Metal Shading Language Specification

Version 2.2

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, 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. The Metal shading language is a single, unified language that allows tighter integration between the graphics and compute programs. Since the Metal shading language is C++-based, you will find it familiar and easy to use.

The Metal shading language 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++14. 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 1.0, 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.

1.3 References

C++14

Stroustrup, Bjarne. *The C++ Programming Language (Fourth Edition)*. Harlow: Addison-Wesley, 2013.

Metal

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

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

1.4 Metal and C++14

The Metal programming language is a C++14-based Specification with extensions and restrictions. Refer to the C++14 Specification (also known as the ISO/IEC JTC1/SC22/WG21 N4431 Language Specification) for a detailed description of the language grammar.

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

For more about Metal preprocessing directives and compiler options, see section 1.5 of this document.

1.4.1 Overloading

Metal supports overloading, as defined by section 13 of the 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.)

1.4.2 Templates

Metal supports templates, as defined by section 14 of the C++14 Specification.

1.4.3 Preprocessing Directives

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

1.4.4 Restrictions

The following C++14 features are not available in Metal (section numbers in this list refer to the C++14 Specification):

- lambda expressions (section 5.1.2)
- recursive function calls (section 5.2.2, item 9)
- 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)
- 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`, or `threadgroup_imageblock` address space attribute. (For more about Metal address space attributes, see section 4 of this document.)
- Function pointers are not supported.

A Metal function cannot be called `main`.

1.5 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, and miscellaneous options. The online and offline Metal compilers support these options.

1.5.1 Preprocessor Compiler Options

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

`-D name`

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

`-D name=definition`

The contents of `definition` are tokenized and processed as if they appeared 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.

1.5.2 Preprocessor Definitions

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

```
__METAL_VERSION__ // Value of SDK version
__METAL_MACOS__   // Set if built against macOS SDK or on a macOS device
__METAL_IOS__     // Set if built against iOS SDK or on an iOS device
```

You can use definitions to conditionally apply shading language features that are only available on later SDKs (by comparing the version number) or for specific platforms (macOS or iOS).

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 2.0, you can use the preprocessor definition in code, as follows:

```
#if __METAL_VERSION__ >= 200

// Code that requires features introduced in Metal 2.0.

#endif
```

1.5.3 Math Intrinsics Compiler Options

These options control compiler behavior regarding floating-point arithmetic, trading off between speed and correctness.

```
    -ffast-math (default)
    -fno-fast-math
```

For more about math functions, see section 6.5. For more about the relative errors of ordinary and fast math functions, see section 7.4.

These options enable (the default) 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 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.
- Fast: Allow algebraically equivalent transformations, such as reassociating floating-point operations that may dramatically change the floating-point results.

1.5.4 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.

1.5.5 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`: Support the unified graphics and computing language revision 1.0 programs for iOS 8.
- `ios-metal1.1`: Support the unified graphics and computing language revision 1.1 programs for iOS 9.
- `ios-metal1.2`: Support the unified graphics and computing language revision 1.2 programs for iOS 10.
- `ios-metal2.0`: Support the unified graphics and computing language revision 2.0 programs for iOS 11.
- `ios-metal2.1`: Support the unified graphics and computing language revision 2.1 programs for iOS 12.
- `ios-metal2.2`: Support the unified graphics and computing language revision 2.2 programs for iOS 13.
- `osx-metal1.1`: Support the unified graphics and computing language revision 1.1 programs for macOS 10.11.
- `osx-metal1.2`: Support the unified graphics and computing language revision 1.2 programs for macOS 10.12.

- `osx-metal2.0`: Support the unified graphics and computing language revision 2.0 programs for macOS 10.13.
- `osx-metal2.1`: Support the unified graphics and computing language revision 2.1 programs for macOS 10.14.
- `osx-metal2.2`: Support the unified graphics and computing language revision 2.2 programs for macOS 10.15.

1.5.6 Compiler Options to Request or Suppress Warnings

The following options are available:

`-Werror`

Make all warnings into errors.

`-W`

Inhibit all warning messages.

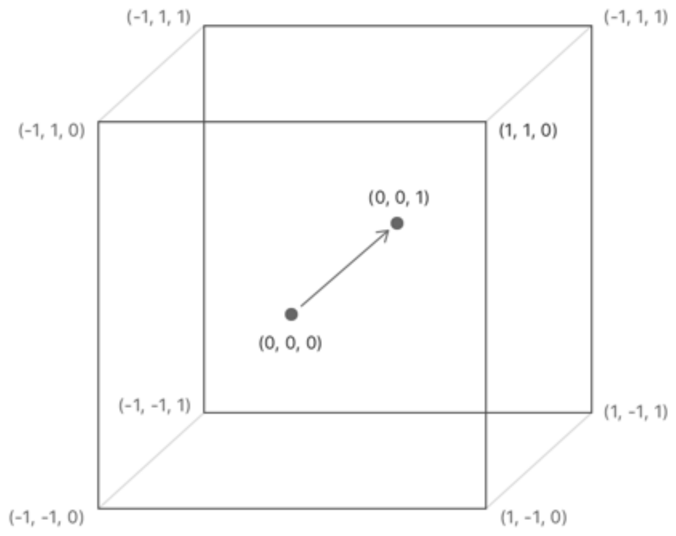
1.6 Metal Coordinate Systems

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*.

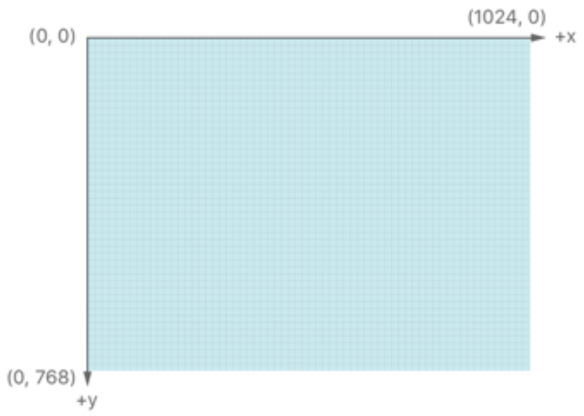
Normalized device coordinates use a *left-handed coordinate system* (see Figure 1) and map to positions in the viewport. These coordinates are independent of viewport size. The lower-left corner of the viewport is at an `(x,y)` coordinate of `(-1.0,-1.0)` and the upper corner is at `(1.0,1.0)`. Positive-`z` 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



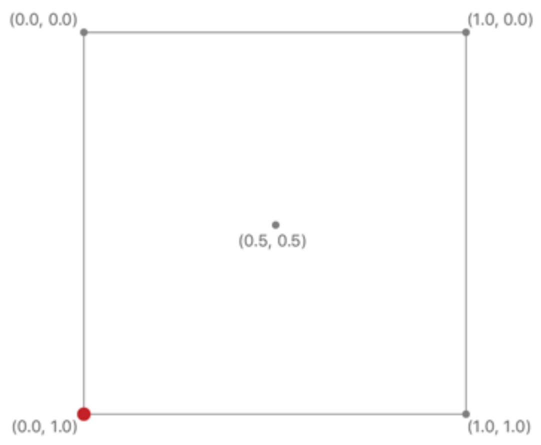
The rasterizer stage transforms normalized-device coordinates into *viewport coordinates* (see Figure 2). The `(x,y)` coordinates in this space are measured in pixels, with the origin in the top-left corner of the viewport and positive values going to the right and down.

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: Since Metal 2.2.
	A signed two's complement 64-bit integer.

`unsigned long`
`uint64_t`
All OS: Since Metal 2.2.
	An unsigned 64-bit integer.

`half`	A 16-bit floating-point. The `half` data type must conform to the IEEE 754 binary16 storage format.

Type	Description
`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 `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

`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

Type	Size (in bytes)	Alignment (in bytes)

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

`size_t`	8	8
`half`	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`
- `floatn`

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

Type	Size (in bytes)	Alignment (in bytes)
`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
`long3` `ulong3`	32	32
`long4` `ulong4`	32	32
`half2`	4	4
`half3`	8	8
`half4`	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);
```

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, which is given as the number of components * size of each component. 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, the vector's components are constructed in order from the components of the arguments. The arguments are consumed from left to right. Each argument has all its components consumed, in order, before any components from the next argument are consumed.

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 aforementioned 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_floatn`

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.)

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_float2`	8	4
`packed_float3`	12	4
`packed_float4`	16	4

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. However, you cannot use the `.xyzw` or `.rgba` selection syntax to access components of a packed vector data type.

Example:

```
packed_float4 f;
f[0] = 1.0f;  // OK
f.x = 1.0f;   // This is illegal and results in a compilation error.
```

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` is composed of `n` `floatm` vectors. Similarly, a matrix of type `halfnxm` is composed of `n` `halfm` vectors.

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

Type	Size (in bytes)	Alignment (in bytes)
`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;
```

You can 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 non-constant 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

You can 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);
```

Matrix components are constructed and consumed in column-major order. The matrix constructor must have just enough values specified 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.

A matrix of type `T` with `n` columns and `m` rows can also be constructed 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);
```

Since Metal 2.0, 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 are **not** supported. A matrix cannot be constructed from combinations of vectors and scalars.

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

2.4 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.

2.5 Atomic Data Types

Objects of atomic types are the only Metal shading language objects that 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 these atomic types:

`atomic_int`	All OS: since Metal 1.0
`atomic_uint`	All OS: since Metal 1.0
`atomic_bool`	iOS: since Metal 2.0; not supported on macOS
`atomic<T>`	iOS: since Metal 2.0; not supported on macOS

`atomic<T>` represents templated types, where `T` can be `int`, `uint`, or `bool`.

Metal atomic functions (as described in section 6.13) can only use Metal atomic data types. These atomic functions are a subset of the C++14 atomic and synchronization functions.

2.6 Pixel Data Types

All OS: Pixel data types supported since Metal 2.0.

The Metal pixel data type is a templated type that describes the pixel format type and its corresponding ALU type. 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 the Metal shading language, 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

Pixel Data Type	Supported values of T	Size (in bytes)	Alignment (in bytes)
`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;
}
```

2.7 Buffers

The Metal shading language 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: Metal does *not* support buffers that contain `long` or `ulong` data types.

The example below shows buffers as arguments to a function that performs the Phong interpolation model. 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 2.12.

2.8 Textures

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. The following templates define specific texture data types:

```
enum class access { sample, read, write, read_write };
texture1d<T, access a = access::sample>
texture1d_array<T, access a = access::sample>
texture2d<T, access a = access::sample>
texture2d_array<T, access a = access::sample>
texture3d<T, access a = access::sample>
texturecube<T, access a = access::sample>
texturecube_array<T, access a = access::sample>
texture2d_ms<T, access a = access::read>
texture2d_ms_array<T, access a = access::read>
```

You must declare textures with depth formats as one of the following texture data types:

```
depth2d<T, access a = access::sample>
depth2d_array<T, access a = access::sample>
depthcube<T, access a = access::sample>
depthcube_array<T, access a = access::sample>
depth2d_ms<T, access a = access::read>
depth2d_ms_array<T, access a = access::read>
```

macOS supports `texture2d_ms_array` and `depth2d_ms_array` since Metal 2.0. All other types supported since Metal 1.0.

iOS supports all types except `texture2d_ms_array` and `depth2d_ms_array` since Metal 1.0.

`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: `read_write` access supported since Metal 1.2. Other access qualifiers since Metal 1.0.

Multisampled textures only support the `read` attribute. Depth textures only support the `sample` and `read` 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.
    …
}
```

2.8.1 Texture Buffers

All OS: Texture buffers supported since Metal 2.1.

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, and 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.

An out-of-bounds write is ignored.

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 2.11.1. For more about texture buffer, see section 6.10.16.

2.9 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 a sampler is initialized 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: No support.
	`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`, then only one level-of-detail is active.

`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.

macOS: Supports `clamp_to_border` address mode and `border_color` since Metal 1.2.

iOS: No support for `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, the behavior is undefined.

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 LOD (level-of-detail) 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 are initialized in the Metal shading language 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)]],
          …)
{
      …
}
```

2.10 Imageblocks

iOS: Supports imageblocks since Metal 2.0.

macOS: No support for 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 2.6), 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 6.11 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`, `A8Unorm`, `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`
`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`

Pixel Storage Type	Compatible Texture Formats
`rgb9e5<T>`	`RGB9E5Float`

2.11 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.)

2.11.1 Arrays of Textures, Texture Buffers, and Samplers

iOS: Arrays of textures supported since Metal 1.2; arrays of samplers since Metal 2.0; arrays of texture buffers since Metal 2.1.
macOS: Arrays of textures supported since Metal 2.0; arrays of samplers since Metal 2.0; arrays of texture buffers since Metal 2.1.

Declare an array of textures as either:

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

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

`typename` shall be a texture type declared with the `access::read` or `access::sample` attribute. On macOS since Metal 2.0, support for an array of writeable textures (`access::write`) is also available. (For more about texture types, see section 2.8.)

Construct an array of texture buffers (see section 2.8.1) 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 2.12), you cannot declare an `array<T, N>` type (an array of textures, texture buffers, or samplers) in a structure.

The Metal shading language 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 2.11.1.1 to 2.11.1.3 are available for the array of textures, array of samplers, and the `array_ref<T>` types.

2.11.1.1 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;
```

2.11.1.2 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);
    }
}
```

2.11.1.3 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) };
```

2.11.2 Structures of Buffers and Textures

Arguments to a graphics, kernel, 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 are examples of invalid use-cases that shall result in a compilation error:

```
kernel void
my_kernel(device Foo& f) // This is an illegal use.
{…}

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.
{…}
```

2.12 Argument Buffers

All OS: Argument buffers supported since Metal 2.0.

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, an ID is automatically assigned according to the following rules:

1. IDs are assigned to structure members in order, by adding 1 to the maximum ID used by 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. IDs are assigned to array elements in order, by adding 1 to the maximum ID used by 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, its structure members or array elements are assigned indices 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. Top-level argument buffer arguments are assigned IDs starting from 0, according to the previous three rules.

2.12.1 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];
}


kernel void
kern(constant texture2d<float> *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);
```

2.13 Uniform Type

All OS: uniform types supported since Metal 2.0.

2.13.1 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 non-uniform; 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 has to 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, the Metal shading language adds a new template class called uniform (available in the header `metal_uniform`) that can be used to declare 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));
}
```

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

```
uniform<int> x = …;
int y = …;
int z = x*y;        // x is converted to a non-uniform for a multiply
uniform<int> b = z; // illegal; 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; compile-time error
```

The results of expressions that combine uniform with non-uniform variables are non- uniform. If the non-uniform result is assigned to a uniform variable, as in the example below, the behaviors 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) {      // non-uniform result for expression (i < j)
```

```
    …
    i++;  // compile-time error, undefined behavior
}
```

The following example is similar:

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

2.13.3 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 non-uniform quantities.

2.14 Type Conversions and Re-interpreting Data

The `static_cast` operator is used to convert from a scalar or vector type to another scalar or vector type with no saturation and with a default rounding mode. (When converting to floating-point, round to nearest; when converting to 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.
// g.w is undefined
float3 g = as_type<float3>(f);
```

2.15 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`.

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;     // 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 or matrix.

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++14 Specification.

3 Operators

All OS: Scalar, vector, and matrix operators supported since Metal 1.0.

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 6.14.3.

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 is converted to the element type used by the vector operand.
     - The scalar type is then widened to a vector that has the same number of components as the vector operand.
     - The operation is performed component-wise, which results in a same size vector.
   - If the two operands are vectors of the same size, the operation is performed component-wise, 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 7.)

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 is converted to the element type of the vector operand.
     - The scalar type is then widened to a vector that has the same number of components as the vector operand.
     - The operation is performed component-wise, which results in a same-size vector.

- If the two operands are vectors of the same size, the operation is performed component-wise, 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 non-zero operands remain defined.

If both operands are non-negative, the remainder is non-negative.

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 component-wise on their operands. The result is the same type as the operand. For post- and pre-increment and decrement, the expression must be assignable to an lvalue. Pre-increment and pre-decrement add or subtract 1 to the contents of the expression on which they operate, and the value of the pre-increment or pre-decrement 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 is converted to the element type of the vector operand.
     - The scalar type is then widened to a vector that has the same number of components as the vector operand.
     - The operation is performed component-wise, which results in a Boolean vector.
   - If the two operands are vectors of the same size, the operation is performed component-wise, 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 6.4.)
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 is converted to the element type of the vector operand.
     - The scalar type is then widened to a vector that has the same number of components as the vector operand.

    - The operation is performed component-wise, which results in a Boolean vector.
  - If the two operands are vectors of the same size, the operation is performed component-wise, 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, the bitwise operators are applied component-wise.
   - If one operand is a scalar and the other is a vector,
     - The scalar is converted to the element type used by the vector operand.
     - The scalar type is then widened to a vector that has the same number of components as the vector operand.
     - The bitwise operation is performed component-wise 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. Only one of the second and third expressions is evaluated. The second and third expressions can be of any type if:
    - the types of the second and third expressions match,
    - or there is a type conversion for one of the expressions that can make their types match (for more about type conversions, see section 2.11),
    - or 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 must 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, the operators are applied component-wise. For the right-shift (`>>`) and left-shift (`<<`) operators, if the first operand is a scalar, the rightmost operand must 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++14 Specification. For the `lvalue = expression` assignment operation, if `expression` is a scalar type and `lvalue` is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is performed component-wise, which results in a same size vector.

Other C++14 operators that are not detailed above (such as `sizeof(T)`, unary (`&`) operator, and comma (`,`) operator) behave as described in the C++14 Specification.

Unsigned integers shall obey the laws of arithmetic modulo 2^n, 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. The operation is done component-wise 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 must 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 a Metal Shading Language programs. An address space attribute specifies the region of memory from where a 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)

All OS: `device`, `threadgroup`, `constant`, and `thread` attribute support since Metal 1.0.

macOS: No support for `threadgroup_imageblock` attribute.

iOS: Supports the `threadgroup_imageblock` attribute since Metal 2.0.

All arguments to a graphics or kernel function that are a pointer or reference to a type must be declared with an address space attribute. For graphics functions, an argument that is a pointer or reference to a type must be declared in the `device` or `constant` address space. For kernel functions, an argument that is a pointer or reference to a type must 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 must be `constant`.

Any variable that is a pointer or reference must 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;
```

Since you always allocate texture objects from the device address space, you do not 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 6.10).

4.2 constant Address Space

The `constant` address space name refers to buffer memory objects allocated from the device memory pool but are read-only. Variables in program scope must be declared in the `constant` address space and initialized during the declaration statement. The initializer(s) expression must be a core constant expression. (Refer to section 5.19 of the C++14 specification.) 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.

To decide which address space (`device` or `constant`) a read-only buffer passed to a graphics or kernel function uses, look at how the buffer is accessed inside the graphics or kernel function. The `constant` address space is optimized for multiple instances executing a graphics or kernel function accessing the same location in the buffer. Some examples of this access pattern are accessing light or material properties for lighting / shading, matrix of a matrix array used for skinning, filter weight accessed from a filter weight array for convolution. If multiple executing instances of a graphics or kernel function are accessing the buffer using an index such as the vertex ID, fragment coordinate, or the thread position in grid, the buffer must be allocated in the `device` address space.

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

Threads are organized into *threadgroups*. Threads in a threadgroup cooperate by sharing data through `threadgroup` memory and by synchronizing their execution to coordinate memory accesses to both `device` and `threadgroup` memory. The threads in a given threadgroup execute concurrently on a single compute unit on the GPU. A GPU may have multiple compute units. Multiple threadgroups can execute concurrently across multiple compute units.

The `threadgroup` address space name is used to allocate variables used by a kernel function or passed as an argument to a fragment function. (`threadgroup` space arguments to fragment functions are only supported on iOS with at least A11 hardware.) Variables declared in the `threadgroup` address space *cannot* be used in vertex functions and *cannot* be declared inside a fragment function.

Variables allocated in the `threadgroup` 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 is executing the kernel.

Variables allocated in the `threadgroup` address space for a mid-render kernel function are allocated for each threadgroup executing the kernel and are persistent across mid-render and fragment kernel functions over a tile.

The example below shows how variables allocated in the `threadgroup` address space can be passed either as arguments or be declared inside a kernel function. (The `[[threadgroup(0)]]` attribute in the code below is explained in section 5.2.1.)

```
kernel void
my_kernel(threadgroup float *a [[threadgroup(0)]],
            …)
```

```
{
      // A float allocated in the threadgroup address space
      threadgroup float x;

      // An array of 10 floats allocated in the
      // threadgroup address space
      threadgroup float b[10];
      …
}
```

4.4.1 SIMD-groups and Quad-groups

macOS: SIMD-group support since Metal 2.0. Quad-group support since Metal 2.1.

iOS: No support for SIMD-groups. Quad-group support since Metal 2.0.

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 must 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 6.8 and its subsections:

- For more about thread synchronization functions, including a SIMD-group barrier, see section 6.8.1.
- For more about SIMD-group functions, see section 6.8.2.
- For more about quad-group functions, see section 6.8.3.

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 2.10). A pointer to a user-defined type allocated in the `threadgroup_address` address space can be an argument to a tile shading function (see section 5.1.4). 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 particular spatial relationship between the threads and the imageblock. The threadgroup dimensions may be smaller than the tile size.

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

All OS: Kernel, vertex, and fragment attributes supported since Metal 1.0.

- Metal supports the following function attributes that specify how to use a function: `vertex`, `fragment`, and `kernel`, which are detailed in sections 5.1.1, 5.1.2, and 5.1.3, respectively. These function attributes are used at the start of a function, before its return type.

Functions that use a `kernel`, `vertex`, or `fragment` function attribute must not call functions that also use these attributes, or a compilation error results.

You can declare functions that use a `kernel`, `vertex`, or `fragment` function attribute within a namespace.

5.1.1 Vertex Functions

You can declare the `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 function with the `vertex` attribute.

```
vertex void
my_vertex_func(…)
{…}
```

For a vertex function, the return type identifies whether the output generated by the function is either per-vertex or per-fragment. If the vertex function does not generate output, it shall return `void`.

5.1.1.1 Post-Tessellation Vertex Functions

All OS: post-tessellation vertex function (`patch` attribute) supported since Metal 1.2.

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 has executed, the tessellated primitives are rasterized.

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` must 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` 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(…)
{…}
```

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 shall return `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 value; that is, the return type of this fragment function includes an element declared with the `[[depth(depth_attribute]]` 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` attribute.

```
kernel void
my_kernel(…)
{…}
```

Functions declared with the `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.

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 shall fail.

5.1.4 Tile Functions

iOS: Tile function support since Metal 2.0.

macOS: No support for 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 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.

5.2 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.) 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 2.7).
- Constant buffer — A pointer or reference to any data type in the `constant` address space (see section 2.7).
- A `texture` object (see section 2.8) or an array of textures.
- A `texture_buffer` object (see section 2.8.1) or an array of texture buffers.
- A `sampler` object (see section 2.9) 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 2.10).
- An argument buffer (see section 2.12).
- A structure with elements that are buffers, textures, or texture buffers.

Buffers (device and constant) 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 and constant buffers: `[[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 do not. `src` is explicitly assigned texture index 0, and `f` is explicitly assigned buffer index 10. 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: raster order group attribute supported since Metal 2.0.

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 must 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(coord, clr);

}
```

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: patch control-point templated type supported since Metal 1.2.

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 5.1 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 5.1. Attributes for 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.
`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.
`vertex_id`	`ushort` or `uint`	The per-vertex identifier, which includes the base vertex value if one is specified.

Notes on vertex function input attribute types:

- If the type used to declare `[[vertex_id]]` is `uint`, the type used to declare `[[base_vertex]]` must be `uint` or `ushort`.
- If the type used to declare `[[vertex_id]]` is `ushort`, the type used to declare `[[base_vertex]]` must be `ushort`.
- If the type used to declare `[[instance_id]]` is `uint`, the type used to declare `[[base_instance]]` must be `uint` or `ushort`.
- If the type used to declare `[[instance_id]]` is `ushort`, the type used to declare `[[base_instance]]` must be `ushort`.

5.2.3.2 Post-Tessellation Vertex Function Input Attributes

Table 5.2 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.

Table 5.2. 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.
`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`.

All OS: All attributes in Table 5.2 are supported since Metal 1.2.

Notes on vertex function input attributes:

- If the type used to declare `[[instance_id]]` is `uint`, the type used to declare `[[base_instance]]` must be `uint` or `ushort`.
- If the type used to declare `[[instance_id]]` is `ushort`, the type used to declare `[[base_instance]]` must be `ushort`.

5.2.3.3 Vertex Function Output Attributes

Table 5.3 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).

Table 5.3. Attributes for vertex function return type

Attribute	**Corresponding Data Types**	**Description**
`clip_distance`	`float` or `float[n]` `n` must be known at compile time	Distance from vertex to clipping plane

Attribute	Corresponding Data Types	Description
`invariant` All OS: Since Metal 2.1	Not applicable; must 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.
`point_size`	`float`	Size of a point primitive
`position`	`float4`	The transformed vertex position
`render_target_array_index` macOS: Since Metal 1.1. iOS: Since Metal 2.1.	`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.
`viewport_array_index` macOS: Since Metal 2.0. iOS: Since Metal 2.1.	`uchar`, `ushort`, or `uint`	The viewport (and scissor rectangle) index value of the primitive.

All OS: Attributes in Table 5.3 are supported since Metal 1.0, unless otherwise indicated.

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 5.9.

`[[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 highly likely (although not guaranteed) to be invariant. 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.

5.2.3.4 Fragment Function Input Attributes

Table 5.4 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 5.4. Attributes for fragment function input arguments

Attribute	**Corresponding Data Types**	**Description**
`barycentric_coord` macOS: Since Metal 2.2. iOS: No support.	`float`, `float2`, or `float3`	The barycentric coordinates.

Attribute	**Corresponding Data Types**	**Description**
`color(m)` macOS: No support. iOS: Since Metal 1.0.	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn` `m` must 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: Since Metal 2.2. iOS: No support.	`uint`	The per-primitive identifier used with barycentric coordinates.
`render_target_array_index` macOS: Since Metal 1.1. iOS: Since Metal 2.1.	`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, or an array slice, 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.

Attribute	Corresponding Data Types	Description
sample_mask, post_depth_coverage All OS: Since Metal 2.0.	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 must be used on the fragment function; otherwise the compilation fails.
thread_index_in_quadgroup All OS: Since Metal 2.2.	ushort or uint	The scalar index of a thread within a quad-group.
thread_index_in_simdgroup All OS: Since Metal 2.2.	ushort or uint	The scalar index of a thread within a SIMD-group.
threads_per_simdgroup All OS: Since Metal 2.2.	ushort or uint	The thread execution width of a SIMD-group.
viewport_array_index macOS: Since Metal 2.0. iOS: Since Metal 2.1.	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 5.9.

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 5.5 lists attributes that can be specified for tile arguments that are input to a fragment function. The data types used to declare `[[pixel_position_in_tile]]` and `[[pixels_per_tile]]` must match.

Table 5.5. 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.

macOS: no support for the attributes in Table 5.5.

iOS: attributes in Table 5.5 are supported since Metal 2.0.

`[[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 5.6 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 5.6. Attributes for fragment function return types

Attribute	Corresponding Data Types	Description

`color(m)`
All OS: Since Metal 1.0.
 	
`color(m), index(i)`
All OS: Since Metal 1.2.
	`floatn`, `halfn`, `intn`, `uintn`, `shortn`, or `ushortn`	Color value output for a color attachment.
		
		`m` is the color attachment index and must 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: Since Metal 1.0.	`float`	Depth value output using the function specified by `depth_argument`.
`sample_mask` All OS: Since Metal 1.0.	`uint`	Coverage mask.
`stencil` All OS: Since Metal 2.1.	`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 5.4). Multiple elements in the fragment function return type that use the same color attachment index for blending must 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)]]` must 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` must be known at compile time.

If a fragment function writes a depth value, the `depth_argument` must be specified with one of the following values:

`any`

`greater`

`less`

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 `thread_execution_width`, determines the recommended size of this smaller group. For best performance, make the total number of threads in the threadgroup a multiple of the `thread_execution_width` .

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 5.7 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.

Table 5.7. Attributes for kernel function input arguments

Attribute	Corresponding Data Types	Description
`dispatch_quadgroups_per_threadgroup` macOS: Since Metal 2.1. iOS: Since Metal 2.0.	`ushort` or `uint`	The quad-group execution width of a threadgroup specified at dispatch.
`dispatch_simdgroups_per_threadgroup` macOS: Since Metal 2.0. iOS: No support.	`ushort` or `uint`	The SIMD-group execution width of a threadgroup specified at dispatch.
`dispatch_threads_per_threadgroup` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup for threads specified at dispatch.
`grid_origin` All OS: Since Metal 1.2.	`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.
`grid_size` All OS: Since Metal 1.2.	`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: Since Metal 2.1. iOS: Since Metal 2.0.	`ushort` or `uint`	The scalar index of a quad-group within a threadgroup.
`quadgroups_per_threadgroup` macOS: Since Metal 2.1. iOS: Since Metal 2.0.	`ushort` or `uint`	The quad-group execution width of a threadgroup.
`simdgroup_index_in_threadgroup` macOS: Since Metal 2.0. iOS: No support.	`ushort` or `uint`	The scalar index of a SIMD-group within a threadgroup.
`simdgroups_per_threadgroup` macOS: Since Metal 2.0. iOS: No support.	`ushort` or `uint`	The SIMD-group execution width of a threadgroup.
`thread_execution_width` All OS: Since Metal 1.0.	`ushort` or `uint`	The execution width of the compute unit.

Attribute	Corresponding Data Types	Description
`thread_index_in_quadgroup` macOS: Since Metal 2.1. iOS: Since Metal 2.0.	`ushort` or `uint`	The scalar index of a thread within a quad-group.
`thread_index_in_simdgroup` macOS: Since Metal 2.0. iOS: No support.	`ushort` or `uint`	The scalar index of a thread within a SIMD-group.
`thread_index_in_threadgroup` All OS: Since Metal 1.0.	`ushort` or `uint`	The scalar index of a thread within a threadgroup.
`thread_position_in_grid` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's position in an N-dimensional grid of threads.
`thread_position_in_threadgroup` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread's unique position within a threadgroup
`threadgroup_position_in_grid` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The threadgroup's unique position within a grid.
`threadgroups_per_grid` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The number of threadgroups in a grid.
`threads_per_grid` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The grid size.
`threads_per_simdgroup` macOS: Since Metal 2.0. iOS: No support.	`ushort` or `uint`	The thread execution width of a SIMD-group.
`threads_per_threadgroup` All OS: Since Metal 1.0.	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2`, or `uint3`	The thread execution width of a threadgroup.

All OS: Since Metal 1.2, `grid_origin` and `grid_size` supported.

macOS: Since Metal 2.0, SIMD-group attributes supported. Since Metal 2.1, quad-group attributes supported. Other attributes supported since Metal 1.0.

iOS: Since Metal 2.0, quad-group attributes supported. No support for SIMD-group attributes.

Other attributes supported since Metal 1.0.

For standard Metal compute functions (other than tile functions), SIMD-groups are linear and one-dimensional. (Threadgroups may be multi-dimensional.) 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 (`[[thread_execution_width]]`):

```
simdgroups_per_threadgroup = ceil(threads_per_threadgroup/
thread_execution_width)
```

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 /
thread_execution_width
```

```
simdgroups_per_threadgroup = ceil(Wx/2)*ceil(Wy/2)*4/thread_execution_width
```

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. Threads within a SIMD-group do not need to perform any barrier operations for synchronization. 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 2.0, 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 used to declare `[[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]] must be a scalar type or a vector type. If it is a vector type, the number of components for the vector types used to declare these arguments must match.
- The data types used to declare [[thread_position_in_grid]] and [[threads_per_grid]] must match.
- The data types used to declare [[thread_position_in_threadgroup]], [[threads_per_threadgroup]], and [[dispatch_threads_per_threadgroup]] must match.
- If [[thread_position_in_threadgroup]] is type uint, uint2 or uint3, then [[thread_index_in_threadgroup]] must be type uint.
- The types used to declare [[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]]must be ushort or uint. The types used to declare these built-in variables must match.
- [[thread_execution_width]] and [[threads_per_simdgroup]] are aliases of one another that reference the same concept.

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.

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[4];
};

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 6.10.3).

```
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];
};

constexpr sampler s = sampler(coord::normalized, address::clamp_to_edge,
                              filter::linear);

vertex VertexOutput
render_vertex(const device VertexInput *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(VertexOutput 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 used to declare 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 okay.

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. 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 structure member declared with the `stage_in` attribute. 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 must be within both the primitive and the centroid of the pixel.

The following example demonstrates how to specify the interpolatation 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]];

};
```

5.5 Per-Fragment Function versus Per-Sample Function

The fragment function is typically executed per-fragment. The sampling attribute identifies if fragment input interpolation will be 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

macOS: No support for imageblocks.

iOS: Support for imageblock since Metal 2.0.

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)]]` must be unique for each data member in the imageblock.
- The attribute name given by `[[user(name)]]` for a data member must match with a data member declared in the master explicit imageblock structure, and their associated data types must 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 must 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. Since 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 used to declare 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 2.6 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 5.4 of section 5.2.3.4). The color index `m` must 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 2.6,

- 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 5.4 of section 5.2.3.4). The color index `m` must 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 [[imagelock_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),
- or 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 and cannot become fragment function input.

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 the attribute name given by `[[user(name)]]` for an element, then this attribute name must match with an element in the return type of the vertex function, and their corresponding data types must also match.
- If you do not specify the `[[user(name)]]` attribute name, then the argument name and types must match.

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;
}
```

5.8 Program Scope Function Constants

All OS: function constants supported since Metal 1.2. Since Metal 2.0, support for using 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 a large number of 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)]]
```

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.

Function constants are not initialized in the Metal function source. Instead, their values are specified during the creation of the render or compute pipeline state. 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)]];
```

Functions 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.

The value of function constants `a` and `b` are specified 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, the behavior is the same as when the value of `is_function_constant_defined(name)` is `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 `non-zero` 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 `non-zero` 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: Using a function constant to specify resource bindings supported since Metal 2.0.

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 must 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: using a function constant to specify a color attachment or raster order group attribute index supported since Metal 2.0.

The `[[color(n)]]` or `[[raster_order_group(index)]]` index can also be a function constant. The function constant used must 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: Using a function constant to identify that a structure element is optional; supported since Metal 2.0.

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 non-zero 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 Per-Primitive Viewport and Scissor Rectangle Index Selection

macOS: Support for the `viewport_array_index` attribute since Metal 2.0.
iOS: Support for the `viewport_array_index` attribute since Metal 2.1.

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.

5.10 Additional Restrictions

Metal shading language 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 must 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 at https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf lists the specific limits. (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 (MSL).

6.1 Namespace and Header Files

The MSL functions and enumerations are declared in the `metal` namespace. In addition to the header files described in the MSL functions, the `<metal_stdlib>` header is available and can access all the functions supported by the MSL.

6.2 Common Functions

The functions in Table 6.1 are defined in the header `<metal_common>`. `T` is one of the scalar or vector 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` `a` must 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.

Built-in Common Functions	Description

`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 1.5.3) 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.

6.3 Integer Functions

The integer functions in Table 6.2 are defined in the header `<metal_integer>`. `T` is one of the scalar or vector integer types. T_u 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|`.
T_u `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`.

Built-in Integer Functions	Description
`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`.
`T extract_bits(T x, uint offset, uint bits)` All OS: Since Metal 1.2.	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: Since Metal 1.2.	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.
`T32 mad24(T32 x, T32 y, T32 z)` All OS: Since Metal 2.1.	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.
`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: Since Metal 2.1.	Returns `max(x, max(y, z))`.

Built-in Integer Functions	**Description**
`T median3(T x, T y, T z)` All OS: Since Metal 2.1.	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: Since Metal 2.1.	Returns `min(x, min(y, z))`.
`T32 mul24(T32 x, T32 y)` All OS: Since Metal 2.1.	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 non-zero bits in `x`.
`T reverse_bits(T x)` All OS: Since Metal 2.1.	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.

6.4 Relational Functions

The relational functions in Table 6.3 are defined in the header `<metal_relational>`. `T` is one of the scalar or vector floating-point types. T_i is one of the scalar or vector integer or boolean types. T_b 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 component-wise 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`.

6.5 Math Functions

The math functions in Table 6.4 are defined in the header `<metal_math>`. `T` is one of the scalar or vector 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.

`T fmax3(T x, T y, T z)`
`T max3(T x, T y, T z)`
All OS: Since Metal 2.1.
	Returns `fmax(x, fmax(y, z))`.

Built-in Math Functions	**Description**

`T fmedian3(T x, T y, T z)`
All OS: Since Metal 1.0.
`T median3(T x, T y, T z)`
All OS: Since Metal 2.1.
	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.

`T fmin3(T x, T y, T z)`
`T min3(T x, T y, T z)`
All OS: Since Metal 2.1.
	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 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 to nearest even rounding mode in floating-point format.
`T round(T x)`	Return the integral value nearest to `x`, rounding halfway cases away from zero.

Built-in Math Functions	**Description**
`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 to 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 of the math functions listed in Table 6.4: the precise and the fast variants. The `ffast-math` compiler option (refer to section 1.5.3) 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: Constants listed in Table 6.5 and Table 6.6 are supported since Metal 1.2.

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 non-infinite 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 loge10
`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 non-infinite half precision floating-point number.
`HUGE_VALH`	A positive half constant expression. `HUGE_VALH` evaluates to +infinity.

Constant Name	Description
`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 loge10
`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

6.6 Matrix Functions

The functions in Table 6.7 are defined in the header `<metal_matrix>`. `T` is `float` or `half`.

Table 6.7. 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 must be a square matrix.

`floatmxn transpose(floatnxm)`
`halfmxn transpose(halfnxm)`
	Transpose a matrix.

Example:

```
float4x4 mA;
float det = determinant(mA);
```

6.7 Geometric Functions

The functions in Table 6.8 are defined in the header `<metal_geometric>`. `T` is a vector floating-point type (`floatn` or `halfn`). T_s refers to the corresponding scalar type. (If `T` is `floatn`, the scalar type T_s is `float`. If `T` is `halfn`, T_s is `half`.)

Table 6.8. 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` must be a 3-component vector type.
T_s `distance(T x, T y)`	Return the distance between `x` and `y`, which is `length(x-y)`
T_s `distance_squared(T x, T y)`	Return the square of the distance between `x` and `y`.
T_s `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`.
T_s `length(T x)`	Return the length of vector `x`, which is `sqrt(x[0]`^{2} `+ x[1]`^{2} `+ …)`
T_s `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`.
`T refract(T I, T N,` T_s `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` must 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 1.5.3). 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.

6.8 Compute Functions

The compute functions in this section and its subsections are defined in the header `<metal_compute>`. You can only call these compute functions from a `kernel` function.

6.8.1 Threadgroup and SIMD-group Synchronization Functions

Table 6.9 lists supported threadgroup and SIMD-group synchronization functions.

Table 6.9. 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 must execute this function before any thread can continue execution beyond the `threadgroup_barrier`.
`void simdgroup_barrier(mem_flags flags)` macOS: Since Metal 2.0. iOS: Since Metal 1.2.	All threads in a SIMD-group executing the kernel must execute this function before any thread can continue execution beyond the `simdgroup_barrier`.

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.

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) must enter the conditional and execute the barrier function.

If `threadgroup_barrier` (or `simdgroup_barrier`) is inside a loop, for each iteration of the loop, all threads in the threadgroup (or SIMD-group) must 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 (reads and writes) to ensure the correct ordering of memory operations to threadgroup or device memory.

Table 6.10 describes the bit field values for the `mem_flags` argument to `threadgroup_barrier` and `simdgroup_barrier`.

Table 6.10. Memory flag enumeration values for barrier functions

mem_flags	Description
`mem_none`	No memory fence is applied, and `threadgroup_barrier` acts only as an execution barrier.
`mem_device`	Ensure correct ordering of memory operations to device memory.
`mem_threadgroup`	Ensure correct ordering of memory operations to threadgroup memory for threads in a threadgroup.
`mem_texture` macOS: Since Metal 1.2. iOS: Since Metal 2.0.	Ensure correct ordering of memory operations to texture memory for threads in a threadgroup.

6.8.2 SIMD-group Functions

macOS: SIMD-group functions supported since Metal 2.0.

iOS: No support for SIMD-groups.

SIMD-group functions allow threads in a SIMD-group (see section 4.4.1) to share data without the use of threadgroup memory or require any synchronization operations such as a barrier.

An *active* thread is a thread that is executed. An *inactive* thread is a thread that is *not* executed: for example, due to the flow of control or insufficient work to fill the group. The *active* and *inactive* thread terminology also applies to helper threads.

A killed helper thread is inactive. The presence of quad- or SIMD-group operations does not impose any requirement for helper threads to be active or not. You can use `simd_is_helper_thread()` (see Table 6.11) to control helper threads participating in operations.

Threads may only read data from another thread in the SIMD-group that is actively participating. If the target thread is inactive, the retrieved value is undefined.

The SIMD-group functions in Table 6.11 are defined in the header `<metal_simdgroup>` and are supported for kernel and fragment functions. In Table 6.11, `T` is one of the common scalar or vector, integer, or floating-point types (excluding `bool`, `size_t`, `ptrdiff_t`, and `void`), except for bitwise operations, where `T` is restricted to an integer type or a vector of integers.

Table 6.11. SIMD-group functions in the Metal standard library

Built-in SIMD-group Functions	Description
`simd_vote simd_active_threads_mask()` macOS: Since Metal 2.1. iOS: No support.	Returns a mask of active threads (via a `simd_vote`). Functionally equivalent to `simd_ballot(true)`. Bits corresponding to active threads shall be set to 1. Bits corresponding to inactive threads shall be set to 0.
`bool simd_all(bool expr)` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if all active threads evaluate `expr` to `true`.
`T simd_and(T data)` macOS: Since Metal 2.1. iOS: No support.	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_any(bool expr)` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if at least one active thread evaluates `expr` to `true`.
`simd_vote simd_ballot (bool expr)` macOS: Since Metal 2.1. iOS: No support.	Returns a wrapper type (see `simd_vote` details following this table) around a bitmask of the evaluation of the Boolean expression for all active threads in SIMD-group for which `expr` is true. Bits corresponding to inactive threads shall be set to 0.
`T simd_broadcast(T data, ushort broadcast_lane_id)` macOS: Since Metal 2.0. iOS: No support.	Broadcast the value of `data` specified by thread whose SIMD lane ID is `broadcast_lane_id`. If `broadcast_lane_id` is not a valid SIMD lane ID or is not the same for all threads in a SIMD-group, the behavior is undefined.
`T simd_broadcast_first(T data)` macOS: Since Metal 2.1. iOS: No support.	Broadcasts the value of `data` of the first active thread (the active thread with the smallest index) in the SIMD-group to all active threads.
`bool simd_is_first()` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if the current thread is the first active thread (the active thread with the smallest index) in the current SIMD-group; otherwise returns `false`.

Built-in SIMD-group Functions	Description
`bool simd_is_helper_thread()` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if the current thread is a helper thread; otherwise returns `false`. If this is neither called inside a fragment function nor called inside a function called from a fragment function, the behavior is undefined and the call may cause a compile-time error.
`T simd_max(T data)` macOS: Since Metal 2.1. iOS: No support.	Returns the maximum value in `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.
`T simd_min(T data)` macOS: Since Metal 2.1. iOS: No support.	Returns the minimum value in `data` across all active threads in the SIMD-group and broadcasts the result to all active threads in the SIMD-group.
`T simd_or(T data)` macOS: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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. For the first thread in the group, return `T(1)`.
`T simd_prefix_exclusive_sum (T data)` macOS: Since Metal 2.1. iOS: No support.	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. For the first thread in the group, return `T(0)`.
`T simd_prefix_inclusive_product (T data)` macOS: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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.

Built-in SIMD-group Functions	Description
`T simd_shuffle(T data, ushort simd_lane_id)` macOS: Since Metal 2.0. iOS: No support.	Returns the value of data specified by the thread whose SIMD lane ID is `simd_lane_id`. The value of `simd_lane_id` does not have to be the same for all threads in the SIMD-group. The `simd_lane_id` must be a valid SIMD lane ID; otherwise the behavior is undefined.
`T simd_shuffle_down(T data, ushort delta)` macOS: Since Metal 2.0. iOS: No support.	Returns the value of `data` specified by the thread whose SIMD lane ID is `delta` added to the caller's SIMD lane ID. The computed SIMD lane ID does not wrap around the value of the SIMD-group size so the upper `delta` lanes remain unchanged. The value of `delta` must be the same for all threads in a SIMD-group; otherwise the behavior is undefined.
`T simd_shuffle_rotate_down(T data, ushort delta)` macOS: Since Metal 2.1. iOS: No support.	Returns the value of `data` specified by the thread whose SIMD lane ID is `delta` added to the caller's SIMD lane ID. The computed SIMD lane ID wraps around the value of the SIMD-group size. The value of `delta` must be the same for all threads in a SIMD-group; otherwise the behavior is undefined.
`T simd_shuffle_rotate_up(T data, ushort delta)` macOS: Since Metal 2.1. iOS: No support.	Returns the value of `data` specified by the thread whose SIMD lane ID is `delta` subtracted from the caller's SIMD lane ID. The computed SIMD lane ID wraps around the value of the SIMD-group size. The value of `delta` must be the same for all threads in a SIMD-group; otherwise the behavior is undefined.
`T simd_shuffle_up(T data, ushort delta)` macOS: Since Metal 2.0. iOS: No support.	Returns the value of `data` specified by thread whose SIMD lane ID `delta` subtracted from the caller's SIMD lane ID. The computed SIMD lane ID does not wrap around the value of the SIMD-group size so the lower `delta` lanes remain unchanged. The value of `delta` must be the same for all threads in a SIMD-group; otherwise the behavior is undefined.
`T simd_shuffle_xor(T value, ushort mask)` macOS: Since Metal 2.0. iOS: No support.	Returns the value of `data` specified by thread whose SIMD lane ID is a bitwise XOR (`^`) of the caller's SIMD lane ID and `mask`. The value of `mask` must be the same for all threads in a SIMD- group; otherwise the behavior is undefined.

Built-in SIMD-group Functions	**Description**
`T simd_sum(T data)` macOS: Since Metal 2.1. iOS: No support.	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.
`T simd_xor(T data)` macOS: Since Metal 2.1. iOS: No support.	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.

For example, consider the following threadgroup:

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

Here `simd_shuffle_up()` shifts each threadgroup by the `delta` number of threads. If `delta` is 2, the resulting computed SIMD lane IDs shift down by 2, as seen below. Negative values for computed SIMD lane IDs indicate invalid IDs. The computed SIMD lane IDs do not wrap around, so the data for the lower invalid SIMD lane IDs do not change.

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

Similarly `simd_shuffle_down()` shifts down each threadgroup by the `delta` number of threads. Starting from the original threadgroup, if `delta` is 2, the resulting computed SIMD lane IDs shift up by 2, as seen below. Computed SIMD lane IDs greater than the SIMD-group size indicate invalid IDs. The computed SIMD lane IDs do not wrap around, so the data for the upper invalid SIMD lane IDs do not change.

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

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 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_ballot` function uses the `simd_vote` wrapper type (see below), which can be explicitly cast to its underlying type. In the following example, note the use of `vote_t` to represent an underlying type, `XXX.` On macOS, `XXX` is `uint64_t`.

```
class simd_vote {
public:
      typedef XXX vote_t;
      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(bool)`, `simd_vote::any()`, and to the equivalent quad functions listed in section 6.8.3.

On hardware with fewer than 64 threads in a SIMD-group, the value of the top bits in `simd_vote::v` is undefined. In particular, since you can initialize these bits, do not assume that the top bits are set to 0.

6.8.3 Quad-group Functions

macOS: Quad-group functions supported since Metal 2.1.

iOS: Some quad-group functions supported since Metal 2.0, including `quad_broadcast`, `quad_shuffle`, `quad_shuffle_up`, `quad_shuffle_down`, and `quad_shuffle_xor`.

A quad-group function is a SIMD-group function (see section 6.8.2) with an execution width of 4. The *active* and *inactive* thread terminology is the same used in section 6.8.2 for SIMD-group.

Helper threads are executed only to support gradients in quad-groups in a fragment shader. Helper threads may be killed after gradients are computed.

The quad-group functions listed in Table 6.12 are supported for kernel and fragment functions. Threads may only read data from another thread in a quad-group that is actively participating. If the target thread is inactive, the retrieved value is undefined.

In Table 6.12, `T` is one of the scalar or vector integer or floating-point types (excluding `bool`, `size_t`, `ptrdiff_t`, and `void`), except for bitwise operations, where `T` is restricted to an integer type or a vector of integers.

Table 6.12. Quad-group functions in the Metal standard library

Built-in Quad-group Functions	Description
`quad_vote quad_active_threads_mask()` macOS: Since Metal 2.1. iOS: No support.	Returns a mask of active threads (via a `quad_vote`). Functionally equivalent to `quad_ballot(true)`. Bits corresponding to active threads shall be set to 1. Bits corresponding to inactive threads shall be set to 0.
`bool quad_all(bool expr)` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if all active threads evaluate `expr` to `true`.
`T quad_and(T data)` macOS: Since Metal 2.1. iOS: No support	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: Since Metal 2.1. iOS: No support.	Returns `true` if at least one active thread evaluates `expr` to `true`.

Built-in Quad-group Functions	Description
`quad_vote quad_ballot (bool expr)` macOS: Since Metal 2.1. iOS: No support.	Returns a wrapper type (see `quad_vote` details after this table) around a bitmask of the evaluation of the Boolean expression for all active threads in quad-group for which `expr` is true. Bits corresponding to inactive threads shall be set to 0.
`T quad_broadcast(T data, ushort broadcast_lane_id)` macOS: Since Metal 2.0. iOS: Since Metal 2.0.	Broadcast the value of `data` specified by thread whose quad lane ID is `broadcast_lane_id`. If `broadcast_lane_id` is not a valid quad lane ID or is not the same for all threads in a quad-group, the behavior is undefined.
`T quad_broadcast_first(T data)` macOS: Since Metal 2.1. iOS: No support.	Broadcasts the value of `data` of the first active thread (the active thread with the smallest index) in the quad-group to all active threads.
`bool quad_is_first()` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if the current thread is the first active thread (the active thread with the smallest index) in the current quad-group; otherwise returns `false`.
`bool quad_is_helper_thread()` macOS: Since Metal 2.1. iOS: No support.	Returns `true` if the current thread is a helper thread; otherwise returns `false`. If this is neither called inside a fragment function nor called inside a function called from a fragment function, the behavior is undefined and the call may cause a compile-time error.
`T quad_max(T data)` macOS: Since Metal 2.1. iOS: No support.	Returns the maximum value in `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.
`T quad_min(T data)` macOS: Since Metal 2.1. iOS: No support.	Returns the minimum value in `data` across all active threads in the quad-group and broadcasts the result to all active threads in the quad-group.
`T quad_or(T data)` macOS: Since Metal 2.1. iOS: No support.	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.

Built-in Quad-group Functions	**Description**
`T quad_prefix_exclusive_product (T data)` macOS: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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.
`T quad_shuffle(T data, ushort quad_lane_id)` macOS: Since Metal 2.0. iOS: Since Metal 2.0.	Returns the value of data specified by the thread whose quad lane ID is `quad_lane_id`. The value of `quad_lane_id` does not have to be the same for all threads in the quad-group. The `quad_lane_id` must be a valid quad lane ID; otherwise the behavior is undefined.
`T quad_shuffle_down(T data, ushort delta)` macOS: Since Metal 2.0. iOS: Since Metal 2.0.	Returns the value of `data` specified by the thread whose quad lane ID is `delta` added to the caller's quad lane ID. The computed quad lane ID does not wrap around the value of the quad-group size so the upper `delta` lanes remain unchanged. The value of `delta` must be the same for all threads in a quad-group; otherwise the behavior is undefined.

Built-in Quad-group Functions	Description
`T quad_shuffle_rotate_down(T data, ushort delta)` macOS: Since Metal 2.1. iOS: No support.	Returns the value of `data` specified by the thread whose quad lane ID is `delta` added to the caller's quad lane ID. The computed quad lane ID wraps around the value of the quad-group size. The value of `delta` must be the same for all threads in a quad-group; otherwise the behavior is undefined.
`T quad_shuffle_rotate_up(T data, ushort delta)` macOS: Since Metal 2.1. iOS: No support.	Returns the value of `data` specified by the thread whose quad lane ID is `delta` subtracted from the caller's quad lane ID. The computed quad lane ID wraps around the value of the quad-group size. The value of `delta` must be the same for all threads in a quad-group; otherwise the behavior is undefined.
`T quad_shuffle_up(T data, ushort delta)` macOS: Since Metal 2.0. iOS: Since Metal 2.0.	Returns the value of `data` specified by thread whose quad lane ID is `delta` subtracted from from the caller's quad lane ID. The computed quad lane ID does not wrap around the value of the quad-group size so the lower `delta` lanes remain unchanged. The value of `delta` must be the same for all threads in a quad-group; otherwise the behavior is undefined.
`T quad_shuffle_xor(T value, ushort mask)` macOS: Since Metal 2.0. iOS: Since Metal 2.0.	Returns the value of `data` specified by thread whose quad lane ID is a bitwise XOR (`^`) of the caller's quad lane ID and `mask`. The value of `mask` must be the same for all threads in a quad- group; otherwise the behavior is undefined.
`T quad_sum(T data)` macOS: Since Metal 2.1. iOS: No support.	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: Since Metal 2.1. iOS: No support.	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 as follows:

- Lane ID 0: Upper-left pixel
- Lane ID 1: Upper-right pixel
- Lane ID 2: Lower-left pixel
- Lane ID 3: Lower-right pixel

For example, consider the following threadgroup:

Quad Lane ID	**0**	**1**	**2**	**3**
`data`	a	b	c	d

Now consider what happens if `quad_shuffle_up()` shifts each threadgroup by the `delta` number of threads. If `delta` is 2, the computed quad lane IDs move down by 2, as seen below. Negative values for computed quad lane IDs indicate invalid IDs. The computed quad lane IDs do not wrap around, so the data for the lower invalid quad lane IDs remain unchanged.

Computed Quad Lane ID	**-2**	**-1**	**0**	**1**
valid	0	0	1	1
`data`	a	b	a	b

Similarly, `quad_shuffle_down()` shifts down each threadgroup by the `delta` number of threads. Starting from the original threadgroup, if `delta` is 2, the computed quad lane IDs shift up by 2, as seen below. Computed quad lane IDs greater than the quad-group size indicate invalid IDs. The computed quad lane IDs do not wrap around, so the data for the upper invalid SIMD lane IDs remain unchanged.

Computed Quad Lane ID	**2**	**3**	**4**	**5**
valid	1	1	0	0
`data`	c	d	c	d

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). Hence, the non-bottom-four bits are guaranteed to be unset when cast to `vote_t`.

6.9 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 `vertex` function or a `fragment` function.

6.9.1 Fragment Functions

You can only call the functions in this section (listed in Table 6.13, Table 6.14, and Table 6.15) 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 non-helper fragment threads, but do not have side effects that modify the render target(s) 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 target(s).
- 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.

6.9.1.1 Fragment Functions – Derivatives

Metal includes the functions in Table 6.13 to compute derivatives. `T` is one of `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3`, or `half4`.

Derivatives are undefined within non-uniform control flow.

Table 6.13. 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))`

6.9.1.2 Fragment Functions – Samples

Metal includes the per-sample functions listed in Table 6.14. `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 6.14. 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.

6.9.1.3 Fragment Functions – Flow Control

The Metal function in Table 6.15 terminates a fragment.

Table 6.15. 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. If any (but not all) of these threads executes the `discard_fragment` function, the behavior of any derivative computations (explicit or implicit) is undefined.

6.10 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,
- `write` - write to a texture,
- texture query (such as `get_width, get_height, get_num_mip_levels, get_array_size`), and
- texture fence.

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 can 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 2.8.)

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 2.9.

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. (macOS: Support since Metal 2.2; iOS: No support.)

On macOS, for `sample_compare` functions, `bias` and `gradient*` are not supported, and `lod` must be a zero constant.

On macOS, since Metal 2.2, 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 do not take an explicit LOD or gradients have a default LOD of 0. The `gather` and `gather_compare` functions called from kernel or vertex functions 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 are 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, or
- 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:

- zero, for components other than alpha.
- one, 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.

On macOS, for texture `write` functions, `lod` must be 0.

Note:

For sections 6.10.1 through 6.10.16, 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>` used to declare 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`.

6.10.1 1D Texture

The following member function can sample from a 1D texture.

```
Tv sample(sampler s, float coord) const
```

The following member functions can perform sampler-less reads from a 1D texture. Since mipmaps are not supported for 1D textures, `lod` must be `0`.

```
Tv read(uint coord, uint lod = 0) const

Tv read(ushort coord, ushort lod = 0) const // All OS: Since Metal 1.2.
```

The following member functions can write to a 1D texture. Since mipmaps are not supported for 1D textures, `lod` must be `0`.

```
void write(Tv color, uint coord, uint lod = 0)

void write(Tv color, ushort coord,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following member functions query a 1D texture. Since mipmaps are not supported for 1D textures, `get_num_mip_levels()` must return `0`, and `lod` must be `0` for `get_width()`:

```
uint get_width(uint lod = 0) const

uint get_num_mip_levels() const
```

6.10.2 1D Texture Array

The following member function can sample from a 1D texture array:

```
Tv sample(sampler s, float coord, uint array) const
```

The following member functions can perform sampler-less reads from a 1D texture array. Since mipmaps are not supported for 1D textures, `lod` must be `0`.

```
Tv read(uint coord, uint array, uint lod = 0) const
```

```
Tv read(ushort coord, ushort array,
        ushort lod = 0) const // All OS: Since Metal 1.2.
```

The following member functions can write to a 1D texture array. Since mipmaps are not supported for 1D textures, `lod` must be `0`.

```
void write(Tv color, uint coord, uint array, uint lod = 0)
```

```
void write(Tv color, ushort coord, ushort array,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following member functions query a 1D texture array. Since mipmaps are not supported for 1D textures, `get_num_mip_levels()` must return `0`, and `lod` must be `0` for `get_width()`.

```
uint get_width(uint lod = 0) const
```

```
uint get_array_size() const
```

```
uint get_num_mip_levels() const
```

6.10.3 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can 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: Since Metal 1.2.
```

The following member functions can write to a 2D texture. On macOS, `lod` must be `0`.

```
void write(Tv color, uint2 coord, uint lod = 0)

void write(Tv color, ushort2 coord,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following member functions can gather 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
```

The following 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
```

6.10.3.1 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);
```

6.10.4 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can 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: Since Metal 1.2.
```

The following member functions can write to a 2D texture array. On macOS, `lod` must be `0`.

```
void write(Tv color, uint2 coord, uint array, uint lod = 0)
void write(Tv color, ushort2 coord, ushort array,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following member functions can gather 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
```

The following 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
```

6.10.5 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can 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: Since Metal 1.2.
```

The following member functions can write to a 3D texture. On macOS, `lod` must be `0`.

```
void write(Tv color, uint3 coord, uint lod = 0)
void write(Tv color, ushort3 coord,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following 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
```

6.10.6 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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 6.16 describes the cube face and the number used to identify the face.

Table 6.16. Cube face number

Face Number	Cube face
0	Positive X
1	Negative X
2	Positive Y
3	Negative Y
4	Positive Z
5	Negative Z

The following member function can gather four samples for bilinear interpolation when sampling a cube texture:

```
Tv gather(sampler s, float3 coord, component c = component::x) const
```

The following member functions can 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: Since Metal 1.2.
```

The following member functions can write to a cube texture. On macOS, `lod` must be `0`.

```
void write(Tv color, uint2 coord, uint face, uint lod = 0)
void write(Tv color, ushort2 coord, ushort face,
           ushort lod = 0) // All OS: Since Metal 1.2.
```

The following 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
```

6.10.7 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member function can gather four samples for bilinear interpolation when sampling a cube texture array:

```
Tv gather(sampler s, float3 coord, uint array,
          component c = component::x) const
```

The following member functions can 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: Since Metal 1.2.
```

The following member functions can write to a cube texture array. On macOS, `lod` must be `0`.

```
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: Since Metal 1.2.
```

The following 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
```

6.10.8 2D Multisampled Texture

The following member functions can 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: Since Metal 1.2.
```

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.

The following member functions query a 2D multisampled texture:

```
uint get_width() const
uint get_height() const
uint get_num_samples() const
```

6.10.9 2D Multisampled Texture Array

macOS: Since Metal 2.0.

iOS: No support.

The following member functions can perform sampler-less reads from a 2D multisampled texture array:

```
Tv read(uint2 coord, uint array, uint lod = 0) const
Tv read(ushort2 coord, ushort array, ushort lod = 0) const
```

The following member functions query a 2D multisampled 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
```

6.10.10 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can sample from a 2D depth texture and compare a single component against the specified 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. On macOS, the supported `lod_options` values are `level` and `min_lod_clamp` (the latter, since Metal 2.2); `lod` must be a zero constant.

The following member functions can 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: Since Metal 1.2.
```

The following built-in functions can gather four samples for bilinear interpolation when sampling a 2D depth texture:

```
Tv gather(sampler s, float2 coord, int2 offset = int2(0)) const
```

The following member functions can gather 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
```

6.10.11 2D 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)
gradient2d(float2 dPdx, float2 dPdy)
min_lod_clamp(float lod) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can sample from a 2D depth texture array and compare a single component against the specified comparison value:

```
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
```

`T` must be a `float` type. On macOS, the supported `lod_options` values are `level` and `min_lod_clamp` (the latter, since Metal 2.2); `lod` must be a zero constant.

The following member functions can perform sampler-less reads from a 2D depth texture array:

```
T read(uint2 coord, uint array, uint lod = 0) const
```

```
T read(ushort2 coord, ushort array,
       ushort lod = 0) const // All OS: Since Metal 1.2.
```

The following member function can gather 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
```

The following member function can gather four samples for bilinear interpolation when sampling a 2D depth texture array and comparing these samples with a specified comparison value:

```
Tv gather_compare(sampler s, float2 coord, uint array, float compare_value,
                  int2 offset = int2(0)) const
```

`T` must be a `float` type.

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
```

6.10.12 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: Since Metal 1.2.
```

The following member functions query a 2D multisampled depth texture:

```
uint get_width() const
```

```
uint get_height() const
uint get_num_samples() const
```

6.10.13 2D Multisampled Depth Texture Array

macOS: Since Metal 2.0.

iOS: No support.

The following member functions can perform sampler-less reads from a 2D multisampled depth texture array:

```
Tv read(uint2 coord, uint array, uint lod = 0) const
Tv read(ushort2 coord, ushort array, ushort lod = 0) 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
uint get_num_mip_levels() const
```

6.10.14 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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 can 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. On macOS, the supported `lod_options` values are `level` and `min_lod_clamp` (the latter, since Metal 2.2), and `lod` must be a zero constant.

The following member functions can 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: Since Metal 1.2.
```

The following member functions can gather four samples for bilinear interpolation when sampling a cube depth texture:

```
Tv gather(sampler s, float3 coord) const
```

The following member functions can gather 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
```

6.10.15 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) // macOS: support since Metal 2.2; iOS: No support
```

The following member functions can 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
```

The following member functions can sample from a cube depth texture 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. On macOS, the supported `lod_options` values are `level` and `min_lod_clamp` (the latter, since Metal 2.2), and `lod` must be a zero constant.

The following member functions can 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: Since Metal 1.2.
```

The following member function can gather four samples for bilinear interpolation when sampling a cube depth texture:

```
Tv gather(sampler s, float3 coord, uint array) const
```

The following member function can gather 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.

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_array_size() const
```

```
uint get_num_mip_levels() const
```

6.10.16 Texture Buffer Functions

All OS: texture buffers and these functions supported since Metal 2.1.

The following member functions can read from and write to an element in a texture buffer (also see section 2.8.1):

```
Tv read(uint coord) const;
```

```
void write(Tv color, uint coord);
```

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;
```

6.10.17 Texture Synchronization Functions

All OS: Texture synchronization functions supported since Metal 1.2.

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(gid, clr);
      …
      // Use fence to ensure that writes by thread become
      // visible to later reads by the thread.
      texA.fence();

      clr_new = texA.read(gid);
      …
}
```

6.10.18 Null Texture Functions

All OS: null texture functions supported since Metal 1.2.

macOS: null texture functions supported for `texture2d_ms_array` and `depth2d_ms_array` since Metal 2.0.

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 it returns `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>);
// texture2d_ms_array is macOS only, since Metal 2.0
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, since Metal 2.0
bool is_null_texture(depth2d_ms_array<T, access>);
```

The behavior of calling any texture member function with a null texture is undefined.

6.11 Imageblock Functions

macOS: No support for imageblocks.

iOS: imageblocks supported since Metal 2.0.

This section lists the Metal member functions for imageblocks. (For more about the imageblock data type, see sections 2.10 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`.

6.11.1 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_index_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.

6.11.2 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 does not have the argument `size` copies the entire width and height of the imageblock.

6.11.3 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 location given by `coord`.

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 is not 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);
}
```

6.12 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 7.7 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.

6.12.1 Unpack and Convert Integers to a Floating-Point Vector

Table 6.17 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 6.17. 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 r, g, and b color values are converted from sRGB to linear RGB.

`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.

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.

6.12.2 Convert Floating-Point Vector to Integers, then Pack the Integers

Table 6.18 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 6.18. 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.

`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.

6.13 Atomic Functions

The Metal programming language implements a subset of the C++14 atomics and synchronization operations. Metal atomic functions must operate on Metal atomic data, as described in section 2.5.

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>`.

6.13.1 Memory Order

The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations (see section 29.3 of the C++14 specification) and may provide for operation ordering. For atomic operations, `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 since this only requires atomicity, but neither ordering nor synchronization.

6.13.2 Atomic Functions

In addition, accesses to atomic objects may establish interthread synchronization and order non-atomic 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 non-atomic type.
- `M` refers to the type of the other argument for arithmetic operations. For atomic integer types, `M` is `C`.

All OS: Functions listed with names that end with `_explicit` (such as `atomic_store_explicit` or `atomic_load_explicit`) supported since Metal 1.0. unless otherwise indicated.

iOS: Support for the `atomic_store`, `atomic_load`, `atomic_exchange`, `atomic_compare_exchange_weak`, `atomic_fetch_key` functions since Metal 2.0.

6.13.2.1 Atomic Store Functions

These functions atomically replace the value pointed to by `object` with `desired`.

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 2.0.
```

```
void atomic_store_explicit(volatile threadgroup A* object, C desired,
                           memory_order order) // All OS: Since Metal 1.0.
void atomic_store_explicit(device A* object, C desired,
                           memory_order order) // All OS: Since Metal 2.0.
void atomic_store_explicit(volatile device A* object, C desired,
                           memory_order order) // All OS: Since Metal 1.0.
```

6.13.2.2 Atomic Load Functions

These functions atomically obtain the value pointed to by `object`.

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 2.0.
C atomic_load_explicit(const volatile threadgroup A* object,
                       memory_order order) // All OS: Since Metal 1.0.
C atomic_load_explicit(const device A* object,
                       memory_order order) // All OS: Since Metal 2.0.
C atomic_load_explicit(const volatile device A* object,
                       memory_order order) // All OS: Since Metal 1.0.
```

6.13.2.3 Atomic Exchange Functions

These functions atomically replace the value pointed to by `object` with `desired` and return the value `object` previously held.

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 2.0.
C atomic_exchange_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 1.0.
C atomic_exchange_explicit(device A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 2.0.
C atomic_exchange_explicit(volatile device A* object,
                           C desired,
                           memory_order order) // All OS: Since Metal 1.0.
```

6.13.2.4 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`.

Copying is performed in a manner similar to `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,
                    C *expected, C desired, memory_order success,
                    memory_order failure) // All OS: Since Metal 2.0.
bool atomic_compare_exchange_weak_explicit(volatile threadgroup A* object,
                    C *expected, C desired, memory_order success,
                    memory_order failure)  // All OS: Since Metal 1.0.
bool atomic_compare_exchange_weak_explicit(device A* object,
                    C *expected, C desired, memory_order success,
                    memory_order failure)  // All OS: Since Metal 2.0.
bool atomic_compare_exchange_weak_explicit(volatile device A* object,
                    C *expected, C desired, memory_order success,
                    memory_order failure)  // All OS: Since Metal 1.0.
```

6.13.2.5 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 2.0.
C atomic_fetch_key_explicit(volatile threadgroup A* object,
                            M operand,
                            memory_order order) // All OS: Since Metal 1.0.
C atomic_fetch_key_explicit(device A* object,
                            M operand,
                            memory_order order) // All OS: Since Metal 2.0.
C atomic_fetch_key_explicit(volatile device A* object,
                            M operand,
                            memory_order order) // All OS: Since Metal 1.0.
```

The `key` in the function name is a placeholder for an operation name listed in the first column of Table 6.19, such as `atomic_fetch_add_explicit`. The operations detailed in Table 6.19 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 6.19). The function returns the value that `object` held previously. There are no undefined results.

All these functions are applicable to any atomic object.

Table 6.19. 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.

6.14 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.

6.14.1 Encoding Render Commands in Indirect Command Buffers

All OS: Indirect Command Buffers for render commands are supported since Metal 2.1.

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_index_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.

`set_render_pipeline_state(…)` and render pipeline states are only available on macOS.

```
enum class primitive_type { point, line, line_strip, triangle,
                            triangle_strip };
struct render_command {
public:
    explicit render_command(command_buffer icb, unsigned cmd_index);


    void set_render_pipeline_state(render_pipeline_state pipeline_state);


    void set_vertex_buffer(device void *buffer, uint index);
    void set_vertex_buffer(constant void *buffer, uint index);


    void set_fragment_buffer(device void *buffer, uint index);
    void set_fragment_buffer(constant void *buffer, uint index);


    void draw_primitives(primitive_type type, uint vertex_start,
          uint vertex_count, uint instance_count, uint base_instance);
    void draw_indexed_primitives(primitive_type type, uint index_count,
          device/constant ushort/uint *index_buffer,
          uint instance_count, uint base_vertex, uint base_instance);
    void draw_patches(uint number_of_patch_control_points,
          uint patch_start, uint patch_count,
          const device/constant uint *patch_index_buffer,
          uint instance_count, uint base_instance,
          const device/constant
          MTLQuadTessellationFactorsHalf/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/constant uint *patch_index_buffer,
        const device/constant void *control_point_index_buffer,
        uint instance_count,
        uint base_instance,
        const device/constant
        MTLQuadTessellationFactorsHalf/MTLTriangleTessellationFactorsHalf
        *buffer,
        uint instance_stride = 0);


//  Reset the entire command. After reset(), execution of this command
//  does not perform any action until more commands are encoded.
    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.

6.14.2 Encoding Compute Commands in Indirect Command Buffers

iOS: Indirect Command Buffers for compute commands are supported since Metal 2.2.

macOS: No support.

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_index_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);

    void set_kernel_buffer(device void *buffer, uint index);
    void set_kernel_buffer(const device void *buffer, uint index);
    void set_kernel_buffer(constant void *buffer, 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(), execution of this command
//  does not perform any action until more commands are encoded.
    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.

6.14.3 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, but 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 6.14.1 and 6.14.2.

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 is not yet modified.
      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);
}
```

7 Numerical Compliance

This chapter covers how Metal represents floating-point numbers with regard to accuracy in mathematical operations. Metal is compliant to a subset of the IEEE 754 standard.

7.1 INF, NaN, and Denormalized Numbers

INF must be supported for single- and half-precision floating-point numbers.

NaNs must be supported for single- and half-precision 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- or half-precision floating-point numbers passed as input to or produced as the output of single- or half-precision floating-point arithmetic operations may be flushed to zero.

7.2 Rounding Mode

Either round to nearest even or round to zero rounding mode may be supported for single- and half-precision floating-point operations.

7.3 Floating-Point Exceptions

Floating-point exceptions are disabled in Metal.

7.4 Relative Error as ULPs

Table 7.1 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 7.1. 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
`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 7.2 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 `-ffast-math-disable` as a compiler option).

Table 7.2. 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}
`x / y`	<= 2.5 ulp for `y` in the domain of 2^{-126} to 2^{126}

Math Function	Minimum Accuracy (ULP Values)
`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) / log(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` and `y = 0`, is undefined.

`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
`fract`	Correctly rounded
`frexp`	0 ulp
`ilogb`	0 ulp
`ldexp`	Correctly rounded

Math Function	**Minimum Accuracy (ULP Values)**
`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 7.3 describes the minimum accuracy of half-precision floating-point basic arithmetic operations and math functions given as ULP values. Table 7.3 applies to iOS, starting with `MTLGPUFamilyApple4` hardware.

Table 7.3. 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
`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
`fma`	Correctly rounded
`fmax`	0 ulp
`fmin`	0 ulp
`fmod`	0 ulp
`fract`	Correctly rounded
`frexp`	0 ulp

Math Function	Minimum Accuracy (ULP Values)
`ilogb`	0 ulp
`ldexp`	Correctly rounded
`log(x)`	<= 1 ulp
`log2(x)`	<= 1 ulp
`log10(x)`	<= 1 ulp
`modf`	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

Even though the precision of individual math operations and functions are specified in Table 7.1, Table 7.2, and Table 7.3, the Metal compiler, in fast math mode, may 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 non-equal finite floating-point numbers nearest `x`. Moreover, ulp(NaN) is NaN.

7.5 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 for non-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 non-flushed 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.

7.6 Conversion Rules for Floating-Point and Integer Types

The round to zero rounding mode is used for conversions from a floating-point type to an integer type. Use the "round to nearest even" or "round to zero" rounding mode for conversions from a floating-point or integer type to a floating-point type.

The conversions from `half` to `float` are lossless. Conversions from `float` to `half` round the mantissa using the round to nearest 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.

7.7 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. In addition, for the texture `read` and `write` functions, the texture coordinates must refer to a region inside the texture. If the texture coordinates are outside the bounds of the texture, the `read` and `write` function behavior is undefined.

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.

7.7.1 Conversion Rules for Normalized Integer Pixel Data Types

This section discusses converting normalized integer pixel data types to floating-point values and vice-versa.

7.7.1.1 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 7.4. The precision of the conversion rules is guaranteed to be <= 1.5 ulp, except for the cases described in the "Corner Cases" column.

Table 7.4. 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

7.7.1.2 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 7.5.

Table 7.5. 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)` $\mathtt{i}_{0:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
2-bit normalized unsigned integer	`x = min(max(f * 3.0, 0.0), 3.0)` $\mathtt{i}_{1:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
4-bit normalized unsigned integer	`x = min(max(f * 15.0, 0.0), 15.0)` $\mathtt{i}_{3:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
5-bit normalized unsigned integer	`x = min(max(f * 31.0, 0.0), 31.0)` $\mathtt{i}_{4:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
6-bit normalized unsigned integer	`x = min(max(f * 63.0, 0.0), 63.0)` $\mathtt{i}_{5:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
8-bit normalized unsigned integer	`x = min(max(f * 255.0, 0.0), 255.0)` $\mathtt{i}_{7:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
10-bit normalized unsigned integer	`x = min(max(f * 1023.0, 0.0), 1023.0)` $\mathtt{i}_{9:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
16-bit normalized unsigned integer	`result = min(max(f * 65535.0, 0.0), 65535.0)` $\mathtt{i}_{15:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
8-bit normalized signed integer	`result = min(max(f * 127.0, -127.0), 127.0)` $\mathtt{i}_{7:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`
16-bit normalized signed integer	`result = min(max(f * 32767.0, -32767.0),32767.0)` $\mathtt{i}_{15:0}$ `=` $\mathtt{int}_{\mathrm{RTNE}}$`(x)`

In Metal 2.0, all conversions to and from unorm data types shall be correctly rounded.

7.7.2 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 to nearest 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.

7.7.3 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.

7.7.4 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 denormals.

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.

7.7.5 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 E_{max} is the maximum allowed biased exponent value (31).

- Clamp the `r`, `g`, and `b` components (in the process, mapping NaN to zero) as follows:

$$r_c = \max(0, \min(\text{sharedexp}_{max}, r)$$
$$g_c = \max(0, \min(\text{sharedexp}_{max}, g)$$
$$b_c = \max(0, \min(\text{sharedexp}_{max}, b)$$

Where $\text{sharedexp}_{max} = ((2^N - 1)/2^N) * 2(E_{max} - B)$.

- Determine the largest clamped component max_c:

$$\text{max}_c = \max(r_c, g_c, b_c)$$

- Compute a preliminary shared exponent exp_p

$$\text{exp}_p = \max(-B - 1, \text{floor}(\log_2(\text{max}_c)) + 1 + B$$

- Compute a refined shared exponent exp_s

$$\text{max}_s = \text{floor}((\text{max}_c / 2^{\text{exp}_p - B - N}) + 0.5f)$$

$\text{exp}_s = \text{exp}_p$, if $0 <= \text{max}_s < 2^N$, and $\text{exp}_s = \text{exp}_p + 1$, if $\text{max}_s = 2^N$.

- Finally, compute three integer values in the range 0 to $2^N - 1$:

$$r_s = \text{floor}(r_c / 2^{\text{exp}_p - B - N}) + 0.5f)$$
$$g_s = \text{floor}(g_c / 2^{\text{exp}_p - B - N}) + 0.5f)$$
$$b_s = \text{floor}(b_c / 2^{\text{exp}_p - B - N}) + 0.5f)$$

Conversion of a half- or single-precision floating-point color values to the `MTLPixelFormatRGB9E5Float` shared exponent floating-point value is <= 0.5 ULP.

7.7.6 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 7.6.

Table 7.6. 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)`

7.7.7 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. The remaining floating-point(integral) value
//  is converted directly to an integer.
```

The precision of the above conversion shall be:

```
fabs(reference result – integer result) < 1.0.
```