying is only supported between commands pointing to compatible command buffers. Two command buffers are compatible only if they have matching ICB descriptors (`MTLIndirectCommandBufferDescriptor` objects). The commands themselves must also refer to valid indexes within the buffers. The following example illustrates using `copy_command` to copy the content of a render command from `cmd0` to `cmd1`:

```
struct arguments {
     command_buffer cmd_buffer;
     render_pipeline_state pipeline_state_0;
     render_pipeline_state pipeline_state_1;
};

[[kernel]] void producer(device arguments &args) {
```

```
    render_command cmd0(args.cmd_buffer, 0);
    render_command cmd1(args.cmd_buffer, 1);
    cmd0.set_render_pipeline_state(args.pipeline_state_0);

    // Make the command at index 1 point to command at index 0.
    cmd1 = cmd0;

    // Change the pipeline state for the command at index 0 in the
    // buffer.
    cmd1.set_render_pipeline_state(args.pipeline_state_0);

    // The command at index 1 in the buffer hasn't modified yet.
    cmd1 = render_command(args.cmd_buffer, 1);

    // Copy the content of the command at index 0 to command at
    // index 1.
    cmd1.copy_command(cmd0);
}
```

6.18 Variable Rasterization Rate

iOS: Metal 2.2 and later support variable rasterization rate and the rasterization rate map.
macOS: Metal 2.3 and later support variable rasterization rate and the rasterization rate map.
iPadOS and visionOS: Metal supports variable rasterization rate and the rasterization rate map.

Variable rasterization rate (VRR) can reduce the shading cost of high-resolution rendering by reducing the fragment shader invocation rate based on screen position. VRR is especially useful to avoid oversampling peripheral information in Augmented Reality (AR) / Virtual Reality (VR) applications.

To support VRR in a shading language function, use the `rasterization_rate_map_decoder` structure to describe the mapping of per-layer rasterization rate data. Each layer contains minimum quality values in screen space and can have a different physical fragment space dimension. For AR/VR, these quality values are based on the lens transform or eye-tracking information.

```
struct rasterization_rate_map_data;

struct rasterization_rate_map_decoder {
    explicit rasterization_rate_map_decoder(
                      constant rasterization_rate_map_data &data) thread;

    float2 map_screen_to_physical_coordinates(float2 screen_coordinates,
                           uint layer_index = 0) const thread;
    uint2 map_screen_to_physical_coordinates(uint2 screen_coordinates,
                           uint layer_index = 0) const thread;
    float2 map_physical_to_screen_coordinates(float2 physical_coordinates,
                           uint layer_index = 0) const thread;
    uint2 map_physical_to_screen_coordinates(uint2 physical_coordinates,
                           uint layer_index = 0) const thread;
};
```

The VRR map describes the mapping between screen space and physical fragment space and enables conversion of the rendering results back to the desired screen resolution. To convert between screen space and physical fragment space in the shader, the app must call the `copyParameterDataToBuffer:offset:` method of `MTLRasterizationRateMap` to fill the buffer with map data before using any of the conversion functions in the `rasterization_rate_map_decoder` structure. Passing anything other than a pointer to the data exported by the `copyParameterDataToBuffer:offset:` method has an undefined behavior.

The following example shows how the app must pass the `rasterization_rate_map_data` at the shader bind point to the constructor of the `rasterization_rate_map_decoder` structure:

```
[[fragment]] float4 fragment_shader(/* other arguments */
         constant rasterization_rate_map_data &data [[buffer(0)]]) {
    float2 screen_coords = ...;
    rasterization_rate_map_decoder map(data);
    float2 physical_coords =
         map.map_screen_to_physical_coordinates(screen_coords);
    ...
}
```

Alternately, the app can compute the offset where the compiled data is stored and use an explicit cast or pointer arithmetic to form the data for a valid `rasterization_rate_map_data`. Since `rasterization_rate_map_data` is an incomplete type, some operations on it are inherently forbidden (such as pointer arithmetic on the pointer type or `sizeof`).

6.19 Ray-Tracing Functions

All OS: Metal 2.3 and later support ray-tracing functions.

Metal defines the ray-tracing functions and types in `<metal_raytracing>` in the namespace `metal::raytracing`. Metal 2.3 and later supports them only in a compute function (kernel function), except where noted below. Metal 2.4 and later offer additional support for them in vertex, fragment, and tile functions.

6.19.1 Acceleration Structure Functions

In Metal 2.3 and later, you can call one of the following functions to check if an acceleration structure (see section 2.17.7) is `null`:

```
bool
is_null_primitive_acceleration_structure(primitive_acceleration_stru
cture)
```

```
bool
is_null_instance_acceleration_structure(instance_acceleration_struct
ure)
```

In Metal 2.4 and later, you can call the following function to check if an acceleration structure is `null`:

```
bool
is_null_acceleration_structure(acceleration_structure<intersection_t
ags…>)
```

In Metal 3.1 and later, you can iterate over the acceleration structure referenced by an instance acceleration structure using the following functions:

- Call the following function to query the number of instances in an instance acceleration structure:

  ```
  uint get_instance_count() const
  ```

- Call the following function to retrieve the acceleration structure referenced by an instance contained in an instance acceleration structure. The return type is the templatized type defined in section 2.17.7.

  ```
  template <typename... intersection_tags>
    acceleration_structure< intersection_tags...>
  get_acceleration_structure(uint instance_id)
  ```

If the declared return type does not match the acceleration structure type reference by the instance contained in an instance acceleration structure, then the results are undefined. Instance acceleration structures that do not use instance and/or primitive motion tags can be returned as an acceleration structure type that does contain those tags. For example, an instance acceleration structure without any motion (instance or primitive) can be returned as:

- `acceleration_structure<instancing>`
- `acceleration_structure<instancing, instance_motion>`
- `acceleration_structure<instancing, primitive_motion>`
- `acceleration_structure<instancing, primitive_motion, instance_motion>`

This capability allows you to avoid providing a dedicated intersector for each set of tags when working with multiple acceleration structure types at the potential performance cost due to traversing an acceleration structure that does not require those tags.

6.19.2 Intersector Intersect Functions

After creating the `intersector<intersection_tags...>` object (see section 2.17.6), you can call one of the following `intersect` functions based on the value of the `intersection_tags`:

```
result_type intersect(…parameters…)
```

Table 6.29 shows the possible `parameters` for `intersect` function. All `intersect` functions must have `ray` and `accel_struct` parameter. The other parameters are optional.

Table 6.29. Intersect functions input parameters

Parameter	Description
`ray`	Ray properties
`accel_struct`	Acceleration structure of `type acceleration_structure<intersection_tags...>`.
`mask`	Intersection mask to be AND'd with instance mask defined in the Metal API `MTLAccelerationStructureInstanceDescriptor`. Instances with nonoverlapping masks are skipped.
`time` All OS: Metal 2.4 and later.	The time associated with the ray. The parameter exists if the `intersection_tags` have `primitive_motion` or `instance_motion`.
`func_table`	Intersection function table of type `intersection_function_table<intersection_tags...>`. See section 2.17.3.
`payload`	User payload object, which is passed by reference. When the user calls `intersect()`, the payload parameter is copied to the `ray_data` address space and passed to the intersection function. The result is copied on the exit of the intersection function (section 5.1.6) and the payload object is updated.
`ifba` All OS: Metal 4 and later.	If the `intersection_tags` include `intersection_function_buffer`, you may optionally pass an object of type `intersection_function_buffer_arguments` (see section 6.19.8). The `ifba.intersection_function_buffer` must be uniform within the SIMD-group of the call.
`user_data` All OS: Metal 4 and later.	If the `intersection_tags` include `user_data`, you may optionally pass a buffer pointing to user data for the intersection function. If you pass a buffer, you also need to pass `ifba`.

The `result_type` is x

```
using result_type = intersection_result<intersection_tags...>;
```

The following set of intersect functions are available only if `intersection_tags` doesn't have `instancing`:

```
result_type
intersect(
  ray ray,
  primitive_acceleration_structure accel_struct) const;


result_type
intersect(
  ray ray,
  primitive_acceleration_structure accel_struct,
  intersection_function_table<intersection_tags...> func_table)
const;


template <typename T>
  result_type
  intersect(
    ray ray,
    primitive_acceleration_structure accel_struct,
    intersection_function_table<intersection_tags...> func_table,
    thread T &payload) const;
```

The following set of intersect functions are available only if `intersection_tags` has `instancing`:

```
 result_type
 intersect(
   ray ray,
   instance_acceleration_structure accel_struct,
   uint mask = ~0U) const;


 result_type
 intersect(
   ray ray,
   instance_acceleration_structure accel_struct,
   intersection_function_table<intersection_tags...> func_table)
 const;
```

The following set of intersect functions are available only if `intersection_tags` has `instancing` and don't have an `intersection_function_buffer`:

```
  template <typename T>
    result_type
    intersect(
      ray ray,
      instance_acceleration_structure accel_struct,
      intersection_function_table<intersection_tags...> func_table,
      thread T &payload) const;

  result_type
  intersect(
    ray ray,
    instance_acceleration_structure accel_struct,
    uint mask,
    intersection_function_table<intersection_tags...> func_table)
  const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      instance_acceleration_structure accel_struct,
      uint mask,
      intersection_function_table<intersection_tags...> func_table,
      thread T &payload) const;
```

In Metal 2.4 and later, the following set of intersect functions are available if `intersection_tags` have `primitive_motion` or `instance_motion`:

```
template <typename T, intersection_tags...>
   result_type
   intersect(
     ray ray,
     acceleration_structure< intersection_tags...> accel_struct,
     float time) const;
```

The following set of intersect functions are available only if `intersection_tags` has `instancing` and don't have an `intersection_function_buffer`:

```
template <typename T, intersection_tags...>
   result_type
```

```
   intersect(
     ray ray,
     acceleration_structure< intersection_tags...> accel_struct,
     float time,
     intersection_function_table<intersection_tags...> func_table)
    const;

template <typename T, intersection_tags...>
   result_type
   intersect(
     ray ray,
     acceleration_structure< intersection_tags...> accel_struct,
     float time,
     intersection_function_table<intersection_tags...> func_table,
     thread T &payload) const;
```

In Metal 2.4 and later, the following set of intersect functions are available only if `intersection_tags` have `instancing` and either `primitive_motion` or `instance_motion`:

```
template <typename T, intersection_tags...>
   result_type
   intersect(
     ray ray,
     acceleration_structure< intersection_tags...> accel_struct,
     uint mask = ~0U,
     float time = 0.0f) const;
```

The following set of intersect functions are available only if `intersection_tags` has `instancing`, and either `primitive_motion` or `instance_motion` don't have an `intersection_function_buffer`:

```
template <typename T, intersection_tags...>
   result_type
   intersect(
     ray ray,
     acceleration_structure< intersection_tags...> accel_struct,
     uint mask,
     float time,
     intersection_function_table<intersection_tags...> func_table)
   const;

template <typename T, intersection_tags...>
```

```
  result_type
  intersect(
    ray ray,
    acceleration_structure< intersection_tags...> accel_struct,
    uint mask,
    float time,
    intersection_function_table<intersection_tags...> func_table,
    thread T &payload) const;
```

In Metal 3.2 and later, it's possible to avoid a copy and directly access the memory of the intersection by using `intersection_result_ref<intersection_tags...>` (see section 2.17.5) and the `ray_data` payload pointer in a callback:

```
template <typename Callable>
void intersect(..., Callable callback)

template <typename Payload, typename Callable>
void intersect(..., const thread Payload &payload_in,
               Callable callback)
```

The lifetime is the `intersection_result_ref` and the `ray_data` payload pointer is the duration of the callback. If you store the `intersection_result_ref` or payload pointer and use it after the `intersect()` call completes, the behavior is undefined because the system may free the memory. You can't perform recursive ray tracing within the callback body. After the callback exits, the shader is free to intersect rays again.

The following is an example of the use of a lambda with the `intersection_result_ref`:

```
[[kernel]] void trace_rays_with_payload(...) {
    intersector<instancing, max_levels<2>, triangle_data> i;
    i.intersect(ray, acceleration_structure, MyPayload{},
        [&](intersection_result_ref<instancing, max_levels<2>,
triangle_data> result,
            const ray_data MyPayload &final_payload)
    {
        result.get_primitive_id();
        // ...
    });
}
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer` and doesn't have `instancing`:

```
  result_type
  intersect(
```

```
    ray ray,
    acceleration_structure<> accel_struct,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      acceleration_structure<> accel_struct,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer` and `instancing`:

```
  result_type
  intersect(
    ray ray,
    acceleration_structure<instancing> accel_struct,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      acceleration_structure<instancing> accel_struct,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;

  result_type
  intersect(
    ray ray,
    uint mask,
    acceleration_structure<instancing> accel_struct,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      uint mask,
      acceleration_structure<instancing> accel_struct,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `instancing`, and `primitive_motion`.

```
  result_type
  intersect(
    ray ray,
    acceleration_structure<instancing, primitive_motion> as,
    float time,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      acceleration_structure<instancing, primitive_motion> as,
      float time,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;

  result_type
  intersect(
    ray ray,
    uint mask,
    float time,
    acceleration_structure<instancing, primitive_motion> as,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      uint mask,
      float time,
      acceleration_structure<instancing, primitive_motion> as,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `instancing`, and `instance_motion`:

```
  result_type
  intersect(
```

```
    ray ray,
    acceleration_structure<instancing, instance_motion> as,
    float time,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      acceleration_structure<instancing, instance_motion> as,
      float time,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;

  result_type
  intersect(
    ray ray,
    uint mask,
    float time,
    acceleration_structure<instancing, instance_motion> as,
    intersection_function_buffer_arguments ifba) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      uint mask,
      float time,
      acceleration_structure<instancing, instance_motion> as,
      intersection_function_buffer_arguments ifba,
      thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `user_data`, and doesn't have `instancing`:

```
  result_type
  intersect(
    ray ray,
    acceleration_structure<> accel_struct,
    intersection_function_buffer_arguments ifba,
    const device void *user_data) const;

  template <typename T>
```

```
  result_type
  intersect(
    ray ray,
    acceleration_structure<> accel_struct,
    intersection_function_buffer_arguments ifba,
    const device void *user_data,
    thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `user_data,` and `instancing`:

```
  result_type
  intersect(
    ray ray,
    acceleration_structure<instancing> accel_struct,
    intersection_function_buffer_arguments ifba,
    const device void *user_data) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      acceleration_structure<instancing> accel_struct,
      intersection_function_buffer_arguments ifba,
      const device void *user_data,
      thread T &payload) const;

  result_type
  intersect(
    ray ray,
    uint mask,
    acceleration_structure<instancing> accel_struct,
    intersection_function_buffer_arguments ifba,
    const device void *user_data) const;

  template <typename T>
    result_type
    intersect(
      ray ray,
      uint mask,
      acceleration_structure<instancing> accel_struct,
      intersection_function_buffer_arguments ifba,
      const device void *user_data,
      thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `user_data`, `instancing`, and `primitive_motion`:

```
result_type
intersect(
  ray ray,
  acceleration_structure<instancing, primitive_motion> as,
  float time,
  intersection_function_buffer_arguments ifba,
  const device void *user_data) const;

template <typename T>
  result_type
  intersect(
    ray ray,
    acceleration_structure<instancing, primitive_motion> as,
    float time,
    intersection_function_buffer_arguments ifba,
    const device void *user_data,
    thread T &payload) const;

result_type
intersect(
  ray ray,
  uint mask,
  float time,
  acceleration_structure<instancing, primitive_motion> as,
  intersection_function_buffer_arguments ifba,
  const device void *user_data) const;

template <typename T>
  result_type
  intersect(
    ray ray,
    uint mask,
    float time,
    acceleration_structure<instancing, primitive_motion> as,
    intersection_function_buffer_arguments ifba,
    const device void *user_data,
    thread T &payload) const;
```

In Metal 4 and later, the following set of intersect functions are available only if `intersection_tags` has an `intersection_function_buffer`, `instancing`, `user_data`, and `instance_motion`:

```
result_type
intersect(
  ray ray,
  acceleration_structure<instancing, instance_motion> as,
  float time,
  intersection_function_buffer_arguments ifba,
  const device void *user_data) const;



template <typename T>
  result_type
  intersect(
    ray ray,
    acceleration_structure<instancing, instance_motion> as,
    float time,
    intersection_function_buffer_arguments ifba,
    const device void *user_data,
    thread T &payload) const;

result_type
intersect(
  ray ray,
  uint mask,
  float time,
  acceleration_structure<instancing, instance_motion> as,
  intersection_function_buffer_arguments ifba,
  const device void *user_data) const;

template <typename T>
  result_type
  intersect(
    ray ray,
    uint mask,
    float time,
    acceleration_structure<instancing, instance_motion> as,
    intersection_function_buffer_arguments ifba,
    const device void *user_data,
    thread T &payload) const;
```

6.19.3 Intersector Functions to Control Traversal Behavior

All OS: Metal 3.1 adds support for curves.

To override the default behavior of the traversal, you can use the following member functions of `intersector<intersection_tags...>` object.

```
  void set_triangle_front_facing_winding(winding);
  void set_geometry_cull_mode(geometry_cull_mode);
  void set_opacity_cull_mode(opacity_cull_mode);
  void force_opacity(forced_opacity);
  void assume_geometry_type(geometry_type);
  void assume_identity_transforms(bool);
  void accept_any_intersection(bool);
```

Triangles have two sides or "faces". The front facing winding determines which triangle face is considered the "front" face when viewed from the ray origin. If the vertices appear in clockwise order when viewed from the ray origin and the front facing winding is clockwise, then the visible face is the front face. The other face is the back face. If the front facing winding is counterclockwise, then the opposite is true. Use the following function to change the default winding (`clockwise`):

```
enum class winding {
  clockwise,
  counterclockwise
};
void set_triangle_front_facing_winding(winding w);
```

To change the default triangle cull mode (`none`), use the following function:

```
enum class triangle_cull_mode {
  none,
  front,
  back
};
void set_triangle_cull_mode(triangle_cull_mode tcm);
```

If the cull mode is set to `front`, then triangles whose front face is visible from the ray `origin` are not considered for intersection. Otherwise, if the cull mode is set to `back`, then triangles whose back face is visible from the ray origin are not considered for intersection.

The following function may be used to set the intersector to cull all bounding box or triangle primitives from the set of candidate geometries. The default geometry cull mode is `none`.

```
enum class geometry_cull_mode {
  none,
  triangle,
  bounding_box,
  curve // Metal 3.1 and later.
};

void set_geometry_cull_mode(geometry_cull_mode gcm);
```

The default opacity cull mode is `none`. Use the following function to change the opacity. See below on how opacity affects triangle and bounding box primitives.

```
enum class opacity_cull_mode {
  none,
  opaque,
  non_opaque
};
void set_opacity_cull_mode(opacity_cull_mode ocm);
```

Call the following function to override per-instance and per-geometry setting of forced capacity. The default is `none`.

```
enum class forced_opacity {
  none,
  opaque,
  non_opaque
};
void force_opacity(forced_opacity fo);
```

Triangle primitives may also be culled based on their opacity: An `opaque` triangle will not run any intersection function. A `non_opaque` triangle runs its intersection function to accept or reject the hit.

The `PrimitiveAccelerationStructure` encodes if the triangle is `opaque` or `non_opaque` by declaring `MTLAccelerationStructureGeometryFlagOpaque`. The opaqueness can be overridden by calling `intersector.force_opacity()`. If used, this takes precedence over the per-instance opaqueness flags (`MTLAccelerationStructureInstanceFlagOpaque` and `MTLAccelerationStructureInstanceFlagNonOpaque`), which in turn takes precedence over the per-geometry opaqueness.

For custom bounding box primitives, the opaqueness will be evaluated in the same way as described for triangles (first `intersector.set_opacity_cull_mode()`, then `InstanceFlags`, then `GeometryFlags`). The `opaque` parameter informs the bounding box

intersection program the resolved opaqueness state. The intersection function may then use this to influence its evaluation of if a hit is encountered or not.

`intersector.set_opacity_cull_mode()` skips over primitive types based on their opaqueness.

If `intersector.force_opacity()` is set to `opaque` or `non_opaque`, then `intersector.set_opacity_cull_mode()` must be `none`. The reverse is also true: Opacity Override and Opacity culling cannot be mixed. The results of illegal combinations are undefined.

Use the following functions to declare if the acceleration structure contains a triangle, bounding box, and/or curve geometry. The default geometry is `geometry_type::triangle | geometry_type::bounding_box`. By default, Metal assumes acceleration structure will not contain curve geometry to improve performance. Call `assume_geometry_type` with a value that includes `geometry_type::curve` to enable curves to be intersected in an intersect call or intersection query step.

```
enum class geometry_type {
  none,
  triangle,
  bounding_box,
  curve, // Metal 3.1 and later.
  all
};
void assume_geometry_type(geometry_type gt)
```

To set the intersector object to assume identify transforms, call the following function with the value `true`. The default is `false`.

```
void assume_identity_transforms(bool value);
```

To set the intersector object to immediately return the first intersection it finds, call the following function with the value `true`. The default is `false`. One use of this function is when you only need to know if one point is visible from another, such as when rendering shadows or ambient occlusion.

```
void accept_any_intersection(bool value);
```

In Metal 3.1 and later, use the following functions to add hints to the `intersector` and `intersection_query` to specify the curve basis, the number of control points, and the curve type to optimize traversal for specific curve types:

Note that `curve_basis` is an enumerated type and not a bitmask.

```
enum class curve_basis {
  bspline,
  catmull_rom,
```

```
  linear,
  bezier,
  all,
};

enum class curve_type {
  round,
  flat,
  all,
};
```

Use the following function to set the curve basis function to assume. Defaults to `curve_basis::all`, meaning that all curve basis functions will be enabled.

```
void assume_curve_basis(curve_basis cb)
```

Use the following function to set the curve type to assume. Defaults to `curve_type::all`, meaning that both curve types will be enabled.

```
void assume_curve_type(curve_type ct)
```

Use the following function to set the number of curve control points to assume. Defaults to 0, meaning that any number of control points, as appropriate for the assumed curve basis (if any), will be enabled. Other valid options are 2, 3, or 4, depending on the curve basis.

```
void assume_curve_control_point_count(uint n)
```

6.19.4 Intersector Functions for Ray Contribution and Geometry Multiplier

All OS: Metal 4 adds support to specify Ray Contribution and Geometry Multiplier.

In Metal 4 and later, you can specify the ray contribution and geometry multiplier by adding state per intersector object if if `intersection_tags` has `intersection_function_buffer`. Note the calculation of base index and geometry multiplier use the lower 4 bits.

Call the following function to set the base ID. The default value of the base ID is `0`.

```
  void set_base_id(uint index);
```

Call the following function to set the geometry multiplier on the intersector. The default value of multiplier is 1.

```
  void set_geometry_multiplier(uint multiplier);
```

6.19.5 Intersection Query Functions

All OS: Metal 2.4 and later support intersection query functions.

All OS: Metal 3.1 and later support intersection query functions for curves.

To start traversals and query traversal specific information, create an intersection query object (see section 2.17.8) with a nondefault constructor or first call `reset(…)`. If not called in this sequence, the behavior is undefined.

Table 6.30, Table 6.32, and Table 6.33 show the list of functions that can be called depending on the geometry type encountered during the traversal, assuming `next()` has returned `true`. Note that some functions come in pairs: a candidate and a committed primitive. When `next()` is called for the first time, the primitive reported after the traversal is always a candidate until the user commits the primitive by calling `commit_triangle_intersection()`, `commit_bounding_box_intersection()`, or `commit_curve_intersection()` on the query object. Note that opaque triangles, tested without user intersection, commit automatically when intersected.

Table 6.30. Intersection query functions

Functions	Triangle	Bounding	Curve
`void reset(…)`	✅	✅	✅
`bool next()`	✅	✅	✅
`void abort()`	✅	✅	✅
`intersection_type get_candidate_intersection_type()`	✅	✅	✅
`intersection_type get_committed_intersection_type()`	✅	✅	✅
`void commit_triangle_intersection()`	✅		
`void commit_bounding_box_intersection(float distance)`		✅	
`void commit_curve_intersection()` All OS: Metal 3.1 and later.			✅

Table 6.31. Intersection query functions with `max_levels<Count>`

Functions	Triangle	Bounding	Curve
`uint get_candidate_instance_count()` All OS: Metal 3.1 and later.	✅	✅	✅
`uint get_candidate_instance_id(uint depth)` All OS: Metal 3.1 and later.	✅	✅	✅
`uint get_candidate_user_instance_id(uint depth)` All OS: Metal 3.1 and later.	✅	✅	✅
`uint get_committed_instance_count()` All OS: Metal 3.1 and later.	✅	✅	✅
`uint get_committed_instance_id(uint depth)` All OS: Metal 3.1 and later.	✅	✅	✅
`uint get_committed_user_instance_id(uint depth)` All OS: Metal 3.1 and later.	✅	✅	✅

Table 6.32. Intersection query ray value functions

Ray values functions	Triangle	Bounding	Curve
`float3 get_world_space_ray_origin()`	✅	✅	✅
`float3 get_world_space_ray_direction()`	✅	✅	✅
`float get_ray_min_distance()`	✅	✅	✅
`intersection_params get_intersection_params()`	✅	✅	✅

Table 6.33. Intersection query candidate value functions

Candidate intersections value functions	Triangle	Bounding	Curve
`float get_candidate_triangle_distance()`	✅		
`uint get_candidate_instance_id()`	✅	✅	✅
`uint get_candidate_user_instance_id()`	✅	✅	✅
`uint get_candidate_geometry_id()`	✅	✅	✅
`uint get_candidate_primitive_id()`	✅	✅	✅
`float2 get_candidate_triangle_barycentric_coord()`	✅		
`bool is_candidate_non_opaque_bounding_box()`		✅	
`bool is_candidate_triangle_front_facing()`	✅		
`float4x3 get_candidate_object_to_world_transform()`	✅	✅	✅
`float4x3 get_candidate_world_to_object_transform()`	✅	✅	✅
`float3 get_candidate_ray_origin()`	✅	✅	✅
`float3 get_candidate_ray_direction()`	✅	✅	✅
`const device void * get_candidate_primitive_data()` All OS: Metal 3 and later.	✅	✅	✅

Table 6.34. Intersect query committed value functions

Committed intersections value functions	Triangle	Bounding	Curve
`float get_committed_distance()`	✅	✅	✅
`uint get_committed_instance_id()`	✅	✅	✅
`uint get_committed_user_instance_id()`	✅	✅	✅
`uint get_committed_geometry_id()`	✅	✅	✅
`uint get_committed_primitive_id()`	✅	✅	✅
`float2 get_committed_triangle_barycentric_coord()`	✅		
`bool is_committed_triangle_front_facing()`	✅		
`float4x3 get_committed_object_to_world_transform()`	✅	✅	✅
`float4x3 get_committed_world_to_object_transform()`	✅	✅	✅
`float3 get_committed_ray_origin()`	✅	✅	✅
`float3 get_committed_ray_direction()`	✅	✅	✅
`const device void * get_committed_primitive_data()` All OS: Metal 3 and later.	✅	✅	✅
`float get_candidate_curve_parameter()` All OS: Metal 3.1 and later.			✅
`float get_committed_curve_parameter()` All OS: Metal 3.1 and later.			✅

In Metal 3.1 and later, intersection query supports the following functions when specified with the `max_levels<Count>` intersection tags:

- Call the following function to query the distance of a candidate triangle hit that needs consideration:

```
float get_candidate_triangle_distance();
```

- Call the following function to query the distance of the currently committed hit:

```
float get_committed_distance();
```

- Call the following function to query the top-level structure instance ID for the current candidate hit:

```
uint get_candidate_instance_id();
```

- Call the following function to query user instance ID provided by user on the bottom level acceleration structure for the current candidate hit:

```
uint get_candidate_user_instance_id();
```

- Call the following function to query the bottom-level structure geometry ID for the current candidate hit:

```
uint get_candidate_geometry_id();
```

- Call the following function to query the bottom-level structure primitive ID within the geometry for the current candidate hit:

```
uint get_candidate_primitive_id();
```

- Call the following function to query the top-level structure instance ID for the current committed hit:

```
uint get_committed_instance_id();
```

- Call the following function to query user instance ID provided by user on the bottom level acceleration structure for the current committed hit:

```
uint get_committed_user_instance_id();
```

- Call the following function to query the bottom-level structure geometry ID for the current committed hit:

```
uint get_committed_geometry_id();
```

- Call the following function to query the bottom-level structure primitive ID within the geometry for the current committed hit:

```
uint get_committed_primitive_id();
```

- Call the following function to query the ray origin in object space for the current hit candidate:

```
float3 get_candidate_ray_origin();
```

- Call the following function to query the ray direction in object space for the current hit candidate:

```
float3 get_candidate_ray_direction();
```

- Call the following function to query the ray origin in object space for the current committed hit:

```
float3 get_committed_ray_origin();
```

- Call the following function to query the ray direction in object space for the current committed hit:

```
float3 get_committed_ray_direction();
```

- Call the following function to query the matrix for transforming ray origin/direction of current hit candidate from object-space to world-space:

```
float4x3 get_candidate_object_to_world_transform();
```

- Call the following function to query the matrix for transforming ray origin/direction of current candidate hit from world-space to object-space:

```
float4x3 get_candidate_world_to_object_transform();
```

- Call the following function to query the matrix for transforming ray origin/direction of current committed hit from object-space to world-space:

```
float4x3 get_committed_object_to_world_transform();
```

- Call the following function to query the matrix for transforming ray origin/direction of current committed hit from world-space to object-space:

```
float4x3 get_committed_world_to_object_transform();
```

- Call the following function to query the candidate hit location barycentric coordinates. Valid when `get_candidate_intersection_type()` returns `triangle`:

```
float2 get_candidate_triangle_barycentric_coord();
```

- For vertex attributes `v0`, `v1`, and `v2`, the value at the specified barycentric point is:

```
v1 * barycentric_coord.x +
v2 * barycentric_coord.y +
v0 * (1.0f - (barycentric_coord.x + barycentric_coord.y))
```

- Call the following function to query the committed hit location barycentric coordinates. Valid when `get_committed_intersection_type()` returns `triangle`:

```
float2 get_committed_triangle_barycentric_coord();
```

- Call the following function to query if the hit triangle candidate is front or back facing. Returns `true` if it is front face and `false` if it is back face. Valid when `get_candidate_intersection_type()` returns `triangle`:

```
bool is_candidate_triangle_front_facing();
```

- Call the following function to query if the committed hit is front or back facing. Returns `true` if it is front face and `false` if it is back face. Valid when `get_committed_intersection_type()` returns `triangle`:

```
bool is_committed_triangle_front_facing();
```

- Call the following function to query the per-primitive data for the current candidate primitive:

```
const device void *get_candidate_primitive_data();
```

- Call the following function to query the per-primitive data for the current committed hit:

```
const device void *get_committed_primitive_data();
```

In Metal 3.1 and later, the following two functions can be called when `get_candidate_intersection_type()` returns `curve` and the intersection tag has `curve_data`:

- Call the following to query the curve parameter for the current candidate curve:

```
float get_candidate_curve_parameter();
```

- Call the following to query the curve parameter for the current committed intersection. Valid when `get_candidate_intersection_type()` returns `curve`.

```
float get_committed_curve_parameter();
```

In Metal 3.1 and later, the rest of the functions in this section can be called when the intersection tag has `max_levels<Count>`:

- Call the following function to query the number of instances in the candidate intersection:

```
uint get_candidate_instance_count();
```

- Call the following function to query the instance ID at level `depth` in the candidate intersection.

```
uint get_candidate_instance_id(uint depth);
```

- Call the following function to query the user instance ID at level `depth` in the candidate intersection:

```
uint get_candidate_user_instance_id(uint depth);
```

- Call the following function to query the number of instances in the committed intersection:

```
uint get_committed_instance_count();
```

- Call the following function to query the instance ID at level `depth` in the committed intersection:

  ```
  uint get_committed_instance_id(uint depth);
  ```

- Call the following function to query the user instance ID at level `depth` in the committed intersection:

  ```
  uint get_committed_user_instance_id(uint depth);
  ```

6.19.6 Indirect Instance Descriptors

In Metal 3.1 and later, fill out indirect instance descriptors from the GPU. Metal provides the following type definitions:

```
enum MTLAccelerationStructureInstanceOptions : uint
{
  MTLAccelerationStructureInstanceOptionNone = 0,
  MTLAccelerationStructureInstanceOptionDisableTriangleCulling =
    (1 << 0),
MTLAccelerationStructureInstanceOptionTriangleFrontFacingWindingCoun
terClockwise = (1 << 1),
  MTLAccelerationStructureInstanceOptionOpaque = (1 << 2),
  MTLAccelerationStructureInstanceOptionNonOpaque = (1 << 3),
};

typedef packed_float3 MTLPackedFloat3;
typedef packed_float3 MTLPackedFloat4x3[4];

struct MTLAccelerationStructureInstanceDescriptor
{
  MTLPackedFloat4x3 transformationMatrix;
  MTLAccelerationStructureInstanceOptions options;
  uint mask;
  uint intersectionFunctionTableOffset;
  uint accelerationStructureIndex;
};

struct MTLAccelerationStructureUserIDInstanceDescriptor
{
  MTLPackedFloat4x3 transformationMatrix;
  MTLAccelerationStructureInstanceOptions options;
  uint mask;
  uint intersectionFunctionTableOffset;
  uint accelerationStructureIndex;
  uint userID;
};
```

To facilitate filing out the descriptor, Metal provides an implicit conversion from `acceleration_structure<intersection_tags…>` to `MTLResourceID`:

```
acceleration_structure<primitive_motion> primitiveAStruct = …;
MTLResourceID resource_id = primitiveAStruct;
```

6.19.7 Curve Utility Functions

Metal 3.1 and later provide a set of curve utility functions that Metal defines in the header `<metal_curves>`. It uses the following abbreviations:

`Ps` is `float` or `half`.

`P` is a scalar or a vector of `Ps`. If Ps is `float`, `P` is `float4`.

The functions return the position or the first or second derivative on a curve given a curve parameter `t`, and control points `p0`, `p1`, etc. As shown in Table 6.35, the functions support quadratic Bézier, cubic Bézier, quadratic B-Spline, cubic B-Spline, cubic Hermite, and Catmull-Rom curves.

Table 6.35. Curve utility functions

Function	Description
`P bezier(` `  Ps_t, P p0, P p1, P p2)`	Returns the position on a quadratic Bézier curve
`P bezier_derivative(` `  Ps_t, P p0, P p1, P p2)`	Returns the first derivative on a quadratic Bézier curve
`P bezier_second_derivative(` `  Ps_t, P p0, P p1, P p2)`	Returns the second derivative on a quadratic Bézier curve
`P bezier(` `  Ps_t, P p0, P p1, P p2, P p3)`	Returns the position on a cubic Bézier curve
`P bezier_derivative(` `  Ps_t, P p0, P p1, P p2, P p3)`	Returns the first derivative on a cubic Bézier curve
`P bezier_second_derivative(` `  Ps_t, P p0, P p1, P p2, P p3)`	Returns the second derivative on a cubic Bézier curve
`P bspline(` `  Ps_t, P p0, P p1, P p2)`	Returns the position on a quadratic B-spline curve
`P bspline_derivative(` `  Ps_t, P p0, P p1, P p2)`	Returns the first derivative on a quadratic B-spline curve
`P bspline_second_derivative(` `  Ps_t, P p0, P p1, P p2)`	Returns the second derivative on a quadratic B-spline curve
`P bspline(` `  Ps_t, P p0, P p1, P p2, P p3)`	Returns the position on a cubic B-spline curve

Function	Description
`P bspline_derivative( Ps_t, P p0, P p1, P p2, P p3)`	Returns the first derivative on a cubic B-spline curve
`P bspline_second_derivative( Ps_t, P p0, P p1, P p2, P p3)`	Returns the second derivative on a cubic B-spline curve
`P hermite( Ps_t, P p0, P p1, P m0, P m1)`	Returns the position on a cubic Hermite curve
`P hermite_derivative( Ps_t, P p0, P p1, P m0, P m1)`	Returns the first derivative on a cubic Hermite curve
`P hermite_second_derivative( Ps_t, P p0, P p1, P m0, P m1)`	Returns the second derivative on a cubic Hermite curve
`P catmull_rom( Ps_t, P p0, P p1, P p2, P p3)`	Returns the position on a Catmull-Rom curve
`P catmull_rom_derivative( Ps_t, P p0, P p1, P p2, P p3)`	Returns the first derivative on a Catmull-Rom curve
`P catmull_rom_second_derivative( Ps_t, P p0, P p1, P p2, P p3)`	Returns the second derivative on a Catmull-Rom curve

6.19.8 Intersection Function Buffer Descriptors

In Metal 4 and later, you can use indirect function buffers to associate geometry in a scene with a set of shaders that operate on that geometry in the acceleration structure. The user provides a buffer containing `intersection_function_buffer_arguments`.

```
struct intersection_function_buffer_arguments
{
  // Buffer containing instruction function handles aligned
  // to 8 bytes.
  const device void * intersection_function_buffer;

  // Maximum range in bytes
  size_t intersection_function_buffer_size;

  // The stride between intersection function entries.
  size_t intersection_function_stride;
};
```

The stride, `intersection_function_stride`, support ranges from `[0. 4096]` in 8 bytes increments.

For convenience, the header provides the Metal `MTLIntersectionFunctionBufferArguments` which is convertible to `intersection_function_buffer_arguments`.

The example above passes a buffer to intersect (see section 6.19.2).

6.20 Logging Functions

All OS: Metal 3.2 and later support logging for Apple silicon.

Metal defines the logging functions and types in `<metal_logging>`. To enable logging, you need to set `-fmetal-enable-logging` (see section 1.6.9).

```
enum log_type
{
  // Captures verbose information useful only for
  // debugging your code.
  log_type_debug,
  // Captures information that is helpful to troubleshoot problems.
  log_type_info,
  // Captures information that is essential for
  // troubleshooting problems.
  log_type_default,
  // Captures errors that occur during the execution of your code.
  log_type_error,
  // Captures information about faults and bugs in your code.
  log_type_fault
};
```

```
struct os_log
{
  os_log(constant char *subsystem, constant char *category)
constant;
  void log_with_type(log_type type, constant char *format, ...)
constant;
  void log_debug(constant char *format, ...) constant;
  void log_info(constant char *format, ...) constant;
  void log(constant char *format, ...) constant;
  void log_error(constant char *format, ...) constant;
  void log_fault(constant char *format, ...) constant;
};
```

The `os_log` logging methods support most of the format specifiers that `std::printf` supports in C++, with the following exceptions:

- They don't support the `%n` and `%s` conversion specifiers.
- They don't support the `%@` and `%.*P` and custom format specifiers that the CPU `os_log` supports.
- Metal supports the `hl` length modifier for 4-byte types like `int` and `float`, which you need to use when printing vectors.
- Vectors may print with `%v[num_elements][length_modifier][conversion_specifier]`. For example, a `float4` can print with `%v4hlf` while a `uchar2` can print as `%v2hhu`.
- Default argument promotion applies to arguments of half type which promote to the double type. Default argument promotion doesn't apply to vectors.
- The format string must be a string literal.

Shaders can perform logging by defining an `os_log` object and using any of the log member functions:

```
constant metal::os_log custom_log("com.custom_log.subsystem",
                                  "custom category");
void test_log(float x) {
  if (x < M_PI_F)
    custom_log.log("custom message %f", x);
}
```

A default `os_log` object `os_log_default` is available to use instead of a custom `os_log` object:

```
void test_log(float x) {
  if (x < M_PI_F)
    os_log_default.log("custom message %f", x);
}
```

Metal places messages from the shader into a log buffer with a size that `MTLLogState` determines. All the draw/dispatches in a command buffer share the log buffer. The system only removes the messages from the log buffer when the command buffer completes. Because multiple command buffers can share a log buffer, the system may block the removal of the messages until other command buffers complete. When the log buffer becomes full, the system drops all subsequent messages. Logging resumes after the CPU has an opportunity to empty the log buffer.

By default, messages that the CPU reads from the log buffer go into the unified logging system with the corresponding subsystem, category, and level. Messages that `os_log_default` logs go into the CPU unified logging system with the corresponding level and subsystem/category being nil. For custom handling of shader logging messages, see the Metal API's `addLogHandler`.

7 Metal Performance Primitives

All OS: Metal 4 and later support Metal Performance Primitives.

Metal Performance Primitives is a library of optimized primitives that are designed to be efficient, portable, and performant on Apple silicon. The header `<MetalPerformancePrimitives/ MetalPerformancePrimitives.h>` defines these functions within the namespace `mpp`. The `tensor_ops` namespace, which resides beneath the `mpp` namespace, contains functions that operate on tensors, including matrix multiplication and convolution. The functions that operate on tensor, Tensor Operations (TensorOps), use `tensor` and `cooperative_tensors` (see section 2.22) and have been tuned for Apple silicon GPUs. For a list of supported GPU families, refer to the Metal Feature Set Tables at developer.apple.com. When instantiating a TensorOp, you pass the scope of execution for the operation, where scope is the number of threads cooperating to execute the operation (see section 7.1). Refer to Metal Performance Primitives (MPP) Programming Guide for performance optimization tips and best practices.

7.1 Execution Scopes

All OS: Metal 4 and later support execution scopes.

Operations like TensorOps can work on a single thread, or cooperatively across threads in a SIMD-group or multiple SIMD-groups. You use execution scopes to specify the scope of cooperation. Table 7.1 outlines the types of execution scope.

Table 7.1. Execution scopes

Scope	Description
`execution_thread`	Indicates the scope of cooperation is a single thread

`execution_simdgroups<N>`
or
`execution_simdgroup for N==1`
	Indicates the scope of cooperation is `N` SIMD-groups. TensorOp support `N` with a value of `1` or `simdgroups_per_threadgroup` (see section 5.2.3.6)
	
	You can use `execution_simdgroup` for `N = 1`.

7.2 Tensor Operations (TensorOps)

All OS: Metal 4 and later support tensor operations (TensorOps).

TensorOps are GPU-accelerated functions that operate on `tensors` and `cooperative_tensors` (see section 2.22). TensorOps are class templates that you instantiate with a set of properties, including the execution scope, to indicate if the operation should run on a single thread or cooperatively across threads in a SIMD-group or multiple SIMD-groups (see section 7.1). When calling the TensorOp `run` method, all threads must call the method within that scope, or the result is undefined. For example, if the scope used to create the TensorOp is `execution_simdgroup`, you must ensure all threads within the same SIMD-group call the `run` method. Note that different SIMD-groups can be divergent with each other in this case.

TensorOps may use a barrier at the level of the execution scope. For example, if you specify the scope of an operation to be the entire threadgroup, you should ensure your code would behave correctly if a barrier is used in the TensorOp implementation.

If the TensorOps writes the result into a tensor whose `ElementType` is in `device` or `threadgroup` address space, you must insert a barrier (see section 6.10.1) at the appropriate thread scope and set the appropriate memory flags before reading the results. You don't need to use a barrier for tensors whose memory is in thread address space or for `cooperative_tensors`. For example, if the TensorOp `run` method writes to a tensor whose `ElementType` is in `threadgroup` memory and scope is `execution_simdgroups<2>,` call `threadgroup_barrier(mem_flags::mem_threadgroup)` before reading the result of the tensor. Another example is if the TensorOp `run` method writes to a tensor whose `ElementType` is in `device` memory and scope is `execution_simdgroup,` call `simdgroup_barrier(mem_flags::mem_device)` before reading the result of the tensor.

Table 7.2. TensorOps

TensorOp template classes	Description

`template <`
`  matmul2d_descriptor Desc,`
`  typename Scope,`
`  class... Args>`
`matmuld2d`
	Defines an object to perform a generalized matrix multiplication:
	
	`C = A*B + C`
	
	`A` and `B` can be host-bound, origin-shifted, or shader-allocated tensors.
	
	`C` can be host-bound, origin-shifted, shader-allocated tensors, or `cooperative_tensor`.

TensorOp template classes	Description
	See section 7.2.1 for more details.
`template < convolution2d_descriptor Desc, typename Scope, typename... ConvArgs> convolution2d`	Defines an object to perform a 2D convolution that occurs in neural networks. 2D stands for two spatial dimensions of width x height. The tensor consumed by this op is 4D. The only `Scope` current supported is `execution_simdgroups<N>` where `N` is `simdgroups_per_threadgroup` . See section 7.2.2 for more details.

7.2.1 Matrix Multiplication

The template class `matmul2d` performs a generalized matrix multiplication of two tensors (`C = A*B)` or matrix multiplication accumulated into a tensor (`C = A*B + C`).

The operation multiplies an `M`×`K` tensor `A` by a `K`×`N` tensor `B` and accumulates the result into an `M`×`N` tensor `C`. `A` and `B` can be host-bound, origin-shifted, or shader-allocated tensors. `C` can be a host-bound, origin-shifted, or shader-allocated tensor, or a `cooperative_tensor`. Table 7.3 shows the supported data type combinations and when each was first introduced.

Table 7.3. MatMul2D data type supported

Support	Tensor A type	Tensor B type	Tensor C type
Metal 4	`char`	`char`	`int`
Metal 4	`char`	`half`	`half`
Metal 4	`char`	`half`	`float`
Metal 4	`char`	`float`	`float`
Metal 4	`half`	`char`	`half`
Metal 4	`half`	`char`	`float`
Metal 4	`half`	`half`	`half`
Metal 4	`half`	`half`	`float`
Metal 4	`half`	`float`	`float`
Metal 4	`float`	`char`	`float`
Metal 4	`float`	`half`	`float`
Metal 4	`float`	`float`	`float`

Support	Tensor A type	Tensor B type	Tensor C type
Metal 4 and OS 26.1	bfloat	bfloat	bfloat
Metal 4 and OS 26.1	bfloat	bfloat	float
Metal 4 and OS 26.1	bfloat	float	float
Metal 4 and OS 26.1	bfloat	char	bfloat
Metal 4 and OS 26.1	bfloat	char	float
Metal 4 and OS 26.1	float	bfloat	float
Metal 4 and OS 26.1	char	bfloat	bfloat
Metal 4 and OS 26.1	char	bfloat	float
Metal 4 and OS 26.1	bfloat	half	bfloat
Metal 4 and OS 26.1	bfloat	half	half
Metal 4 and OS 26.1	bfloat	half	float
Metal 4 and OS 26.1	half	bfloat	bfloat
Metal 4 and OS 26.1	half	bfloat	half
Metal 4 and OS 26.1	half	bfloat	float
Metal 4 and OS 26.4	half	uchar	half
Metal 4 and OS 26.4	uchar	half	half
Metal 4 and OS 26.4	half	uchar	float
Metal 4 and OS 26.4	float	uchar	float
Metal 4 and OS 26.4	uchar	half	float
Metal 4 and OS 26.4	uchar	float	float
Metal 4 and OS 26.4	uchar	uchar	int
Metal 4 and OS 26.4	bfloat	uchar	bfloat
Metal 4 and OS 26.4	bfloat	uchar	float
Metal 4 and OS 26.4	uchar	bfloat	bfloat
Metal 4 and OS 26.4	uchar	bfloat	float
Metal 4 and OS 26.4	half	int4b_format	half
Metal 4 and OS 26.4	half	int4b_format	float
Metal 4 and OS 26.4	half	uint4b_format	half

Support	Tensor A type	Tensor B type	Tensor C type
Metal 4 and OS 26.4	`half`	`uint4b_format`	`float`
Metal 4 and OS 26.4	`char`	`int4b_format`	`int`
Metal 4 and OS 26.4	`uchar`	`uint4b_format`	`int`
Metal 4 and OS 26.4	`bfloat`	`int4b_format`	`bfloat`
Metal 4 and OS 26.4	`bfloat`	`uint4b_format`	`bfloat`
Metal 4 and OS 26.4	`bfloat`	`int4b_format`	`float`
Metal 4 and OS 26.4	`bfloat`	`uint4b_format`	`float`
Metal 4.1	`char`	`int2b_format`	`int`
Metal 4.1	`uchar`	`uint2b_format`	`int`
Metal 4.1	`half`	`int2b_format`	`half`
Metal 4.1	`half`	`int2b_format`	`float`
Metal 4.1	`half`	`uint2b_format`	`half`
Metal 4.1	`half`	`uint2b_format`	`float`
Metal 4.1	`bfloat`	`int2b_format`	`bfloat`
Metal 4.1	`bfloat`	`uint2b_format`	`bfloat`
Metal 4.1	`bfloat`	`int2b_format`	`float`
Metal 4.1	`bfloat`	`uint2b_format`	`float`
Metal 4.1	`half`	`metal_fp4_e2m1_format`	`half`
Metal 4.1	`half`	`metal_fp4_e2m1_format`	`float`
Metal 4.1	`half`	`metal_fp8_e4m3_format`	`half`
Metal 4.1	`half`	`metal_fp8_e4m3_format`	`float`
Metal 4.1	`half`	`metal_fp8_e5m2_format`	`half`
Metal 4.1	`half`	`metal_fp8_e5m2_format`	`float`
Metal 4.1	`metal_fp4_e2m1_format`	`metal_fp4_e2m1_format`	`half`

Support	Tensor A type	Tensor B type	Tensor C type
Metal 4.1	`metal_fp4_e2m1_format`	`metal_fp4_e2m1_format`	`float`
Metal 4.1	`metal_fp8_e4m3_format`	`metal_fp8_e4m3_format`	`half`
Metal 4.1	`metal_fp8_e4m3_format`	`metal_fp8_e4m3_format`	`float`
Metal 4.1	`metal_fp8_e5m2_format`	`metal_fp8_e5m2_format`	`half`
Metal 4.1	`metal_fp8_e5m2_format`	`metal_fp8_e5m2_format`	`float`

To create the `matmul2d`, you first build a descriptor using the following constructor:

```
matmul2d_descriptor(int_M, int N, int K = dynamic_length_v<int>,
                    bool transpose_left = false,
                    bool transpose_right = false,
                    bool relaxed_precision = false,
                    mode matmul_mode = mode::multiply);
```

Table 7.4. MatMul2D descriptor parameters

Parameter	Description
`M, N, K`	Tensor dimensions where `M` x `K` tensor `A`, `K` x `N` tensor `B`, and `M` x `N` tensor `C`.
`transpose_left`	Transpose matrix `A` before multiplying. The default is `false`.
`transpose_right`	Transpose matrix `B` before multiplying. The default is `false`.
`relaxed_precision`	Specifies if the operation can use relaxed precision for `float` data type. Relaxed precision allows the operation to truncate the mantissa before the multiplication. The default is `false`.
`matmul_mode`	Specifies whether to perform a `multiply` or `multiply_accumulate`. The default is `multiply`.

Table 7.5. MatMul2D member functions

MatMul2D member functions	Description
`template < typename LeftOperandType, typename RightOperandType, typename DestinationOperandType> void run(thread LeftOperandType &left, thread RightOperandType &right, thread DestinationOperandType &destination);`	Executes a matrix multiply of `C=A*B` where `C` is the `destination` tensor, `A` is `left` tensor, and `B` is `right` tensor.
`template < typename LeftOperandType, typename RightOperandType, typename ElementType, typename CoordType = int> cooperative_tensor<...> get_destination_cooperative_tensor() thread const;`	Returns a `cooperative_tensor` that can store the result of the matrix multiply.
`template < typename LeftOperandType, typename RightOperandType, typename ElementType, typename CoordType = int> cooperative_tensor<...> get_row_reduction_destination_cooperative_tensor() thread const;`	Returns a `cooperative_tensor` that can store the result of the row reduction on the result of the matrix multiply.
`template < typename LeftElementType, typename RightElementType, typename ElementType, typename CoordType = int> cooperative_tensor<...> get_left_input_cooperative_tensor() thread const;` All OS: Metal 4 and OS 26.1 and later	Returns a `cooperative_tensor` that can be used as the left input to a matrix multiply. Only valid if the op is scoped to `execution_simdgroup`.
`template < typename LeftElementType, typename RightElementType, typename ElementType, typename CoordType = int> cooperative_tensor<...> get_right_input_cooperative_tensor() thread const;`	Returns a `cooperative_tensor` that can be used as the right input to a matrix multiply.

All OS: Metal 4 and OS 26.1 and later	

`template <typename LeftOperandType,`
`typename RightOperandType,`
`typename ElementType,`
`typename CoordType = int>`
`cooperative_tensor<...>`
`get_column_reduction_destination_cooperative_tensor() thread const;`
	Returns a `cooperative_tensor` that can store the result of the column reduction on the result of the matrix multiply. Only valid if the op is scoped to `execution_simdgroup`.

`template <`
`typename LeftElementType,`
`typename RightElementType,`
`typename ElementType,`
`typename CoordType = int,`
`typename SrcElemType,`
`typename SrcExtents,`
`typename SrcLayout>`
`bool`
`is_compatiable_as_left_input(`
`const thread cooperative_tensor<…> & src)`
`thread const;`
 	
All OS: Metal 4 and OS 26.1 and later
	Returns `true` if the `cooperative_tensor` can be used as a left input.

`template <`
`typename LeftElementType,`
`typename RightElementType,`
`typename ElementType,`
`typename CoordType = int,`
`typename SrcElemType,`
`typename SrcExtents,`
`typename SrcLayout>`
`bool`
`is_compatiable_as_right_input(`
`const thread cooperative_tensor<…> & src)`
`thread const;`
 	
All OS: Metal 4 and OS 26.1 and later
	Returns `true` if the `cooperative_tensor` can be used as a right input.

To instantiate the template `matmul2d`, you pass the descriptor and the execution scope to the template:

```
template < matmul2d_descriptor Desc,
           typename Scope,
           class... Args>  matmuld2d;
```

To execute the matrix multiplication, call the `matmul2d` run method by passing the left tensor (A), the right tensor (B), and the destination tensor (C):

```
  template <
    typename LeftOperandType,
    typename RightOperandType,
    typename DestinationOperandType>
  void run(thread LeftOperandType &left,
           thread RightOperandType &right,
           thread DestinationOperandType &destination);
```

See Table 7.3 and **Error! Reference source not found.** for the element type supported for tensor A, B, C.

The example below illustrates the use of a `matmul2d` TensorOp with tensors:

```
#include <metal_tensor>
#include <MetalPerformancePrimitives/MetalPerformancePrimitives.h>
using namespace metal;
using namespace mpp;

[[ kernel ]] void matrixMultiply(
  tensor<device half, dextents<int, 2>> a [[ buffer(0) ]],
  tensor<device half, dextents<int, 2>> b [[ buffer(1) ]],
  tensor<device half, dextents<int, 2>> c [[ buffer(2) ]],
  uint2 tgid [[threadgroup_position_in_grid]]) {

  // Create a matmul op for a threadgroup made of 4 SIMD-groups.
  constexpr auto matmulDescriptor =
    tensor_ops::matmul2d_descriptor(64, 32, 0);

  tensor_ops::matmul2d<matmulDescriptor,
                       execution_simdgroups<4>> matmulOp;

  // Create the appropriate slice for this threadgroup to work on.
  auto mA = a.slice(0, tgid.y * 64);
  auto mB = b.slice(tgid.x * 32, 0);
  auto mC = c.slice(tgid.x * 32, tgid.y * 64);

  // Execute the operation assuming C is initialized to zero.
  matmulOp.run(mA, mB, mC);
}
```

To use a `cooperative_tensor` for the destination of a `matmul2d` TensorOp, use the following member function. The function returns a `cooperative_tensor` whose storage is divided across the threads in the scope of the `matmul2d`:

```
template <typename LeftOperandType,
          typename RightOperandType,
          typename ElementType, typename CoordType = int>
  cooperative_tensor<...>
  get_destination_cooperative_tensor() thread const;
```

The example below illustrates the use of a `matmul2d` TensorOp with `cooperative_tensor`:

```
#include <metal_tensor>
#include <MetalPerformancePrimitives/MetalPerformancePrimitives.h>
using namespace metal;
using namespace mpp;


[[ kernel ]] void gemmBias(
  tensor<device float, dextents<int, 2>> a [[ buffer(0) ]],
  tensor<device float, dextents<int, 2>> b [[ buffer(1) ]],
  tensor<device float, dextents<int, 2>> c [[ buffer(2) ]],
  device float*                          bufBias [[buffer(3)]],
  uint2 tgid [[threadgroup_position_in_grid]]) {

  // Build the bias tensor from the buffer.
  array<int,1> stride = {1};
  tensor<device float, dextents<int, 1>, tensor_inline>
    tBias(bufBias, dextents<int,1>(64), stride);

  // Create a matmul op for a threadgroup made of 4 SIMD-groups.
  constexpr auto matmulDescriptor =
    tensor_ops::matmul2d_descriptor(
      64, 32, 0, false, false, false,
      tensor_ops::matmul2d_descriptor::mode::multiply_accumulate);

  tensor_ops::matmul2d<matmulDescriptor,
                       execution_simdgroups<4>> matmulOp;

  // Create the cooperative tensor.
  auto cTc = matmulOp.get_destination_cooperative_tensor<
              decltype(a), decltype(b), float>();

  // Load the bias, run the matrix multiple and store the result.
  cTc.load(tBias);
  matmulOp.run(a, b, cTc);
  cTc.store(c);
}
```

To use a `cooperative_tensor` as the left or right input of a `matmul2d` TensorOp, use one of the following corresponding member functions. The function returns a

`cooperative_tensor` whose storage is divided across the threads participating in the `matmul2d` operation:

```
template <
   typename LeftElementType,
   typename RightElementType,
   typename ElementType,
   typename CoordType = int>
cooperative_tensor<...>
get_left_input_cooperative_tensor() thread const;

template <
   typename LeftElementType,
   typename RightElementType,
   typename ElementType,
   typename CoordType = int>
cooperative_tensor<...>
get_right_input_cooperative_tensor() thread const;
```

The example below illustrates the use of a `matmul2d` TensorOp with a `cooperative_tensor`:

```
#include <metal_tensor>
#include <MetalPerformancePrimitives/MetalPerformancePrimitives.h>
using namespace metal;
using namespace mpp;


[[ kernel ]] void gemmBias(
  tensor<device half, dextents<int, 2>> a [[ buffer(0) ]],
  tensor<device half, dextents<int, 2>> b [[ buffer(1) ]],
  tensor<device float, dextents<int, 2>> c [[ buffer(2) ]],
  device float*                          bufBias [[buffer(3)]],
  uint2 tgid [[threadgroup_position_in_grid]]) {


  // Create a matmul op for a threadgroup made of 1 SIMD-group.
  constexpr auto matmulDescriptor =
    tensor_ops::matmul2d_descriptor(
      64, 32, 0, false, false, false,
      tensor_ops::matmul2d_descriptor::mode::multiply_accumulate);

  tensor_ops::matmul2d<matmulDescriptor,
                       execution_simdgroup> gemmOp;

  // Create the left input cooperative tensor.
```

```
  auto ctLeft = gemmOp.get_left_input_cooperative_tensor<
                  half, half, float>();
  ctLeft.load(a);

  // Updated input operand.
  for (int i = 0; i < ctLeft.get_capacity(); i++)
    ctLeft[i] *= 2.0f;

  // Create the cooperative tensor.
  auto cTc = gemmOp.get_destination_cooperative_tensor<
               decltype(a), decltype(b), float>();

  // Run the matrix multiple and store the result.
  gemmOp.run(ctLeft, b, cTc);
  cTc.store(c);
}
```

You can do a row or column `sum`, `max`, or `min` reduction of a `cooperative_tensor` into a destination 1D `cooperative_tensor` if the scope of the `matmul2d` is `execution_simdgroup`.

Table 7.6. Reduction related functions for cooperative tensors

Reduction related functions	Description

`template <`
`  class ElementType,`
`  class SrcExtents,`
`  class DstExtents,`
`  class SrcLayout,`
`  class DstLayout>`
`inline void reduce_rows(`
`  thread metal::cooperative_tensor<`
`  ElementType, SrcExtents,`
`  SrcLayout> &source,`
`  thread metal::cooperative_tensor<`
`  ElementType, DstExtents,`
`  DstLayout> &destination,`
`  reduction_operation op =`
`  reduction_operation::sum,`
`  ElementType identity =`
`  reduction_operation_identity<`
`  ElementType>::sum_identity);`
	Returns the reduction of each row and stores the result into the 1D destination `cooperative_tensor`. The default is a `sum` reduction for each row.

`template <` `  class ElementType,`	Returns the reduction of each column and stores the result into

`  class SrcExtents,`
`  class DstExtents,`
`  class SrcLayout,`
`  class DstLayout>`
`inline void reduce_columns(`
`  thread metal::cooperative_tensor<`
`  ElementType, SrcExtents,`
`  SrcLayout> &source,`
`  thread metal::cooperative_tensor<`
`  ElementType, DstExtents,`
`  DstLayout> &destination,`
`  reduction_operation op =`
`  reduction_operation::sum,`
`  ElementType identity =`
`  reduction_operation_identity<`
`  ElementType>::sum_identity);`
	the 1D destination `cooperative_tensor`. The default is a `sum` reduction for each column.

`template <`
`  class SrcElementType,`
`  class DstElementType,`
`  class SrcExtents,`
`  class DstExtents,`
`  class SrcLayout,`
`  class DstLayout>`
`inline bool is_iterator_compatible(`
`  const thread metal::cooperative_tensor<`
`  SrcElementType,`
`  SrcExtents,`
`  SrcLayout> &source,`
`  const thread metal::cooperative_tensor<`
`  DstElementType,`
`  DstExtents,`
`  DstLayout> &destination);`
	Returns `true` if you can use the result of the reduction with another tensor using the `map_iterator`. To check if the iterators are compatible, call the following nonmember function.

To get the destination tensor for a row reduction, call the following member function:

```
template <typename LeftOperandType,
        typename RightOperandType,
        typename ElementType, typename CoordType = int>
cooperative_tensor<...>
get_row_reduction_destination_cooperative_tensor() thread const;
```

To get the destination tensor for a column reduction, call the following member function:

```
template <typename LeftOperandType,
```

```
          typename RightOperandType,
          typename ElementType, typename CoordType = int>
  cooperative_tensor<...>
  get_column_reduction_destination_cooperative_tensor() thread
                                                        const;
```

Use the enumeration to define the type of reduction:

```
  enum class reduction_operation {
    sum, // Take the sum of the element of the row/column.
    max, // Take the max value of all elements in row/column.
    min, // Take the min value of all elements in row/column.
};
```

Use the following structure to define the identity value for the type of reduction:

```
template <typename ElementType>
struct reduction_operation_identity
{
  static const constant ElementType sum_identity;
  static const constant ElementType max_identity;
  static const constant ElementType min_identity;
};
```

Call the following nonmember function to return the reduction of each row and store the result into the 1D destination `cooperative_tensor`. The default is a `sum` reduction for each row.

```
  template <class ElementType, class SrcExtents,
            class DstExtents, class SrcLayout,
            class   DstLayout>
  inline void reduce_rows(
    thread metal::cooperative_tensor<ElementType, SrcExtents,
                                     SrcLayout> &source,
    thread metal::cooperative_tensor<ElementType, DstExtents,
                                     DstLayout> &destination,
    reduction_operation op = reduction_operation::sum,
    ElementType identity =
        reduction_operation_identity<ElementType>::sum_identity);
```

Call the following nonmember function to return the reduction of each column and store the result into the 1D destination `cooperative_tensor`. The default is a `sum` reduction for each column.

```
template <class ElementType, class SrcExtents, class DstExtents,
          class SrcLayout, class DstLayout>
  inline void reduce_columns(
    thread metal::cooperative_tensor<ElementType, SrcExtents,
                                     SrcLayout> &source,
    thread metal::cooperative_tensor<ElementType, DstExtents,
                                     DstLayout> &destination,
    reduction_operation op = reduction_operation::sum,
    ElementType identity =
        reduction_operation_identity<ElementType>::sum_identity);
```

The example below demonstrates how to do a row reduction:

```
[[ kernel ]] void gemm_reduce(
  tensor<device float, dextents<int, 2>> aT [[ buffer(0) ]],
  tensor<device float, dextents<int, 2>> bT [[ buffer(1) ]],
  tensor<device float, dextents<int, 2>> cT [[ buffer(2) ]],
  tensor<device float, dextents<int, 1>> dR [[ buffer(3) ]],
  uint2 tgid [[threadgroup_position_in_grid]]) {

  constexpr auto matmulDescriptor =
    tensor_ops::matmul2d_descriptor(64, 32, 0);

  tensor_ops::matmul2d<matmulDescriptor,
                       execution_simdgroup> matmulOp;

  // Create the cooperative tensor.
  auto cTdest = matmulOp.get_destination_cooperative_tensor<
                  decltype(aT), decltype(bT), float>();

  // Run the matrix multiple.
  matmulOp.run(aT, bT, cTdest);

  // Sum up each row and store the results.
  auto cTred =
    matmulOp.get_row_reduction_destination_cooperative_tensor<
               decltype(aT), decltype(bT), float>();

  reduce_rows(cTdest, cTred, tensor_ops::reduction_operation::sum,
              0.0f);
  cTred.store(dR);
}
```

You can use the result of the reduction with another tensor using the `map_iterator`. To check if the iterators are compatible, call the following nonmember function:

```
  template <class SrcElementType, class DstElementType,
            class SrcExtents, class DstExtents,
            class SrcLayout, class DstLayout>
    inline bool is_iterator_compatible(
                  const thread metal::cooperative_tensor<
                                SrcElementType,
                                SrcExtents,
                                SrcLayout> &source,
                  const thread metal::cooperative_tensor<
                                DstElementType,
                                DstExtents,
                                DstLayout> &destination);
```

The following example shows a use of `is_iterator_compatible` and `map_iterator`:

```
[[ kernel ]] void gemm_map(
  tensor<device float, dextents<int, 2>> aT [[ buffer(0) ]],
  tensor<device float, dextents<int, 2>> bT [[ buffer(1) ]],
  tensor<device float, dextents<int, 2>> dT [[ buffer(2) ]])
{
  constexpr auto matmulDescriptor =
    tensor_ops::matmul2d_descriptor(64, 32, 0);

  tensor_ops::matmul2d<matmulDescriptor,
                       execution_simdgroup> matmulOp;

  // Create the cooperative tensor.
  auto cTdest = matmulOp.get_destination_cooperative_tensor<
                  decltype(aT), decltype(bT), float>();

  // Load the bias, run the matrix multiple, and store the result.
  matmulOp.run(aT, bT, cTdest);

  auto cTred =
    matmulOp.get_row_reduction_destination_cooperative_tensor<
                decltype(aT), decltype(bT), float>();

  auto identity = metal::numeric_limits<float>::lowest();
  reduce_rows(cTdest, cTred, tensor_ops::reduction_operation::min,
              identity);

  // Check if the iterators are compatible and if so, add
```

```
  // the min across the rows.
  if (tensor_ops::is_iterator_compatible(cTdest, cTred)) {
    for (auto it = cTdest.begin(); it != cTdest.end(); it++) {
      auto cTred_it = cTred.map_iterator(it);
      *it += *cTred_it;
    }
  }
  else {
     // Do something else.
  }

  cTdest.store(dT);
}
```

For more detailed information, see the `MPPTensorOpsMatMul2d.h` header.

7.2.2 Convolution

The template class `convolution2d` performs a 2D convolution where 2D stands for two spatial dimensions of width x height. The operation takes an activation and a weight tensor to produce a `tensor` or `cooperative_tensor` as described in Table 7.7.

To create a `convolution2d`, you first build a descriptor using the constructor below:

```
  enum class convolution2d_activation_layout {
    nhwc,
  };

  enum class convolution2d_weights_layout {
    hwio,
  };

  convolution2d_descriptor(
    int4 destination_dimensions,
    int4 source_dimensions,
    int2 kernel_dimensions,
    convolution2d_activation_layout activation_layout =
      convolution2d_activation_layout::nhwc,
    convolution2d_weights_layout weight_layout =
      convolution2d_weights_layout::hwio,
    int2 strides = int2(1, 1),
    int2 dilations = int2(1, 1),
    int  groups = 1,
    bool relaxed_precision = false,
    mode convolution2d_mode = mode::multiply);
```

Table 7.7. Convolution2d parameters

Parameter	Description
`destination_dimensions`	Specifies the dimension of the output tensor.
`source_dimensions`	Specifies the dimension of the input tensor.
`kernel_dimensions`	Specifies the size of the convolution window.
`activation_layout`	Specifies the layout of the activation tensor.
`weights_layout`	Specifies the layout of the weight tensor.
`strides`	Specifies the stride of the convolution
`dilations`	Specifies the spacing between kernel elements.
`groups`	Specifies the number of groups the input is split to the channel axis.
`relaxed_precision`	Specifies if the operation can use relaxed precision for `float` data type. Relaxed precision allows the operation to truncate the mantissa before the multiplication.
`convolution2d_mode`	Specifies whether to perform a `multiply` or `multiply_accumulate`.

To instantiate the template `convolution2d`, you pass the descriptor and scope. Currently, the only scope supported is `execution_simdgroups<N>` where `N` is `simdgroups_per_threadgroup`.

```
template <
  convolution2d_descriptor Desc,
  typename Scope,
  typename... ConvArgs>
convolution2d;
```

To execute the convolution, call the `convolution2d` run method:

```
template <typename ActivationTensorType,
          typename WeightsTensorType,
          typename DestinationTensorType, typename... RunArgs>
void run(thread ActivationTensorType &activation,
         thread WeightsTensorType &weights,
         thread DestinationTensorType &destination) const;
```

Table 7.8. Convolution run parameter

Parameter	Description
`activation`	The activation tensor with `NHWC` layout: `N` = batch (slowest moving dimension) `H` = height `W` = width `C` = input channels (fastest moving dimension)
`weights`	The weights tensor with `HWIO` layout: `H` = kernel height `W` = kernel width `I` = input channels `O` = output channels (fastest moving dimension)
`destination`	The destination tensor which can be a `tensor` or a `cooperative_tensor`. If it is a `tensor`, the format is `NHWO` layout: `N` = batch (slowest moving dimension) `H` = height `W` = width `O` = output channels (fastest moving dimension)

For more detailed information, please see the `MPPTensorOpsConvolution2d.h` header.

8 Numerical Compliance

This chapter covers how Metal represents floating-point numbers regarding accuracy in mathematical operations. Metal is compliant to a subset of the IEEE 754 standard.

8.1 INF, NaN, and Denormalized Numbers

INF must be supported for single-precision, half-precision, and brain floating-point numbers.

NaNs must be supported for single-precision, half-precision, and brain floating-point numbers (with fast math disabled). If fast math is enabled the behavior of handling NaN or INF (as inputs or outputs) is undefined. Signaling NaNs are not supported.

Denormalized single-precision, half-precision, or brain floating-point numbers passed as input to or produced as the output of single-precision, half-precision, or brain floating-point arithmetic operations may be flushed to zero.

8.2 Rounding Mode

Either round ties to even or round toward zero rounding mode may be supported for single-precision, half-precision, and brain floating-point operations.

8.3 Floating-Point Exceptions

Floating-point exceptions are disabled in Metal.

8.4 ULPs and Relative Error

Table 8.1 describes the minimum accuracy of single-precision floating-point basic arithmetic operations and math functions given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Table 8.1. Accuracy of single-precision floating-point operations and functions

Math function	Minimum accuracy (ULP values)
`x + y`	Correctly rounded
`x - y`	Correctly rounded
`x * y`	Correctly rounded
`1.0 / x`	Correctly rounded
`x / y`	Correctly rounded

Math function	Minimum accuracy (ULP values)
`acos`	<= 4 ulp
`acosh`	<= 4 ulp
`asin`	<= 4 ulp
`asinh`	<= 4 ulp
`atan`	<= 5 ulp
`atan2`	<= 6 ulp
`atanh`	<= 5 ulp
`ceil`	Correctly rounded
`copysign`	0 ulp
`cos`	<= 4 ulp
`cosh`	<= 4 ulp
`cospi`	<= 4 ulp
`exp`	<= 4 ulp
`exp2`	<= 4 ulp
`exp10`	<= 4 ulp
`fabs`	0 ulp
`fdim`	Correctly rounded
`floor`	Correctly rounded
`fma`	Correctly rounded
`fmax`	0 ulp
`fmin`	0 ulp
`fmod`	0 ulp
`fract`	Correctly rounded
`frexp`	0 ulp
`ilogb`	0 ulp
`ldexp`	Correctly rounded
`log`	<= 4 ulp
`log2`	<= 4 ulp

Math function	Minimum accuracy (ULP values)
`log10`	<= 4 ulp
`modf`	0 ulp
`nextafter`	0 ulp
`pow`	<= 16 ulp
`powr`	<= 16 ulp
`rint`	Correctly rounded
`round`	Correctly rounded
`rsqrt`	Correctly rounded
`sin`	<= 4 ulp
`sincos`	<= 4 ulp
`sinh`	<= 4 ulp
`sinpi`	<= 4 ulp
`sqrt`	Correctly rounded
`tan`	<= 6 ulp
`tanpi`	<= 6 ulp
`tanh`	<= 5 ulp
`trunc`	Correctly rounded

Table 8.2 describes the minimum accuracy of single-precision floating-point arithmetic operations given as ULP values with fast math enabled (which is the default unless you specify `-fno-fast-math` as a compiler option).

Table 8.2. Accuracy of single-precision operations and functions with fast math enabled

Math function	Minimum accuracy (ULP values)
`x + y`	Correctly rounded
`x - y`	Correctly rounded
`x * y`	Correctly rounded
`1.0 / x`	<= 1 ulp for `x` in the domain of 2^{-126} to 2^{126}

Math function	Minimum accuracy (ULP values)
`x / y`	<= 2.5 ulp for `y` in the domain of 2^{-126} to 2^{126}
`acos(x)`	<= 5 ulp for `x` in the domain [-1, 1]
`acosh(x)`	Implemented as `log(x + sqrt(x * x - 1.0))`
`asin(x)`	<= 5 ulp for `x` in the domain [-1, 1] and |`x`| >= 2^{-125}
`asinh(x)`	Implemented as `log(x + sqrt(x * x + 1.0))`
`atan(x)`	<= 5 ulp
`atanh(x)`	Implemented as `0.5 * (log( (1.0 + x) / (1.0 - x) )`
`atan2(y, x)`	Implemented as if `x > 0`, `atan(y / x)`, if `x < 0` and `y > 0`, `atan(y / x) + M_PI_F` if `x < 0` and `y < 0`, `atan(y / x) - M_PI_F` and if `x = 0` or `y = 0`, the result is undefined.
`ceil`	Correctly rounded
`copysign`	0 ulp
`cos(x)`	For `x` in the domain [-pi, pi], the maximum absolute error is <= 2^{-13} and larger otherwise.
`cosh(x)`	Implemented as `0.5 * (exp(x) + exp(-x))`
`cospi(x)`	For `x` in the domain [-1, 1], the maximum absolute error is <= 2^{-13} and larger otherwise.
`exp(x)`	<= `3 + floor(fabs(2 * x))` ulp
`exp2(x)`	<= `3 + floor(fabs(2 * x))` ulp
`exp10(x)`	Implemented as `exp2(x * log2(10))`
`fabs`	0 ulp
`fdim`	Correctly rounded
`floor`	Correctly rounded
`fma`	Correctly rounded
`fmax`	0 ulp
`fmin`	0 ulp
`fmod`	Undefined

Math function	Minimum accuracy (ULP values)
`fract`	Correctly rounded
`frexp`	0 ulp
`ilogb`	0 ulp
`ldexp`	Correctly rounded
`log(x)`	For `x` in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise if `x > 0` the maximum error is <= 3 ulp; otherwise the results are undefined.
`log2(x)`	For `x` in the domain [0.5, 2], the maximum absolute error is <= 2^{-22}; otherwise if `x > 0` the maximum error is <= 2 ulp; otherwise the results are undefined.
`log10(x)`	Implemented as `log2(x) * log10(2)`
`modf`	0 ulp
`pow(x, y)`	Implemented as `exp2(y * log2(x))`. Undefined for `x = 0` and `y = 0`.
`powr(x, y)`	Implemented as `exp2(y * log2(x))`. Undefined for `x = 0` and `y = 0`.
`rint`	Correctly rounded
`round(x)`	Correctly rounded
`rsqrt`	<= 2 ulp
`sin(x)`	For `x` in the domain [-pi, pi], the maximum absolute error is <= 2^{-13} and larger otherwise.
`sinh(x)`	Implemented as `0.5 * (exp(x) - exp(-x))`
`sincos(x)`	ULP values as defined for `sin(x)` and `cos(x)`
`sinpi(x)`	For `x` in the domain [-1, 1], the maximum absolute error is <= 2^{-13} and larger otherwise.
`sqrt(x)`	Implemented as `x * rsqrt(x)` with special cases handled correctly.
`tan(x)`	Implemented as `sin(x) * (1.0 / cos(x))`
`tanh(x)`	Implemented as `(t - 1.0)/(t + 1.0)`, where `t = exp(2.0 * x)`
`tanpi(x)`	Implemented as `tan(x *` pi`)`
`trunc`	Correctly rounded

Table 8.3 describes the minimum accuracy of half-precision floating-point basic arithmetic operations and math functions given as ULP values. Table 8.3 applies to iOS and macOS, starting with Apple GPU Family 4 hardware.

Table 8.3. Accuracy of half-precision floating-point operations and functions

Math function	Minimum accuracy (ULP values)
`x + y`	Correctly rounded
`x - y`	Correctly rounded
`x * y`	Correctly rounded
`1.0 / x`	Correctly rounded
`x / y`	Correctly rounded
`acos(x)`	<= 1 ulp
`acosh(x)`	<= 1 ulp
`asin(x)`	<= 1 ulp
`asinh(x)`	<= 1 ulp
`atan(x)`	<= 1 ulp
`atanh(x)`	<= 1 ulp
`atan2(y, x)`	<= 1 ulp
`ceil`	Correctly rounded
`copysign`	0 ulp
`cos(x)`	<= 1 ulp
`cosh(x)`	<= 1 ulp
`cospi(x)`	<= 1 ulp
`exp(x)`	<= 1 ulp
`exp2(x)`	<= 1 ulp
`exp10(x)`	<= 1 ulp
`fabs`	0 ulp
`fdim`	Correctly rounded
`floor`	Correctly rounded

Math function	Minimum accuracy (ULP values)
`fma`	Correctly rounded
`fmax`	0 ulp
`fmin`	0 ulp
`fmod`	0 ulp
`fract`	Correctly rounded
`frexp`	0 ulp
`ilogb`	0 ulp
`ldexp`	Correctly rounded
`log(x)`	<= 1 ulp
`log2(x)`	<= 1 ulp
`log10(x)`	<= 1 ulp
`modf`	0 ulp
`nextafter`	0 ulp
`rint`	Correctly rounded
`round(x)`	Correctly rounded
`rsqrt`	Correctly rounded
`sin(x)`	<= 1 ulp
`sinh(x)`	<= 1 ulp
`sincos(x)`	ULP values as defined for sin(`x`) and cos(`x`)
`sinpi(x)`	<= 1 ulp
`sqrt(x)`	Correctly rounded
`tan(x)`	<= 1 ulp
`tanh(x)`	<= 1 ulp
`tanpi(x)`	<= 1 ulp
`trunc`	Correctly rounded

Table 8.4 describes the minimum accuracy of brain floating-point basic arithmetic operations and math functions given as ULP values. Table 8.4 applies to all OS, starting with Apple GPU Family 6 or Metal GPU Family 3.

Table 8.4. Accuracy of brain floating-point operations and functions

Math function	Minimum accuracy (ULP values)
`x + y`	Correctly rounded
`x - y`	Correctly rounded
`x * y`	Correctly rounded
`1.0 / x`	Correctly rounded
`x / y`	Correctly rounded

Table 8.5. Accuracy of brain floating-point operations and functions with fast math enabled

Math function	Minimum accuracy (ULP values)
`x + y`	Correctly rounded
`x - y`	Correctly rounded
`x * y`	Correctly rounded
`1.0 / x`	<= 0.6 ulp for `x` in the domain of 2^{-126} to 2^{126}
`x / y`	<= 0.6 ulp for `y` in the domain of 2^{-126} to 2^{126}

Even though the precision of individual math operations and functions are specified in Table 8.1, Table 8.2, Table 8.3, Table 8.4, and Table 8.5, the Metal compiler, in fast math mode (see section 1.6.5), may do various optimization like reassociate floating-point operations that may dramatically change results in floating-point. Reassociation may change or ignore the sign of zero, allow optimizations to assume the arguments and result are not `NaN` or +/-`INF`, inhibit or create underflow or overflow and thus cannot be in code that relies on rounding behavior such as `(x + 2^52) - 2^52`, or ordered floating-point comparisons.

The ULP is defined as follows:

If `x` is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then `ulp(x) = |b - a|`, otherwise `ulp(x)` is the distance between the two nonequal finite floating-point numbers nearest `x`. Moreover, `ulp(NaN)` is `NaN`.

8.5 Edge Case Behavior in Flush to Zero Mode

If denormalized values are flushed to zero, then a function may return one of four results:

1. Any conforming result when not in flush to zero mode.

2. If the result given by step 1 is a subnormal before rounding, it may be flushed to zero.
3. Any nonflushed conforming result for the function if one or more of its subnormal operands are flushed to zero.
4. If the result of step 3 is a subnormal before rounding, the result may be flushed to zero.

In each of the above cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

8.6 Conversion Rules for Floating-Point and Integer Types

When converting from a floating-point type to an integer, the conversion uses round toward zero rounding mode. Use the "round ties to even" or "round toward zero" rounding mode for conversions from a floating-point or integer type to a floating-point type.

The conversions from `half` and `bfloat` to `float` are lossless. Conversions from `float` to `half` or to `bfloat` round the mantissa using the round ties to even rounding mode. When converting a `float` to a `half`, denormalized numbers generated for the `half` data type may not be flushed to zero.

When converting a floating-point type to an integer type, if the floating-point value is NaN, the resulting integer is 0.

Note that fast math does not change the accuracy of conversion operations.

8.7 Texture Addressing and Conversion Rules

The texture coordinates specified to the `sample`, `sample_compare`, `gather`, `gather_compare`, `read`, and `write` functions cannot be INF or NaN. An out-of-bound texture `read` returns the default value for each component, as described in section 6.13, and Metal ignores an out-of-bound texture `write`.

The following sections discuss the application of conversion rules when reading and writing textures in a graphics or kernel function. When performing a multisample resolve operation, these conversion rules do not apply.

8.7.1 Conversion Rules for Normalized Integer Pixel Data Types

This section discusses converting normalized integer pixel data types to floating-point values and vice-versa.

8.7.1.1 Converting Normalized Integer Pixel Data Types to Floating-Point Values

For textures that have 8-, 10-, or 16-bit normalized unsigned integer pixel values, the texture sample and read functions convert the pixel values from an 8- or 16-bit unsigned integer to a normalized single- or half-precision floating-point value in the range `[0.0 … 1.0]`.

For textures that have 8- or 16-bit normalized signed integer pixel values, the texture sample and read functions convert the pixel values from an 8- or 16-bit signed integer to a normalized single- or half-precision floating-point value in the range `[−1.0 … 1.0]`.

These conversions are performed as listed in the second column of Table 8.6. The precision of the conversion rules is guaranteed to be <= 1.5 ulp, except for the cases described in the "Corner Cases" column.

Table 8.6. Conversion to a normalized float value

Convert from	Conversion rule to normalized float	Corner cases
1-bit normalized unsigned integer	`float(c)`	0 must convert to 0.0 1 must convert to 1.0
2-bit normalized unsigned integer	`float(c) / 3.0`	0 must convert to 0.0 3 must convert to 1.0
4-bit normalized unsigned integer	`float(c) / 15.0`	0 must convert to 0.0 15 must convert to 1.0
5-bit normalized unsigned integer	`float(c) / 31.0`	0 must convert to 0.0 31 must convert to 1.0
6-bit normalized unsigned integer	`float(c) / 63.0`	0 must convert to 0.0 63 must convert to 1.0
8-bit normalized unsigned integer	`float(c) / 255.0`	0 must convert to 0.0 255 must convert to 1.0
10-bit normalized unsigned integer	`float(c) / 1023.0`	0 must convert to 0.0 1023 must convert to 1.0
16-bit normalized unsigned integer	`float(c) / 65535.0`	0 must convert to 0.0 65535 must convert to 1.0
8-bit normalized signed integer	`max(−1.0, float(c)/127.0)`	-128 and -127 must convert to -1.0 0 must convert to 0.0 127 must convert to 1.0
16-bit normalized signed integer	`max(−1.0, float(c)/32767.0)`	-32768 and -32767 must convert to -1.0 0 must convert to 0.0 32767 must convert to 1.0

8.7.1.2 Converting Floating-Point Values to Normalized Integer Pixel Data Types

For textures that have 8-, 10-, or 16-bit normalized unsigned integer pixel values, the texture write functions convert the single- or half-precision floating-point pixel value to an 8- or 16-bit unsigned integer.

For textures that have 8- or 16-bit normalized signed integer pixel values, the texture write functions convert the single- or half-precision floating-point pixel value to an 8- or 16-bit signed integer.

NaN values are converted to zero.

Conversions from floating-point values to normalized integer values are performed as listed in Table 8.7.

Table 8.7. Conversion from floating-point to a normalized integer value

Convert to	Conversion rule to normalized integer
1-bit normalized unsigned integer	`x = min(max(f, 0.0), 1.0)` `i0:0 = intRTNE(x)`
2-bit normalized unsigned integer	`x = min(max(f * 3.0, 0.0), 3.0)` `i1:0 = intRTNE(x)`
4-bit normalized unsigned integer	`x = min(max(f * 15.0, 0.0), 15.0)` `i3:0 = intRTNE(x)`
5-bit normalized unsigned integer	`x = min(max(f * 31.0, 0.0), 31.0)` `i4:0 = intRTNE(x)`
6-bit normalized unsigned integer	`x = min(max(f * 63.0, 0.0), 63.0)` `i5:0 = intRTNE(x)`
8-bit normalized unsigned integer	`x = min(max(f * 255.0, 0.0), 255.0)` `i7:0 = intRTNE(x)`
10-bit normalized unsigned integer	`x = min(max(f * 1023.0, 0.0), 1023.0)` `i9:0 = intRTNE(x)`
16-bit normalized unsigned integer	`result = min(max(f * 65535.0, 0.0), 65535.0)` `i15:0 = intRTNE(x)`
8-bit normalized signed integer	`result = min(max(f * 127.0, -127.0), 127.0)` `i7:0 = intRTNE(x)`
16-bit normalized signed integer	`result = min(max(f * 32767.0, -32767.0),32767.0)` `i15:0 = intRTNE(x)`

In Metal 2, all conversions to and from unormalized data types round correctly.

8.7.2 Conversion Rules for Half-Precision Floating-Point Pixel Data Type

For textures that have half-precision floating-point pixel color values, the conversions from `half` to `float` are lossless. Conversions from `float` to `half` round the mantissa using the round ties to even rounding mode. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may not be flushed to zero. A `float` NaN may

be converted to an appropriate NaN or be flushed to zero in the `half` type. A `float` INF must be converted to an appropriate INF in the `half` type.

8.7.3 Conversion Rules for Single-Precision Floating-Point Pixel Data Type

The following rules apply for reading and writing textures that have single-precision floating-point pixel color values:

- NaNs may be converted to a NaN value(s) or be flushed to zero.
- INFs must be preserved.
- Denormalized numbers may be flushed to zero.
- All other values must be preserved.

8.7.4 Conversion Rules for 10- and 11-bit Floating-Point Pixel Data Type

The floating-point formats use 5 bits for the exponent, with 5 bits of mantissa for 10-bit floating-point types, or 6-bits of mantissa for 11-bit floating-point types with an additional hidden bit for both types. There is no sign bit. The 10- and 11-bit floating-point types preserve denormalizes.

These floating-point formats use the following rules:

- If the exponent and mantissa are 0, the floating-point value is 0.0.
- If the exponent is 31 and the mantissa is != 0, the resulting floating-point value is a NaN.
- If the exponent is 31 and the mantissa is 0, the resulting floating-point value is positive infinity.
- If 0 <= exponent <= 31, the floating-point value is 2 ^ (exponent - 15) * (1 + mantissa/N).
- If the exponent is 0 and the mantissa is != 0, the floating-point value is a denormalized number given as 2 ^ (exponent – 14) * (mantissa / N). If mantissa is 5 bits, N is 32; if mantissa is 6 bits, N is 64.

Conversion of a 10- or 11-bit floating-point pixel data type to a half- or single-precision floating-point value is lossless. Conversion of a half or single precision floating-point value to a 10- or 11-bit floating-point value must be <= 0.5 ULP. Any operation that results in a value less than zero for these floating-point types is clamped to zero.

8.7.5 Conversion Rules for 9-bit Floating-Point Pixel Data Type with a 5-bit Exponent

The `RGB9E5_SharedExponent` shared exponent floating-point format uses 5 bits for the exponent and 9 bits for the mantissa. There is no sign bit.

Conversion from this format to a half- or single-precision floating-point value is lossless and computed as 2 ^ (shared exponent – 15) * (mantissa/512) for each color channel.

Conversion from a half or single precision floating-point RGB color value to this format is performed as follows, where `N` is the number of mantissa bits per component (9), `B` is the exponent bias (15) and `Emax` is the maximum allowed biased exponent value (31).

- Clamp the `r`, `g`, and `b` components (in the process, mapping NaN to zero) as follows:

```
rc = max(0, min(sharedexpmax, r)
gc = max(0, min(sharedexpmax, g)
bc = max(0, min(sharedexpmax, b)
```

Where `sharedexpmax = ((2^N – 1)/2^N) * 2(Emax– B)`:

- Determine the largest clamped component `maxc`:

```
maxc = max(rc, gc, bc)
```

- Compute a preliminary shared exponent `expp`:

```
expp = max(–B – 1, floor(log2(maxc)) + 1 + B
```

- Compute a refined shared exponent `exps`:

```
maxs = floor((maxc / 2^(expp–B–N) + 0.5f)
exps = expp, if 0 <= maxs < 2^N, and exps = expp + 1, if maxs = 2^N.
```

- Finally, compute three integer values in the range `0` to `2N – 1`:

```
rs = floor(rc / 2^(exps–B–N) + 0.5f)
gs = floor(gc / 2^(exps–B–N) + 0.5f)
bs = floor(bc / 2^(exps–B–N) + 0.5f)
```

Conversion of a half- or single-precision floating-point color values to the `MTLPixelFormatRGB9E5Float` shared exponent floating-point value is <= 0.5 ULP.

8.7.6 Conversion Rules for Signed and Unsigned Integer Pixel Data Types

For textures that have an 8- or 16-bit signed or unsigned integer pixel values, the texture sample and read functions return a signed or unsigned 32-bit integer pixel value. The conversions described in this section must be correctly saturated.

Writes to these integer textures perform one of the conversions listed in Table 8.8.

Table 8.8. Conversion between integer pixel data types

Convert from	To	Conversion rule
32-bit signed integer	8-bit signed integer	`result = convert_char_saturate(val)`
32-bit signed integer	16-bit signed integer	`result = convert_short_saturate(val)`
32-bit unsigned integer	8-bit unsigned integer	`result = convert_uchar_saturate(val)`
32-bit unsigned integer	16-bit unsigned integer	`result = convert_ushort_saturate(val)`

8.7.7 Conversion Rules for sRGBA and sBGRA Textures

Conversion from sRGB space to linear space is automatically done when sampling from an sRGB texture. The conversion from sRGB to linear RGB is performed before the filter specified in the sampler specified when sampling the texture is applied. If the texture has an alpha channel, the alpha data is stored in linear color space.

Conversion from linear to sRGB space is automatically done when writing to an sRGB texture. If the texture has an alpha channel, the alpha data is stored in linear color space.

The following is the conversion rule for converting a normalized 8-bit unsigned integer from an sRGB color value to a floating-point linear RGB color value (call it `c`):

```
if (c <= 0.04045)
     result = c / 12.92;
else
     result = powr((c + 0.055) / 1.055, 2.4);
```

The precision of the above conversion must ensure that the delta between the resulting infinitely precise floating-point value when converting `result` back to an unnormalized sRGB value but without rounding to an 8-bit unsigned integer value (call it `r`) and the original sRGB 8-bit unsigned integer color value (call it `rorig`) is <= 0.5; for example:

```
fabs(r – rorig) <= 0.5
```

Use the following rules for converting a linear RGB floating-point color value (call it `c`) to a normalized 8-bit unsigned integer sRGB value:

```
if (isnan(c)) c = 0.0;
if (c > 1.0)
     c = 1.0;
else if (c < 0.0)
     c = 0.0;
else if (c < 0.0031308)
     c = 12.92 * c;
else
     c = 1.055 * powr(c, 1.0/2.4) – 0.055;

// Convert to integer scale: c = c * 255.0.
// Convert to integer: c = c + 0.5.
// Drop the decimal fraction.
// Convert the remaining floating-point(integral) value
// to an integer.
```

The precision of the above conversion shall be:

```
fabs(reference result – integer result) < 1.0.
```

9 Appendix

9.1 New in Metal 3.2

Metal 3.2 introduces the following new features:

- Relaxed Math (section 1.6.3)
- Intersection Result Reference (section 2.17.5)
- Texture and Buffer Memory Coherency (section 2.9 and section 4.8), Thread Scope (section 6.16.2), and Fence Functions (section 6.16.3)
- Global Bindings (section 5.9)
- Logging (section 6.20)

9.2 New in Metal 4

Metal 4 introduces the following new features:

- C++17 based (section 1.5)
- Sampler LOD bias, minimum and maximum reduction (section 2.10)
- Intersection Function Buffers (section 2.17.1, 5.1.6, 5.2.3.7 , 6.19.2, 6.19.4,and 6.19.8)
- Per-Vertex values (section 2.19)
- Tensors (section 2.22)
- User annotations (section 5.1.12)
- Texture atomics for cube and cube array textures (section 6.13.6 and 6.13.7)
- Pack and unpack of `snorm10a2` (section 6.15)
- Indirect command buffer support for raster and depth stencil states (section 6.17.1)
- Metal Performance Primitives (section 7)

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
408-996-1010