Metal Shading Language Specification

Version 2.0

Developer

1 Introduction

This document describes the Metal Unified Graphics and Compute Language. Metal is a C++ based programming language that developers can use to write code that is executed on the GPU for graphics and general-purpose data-parallel computations. Since Metal is based on C++, developers will find it familiar and easy to use. With Metal, both graphics and compute programs can be written with a single, unified language, which allows tighter integration between the two.

Metal is designed to work together with the Metal framework, which manages the execution, and optionally the compilation, of Metal code. Metal uses clang and LLVM so developers get a compiler that delivers close to the metal performance for code executing on the GPU.

1.1 Audience

Developers who are writing code with the Metal framework will want to read this document, because they will need to use the Metal shading language to write graphics and compute programs to be executed 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 and covers the similarities and differences between Metal and C++14.
- "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.
- "Function and Variable Declarations" details how functions and variables are declared, sometimes with attributes that restrict how they are used.
- "Metal Standard Library" defines a collection of built-in Metal functions.
- "Compiler Options" details the options for the Metal compiler, including pre-processor directives, options for math intrinsics, and options that control optimization.
- "Numerical Compliance" describes requirements for representing floating-point numbers, including accuracy in mathematical operations.

1.3 References

C++14

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

Metal

An overview of the official Metal documentation is located here:

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

1.4 Metal and C++14

The Metal programming language is based on the C++14 Specification (a.k.a., the ISO/IEC JTC1/SC22/WG21 N4431 Language Specification) with specific extensions and restrictions. Please refer to the C++14 Specification for a detailed description of the language grammar.

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

For more information about Metal pre-processing directives and compiler options, see section 6 of this document.

1.4.1 Overloading

Metal supports overloading as defined by section 13 of the C++14 Specification. The function overloading rules are extended to include the address space attribute of an argument. Metal graphics and kernel functions cannot be overloaded. (For definition of graphics and kernel functions, see section 4.1 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 pre-processing directives 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)
- dynamic_cast operator (section 5.2.7)
- type identification (section 5.2.8)
- recursive function calls (section 5.2.2, item 9)
- 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)

The C++ standard library must **not** be used in Metal code. Instead of the C++ standard library, Metal has its own standard library that is discussed in Chapter 5 of this document.

Metal restricts the use of pointers:

- Arguments to Metal graphics and kernel functions declared in a program that are pointers must be declared with the Metal `device`, `threadgroup`, `threadgroup_imageblock`, or `constant` address space attribute. (See section 4.2 of this document for more about Metal address space attribute.)
- Function pointers are not supported.

A Metal function cannot be called `main`.

1.5 Metal Pixel Coordinate System

In Metal, the origin of the pixel coordinate system of a framebuffer attachment is defined at the top left corner. Similarly, the origin of the pixel coordinate system of a framebuffer attachment is the top left corner.

2 Data Types

This chapter details the Metal data types, including types that represent vectors and matrices. Atomic data types, buffers, textures, samplers, arrays, and user-defined structs are also discussed. Type alignment and type conversion are also described.

2.1 Scalar Data Types

Metal supports the scalar types listed in Table 1. Metal does **not** support the `double`, `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double` data types.

Table 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`
`unit16_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.

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

Type	Description
`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.

NOTE: Metal supports the standard `f` or `F` suffix to specify a single precision floating-point literal value (e.g., `0.5f` or `0.5F`). In addition, Metal supports the `h` or `H` suffix to specify a half precision floating-point literal value (e.g., `0.5h` or `0.5H`). Metal also supports the `u` or `U` suffix for unsigned integer literals.

Table 2 lists the size and alignment of most of the scalar data types.

Table 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`
`unit16_t`
	2	2

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

`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.

The vector type names supported are:

`booln`,

`charn`, `shortn`, `intn`, `ucharn`, `ushortn`, `uintn`,

`halfn` and `floatn`

`n` is 2, 3, or 4 representing a 2-, 3- or 4- component vector type. Table 3 lists the size and alignment of the vector data types.

Table 3 Size and Alignment of Vector Data Types

Type	Size (in bytes)	Alignment (in bytes)
`bool2`	2	2
`bool3`	4	4
`bool4`	4	4
`char2` `uchar2`	2	2
`char3` `uchar3`	4	4
`char4` `uchar4`	4	4
`short2` `ushort2`	4	4
`short3` `ushort3`	8	8
`short4` `ushort4`	8	8
`int2` `uint2`	8	8
`int3` `uint3`	16	16
`int4` `uint4`	16	16
`half2`	4	4
`half3`	8	8
`half4`	8	8
`float2`	8	8

Type	Size (in bytes)	Alignment (in bytes)
float3	16	16
float4	16	16

2.2.1 Accessing Vector Components

Vector components can be accessed using an array index. 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 the 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
```

In the following code, the vector test is initialized, and then components are accessed using the `.xyzw` or `.rgba` selection syntax:

```
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 multiple components to be selected.

```
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 components to be permuted or replicated.

```
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 of an expression. To form the lvalue, swizzling may be applied. 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. For instance:

  ```
  float2 pos;
  pos.x = 1.0f; // is legal; so is y
  pos.z = 1.0f; // is illegal; so is w
  ```

```
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

- Accessing the same component twice on the left-hand side is ambiguous; for instance,

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

// illegal - mismatch between float2 and float4
pos.xy = float4(1.0f, 2.0f, 3.0f, 4.0f);
```

- The `.rgba` and `.xyzw` attributes cannot be intermixed in a single access; for instance,

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
pos.x = 1.0f;     // OK
pos.g = 2.0f;     // OK
pos.xg = float2(3.0f, 4.0f); // illegal - mixed attributes used
float3 coord = pos.ryz;  // illegal - mixed attributes used
```

- A pointer or reference to a vector with swizzles; for instance

```
float4 pos = float4(1.0f, 2.0f, 3.0f, 4.0f);
my_func(&pos.xy); // illegal
```

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

Constructors can be used to create vectors from a set of scalars or vectors. When a vector is initialized, its parameter signature determines how it is constructed. 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 a vector is constructed from multiple scalars, one or more vectors, or a mixture of these, 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 complete list of constructors that are available 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 complete list of constructors that are available 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 complete list of constructors that are available for `float2`:

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

The following examples illustrate uses of the constructors:

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

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

Under-initializing a vector constructor is a compile-time error.

2.2.3 Packed Vector Types

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

The packed vector type names supported are:

`packed_charn`, `packed_shortn`, `packed_intn`,

`packed_ucharn`, `packed_ushortn`, `packed_uintn`,

`packed_halfn` and `packed_floatn`

`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 4 lists the size and alignment of the packed vector data types.

Table 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

Type	Size (in bytes)	Alignment (in bytes)
packed_float2	8	4
packed_float3	12	4
packed_float4	16	4

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

Examples:

```
device float4 *buffer;
device packed_float4 *packed_buffer;

int i;
packed_float4 f ( buffer[i] );
pack_buffer[i] = buffer[i];

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

Components of a packed vector data type can be accessed with an array index. However, components of a packed vector data type cannot be accessed with the `.xyzw` or `.rgba` selection syntax.

Example:

```
packed_float4 f;
f[0] = 1.0f;  // OK
f.x = 1.0f;   // Illegal - compilation error
```

2.3 Matrix Data Types

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

The matrix type names supported are:

`halfnxm` and `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 5 lists the size and alignment of the matrix data types.

Table 5 Size and Alignment of Matrix Data Types

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

2.3.1 Accessing Matrix Components

The components of a matrix can be accessed using the array subscripting syntax. Applying a single subscript to a matrix treats the matrix as an array of column vectors. The top column is column 0. A second subscript would then operate on the resulting vector, as defined earlier for vectors. Hence, two subscripts select a column and then a row.

```
float4x4 m;

// sets the 2nd column to all 2.0
m[1] = float4(2.0f);

// sets the 1st element of the 1st column to 1.0
m[0][0] = 1.0f;

// sets the 4th element of the 3rd column to 3.0
m[2][3] = 3.0f;
```

The `floatnxm` and `halfnxm` matrices can be accessed 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

Constructors can be used to create matrices from a set of scalars, vectors or matrices. When a matrix is initialized, its parameter signature determines how it is constructed. For example, if a matrix is initialized 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
```

A matrix can also be constructed from another matrix that is of the same size; i.e., 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);
```

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

The `alignas` alignment specifier can be used to specify the alignment requirement of a type or an object. The `alignas` specifier may be applied to the declaration of a variable or a data member of a struct or class. It may also be applied to the declaration of a struct, 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 can assume that the pointee 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.

For all versions of Metal, these atomic types are supported:

```
atomic_int
atomic_uint
```

For `ios-metal2.0`, these atomic types are also supported `atomic_bool`

`atomic<T>`

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

The Metal atomic data type is restricted for use by Metal atomic functions, as described in section 5.12. These atomic functions are a subset of the C++14 atomic and synchronization functions.

2.6 Pixel Data Types

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. The pixel data types are generally available in all address spaces (for details on address spaces, see section 4.2).

Table 6 lists supported pixel data types in the Metal shading language, as well as their size and alignment.

Table 6 Metal Pixel Data Types

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

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

Examples:

```
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;
}

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

Metal implements buffers as a pointer to a built-in or user defined data type described in the `device`, `threadgroup`, or `constant` address space. (Refer to section 4.2 for a full description of these address attributes.). These buffers can be declared in program scope or passed as arguments to a function.

Examples:

```
device float4 *device_buffer;

struct my_user_data {
      float4 a;
      float b;
      int2 c;
};

constant my_user_data *user_data;
```

Ordinary Metal buffers may contain:

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

For 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>
```

Textures with depth formats must be declared 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>
```

`T` specifies the color type returned when reading from a texture or the color type 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`.

NOTE: 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.

The `access` attribute describes how the texture can be accessed. The supported access attributes are:

- `sample` – The texture object can be sampled. `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.

NOTE:

- **For multisampled textures, only the `read` attribute is supported.**
- **For depth textures, only the `sample` and `read` attributes are supported.**

The following example uses these access attributes 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)]])
{…}
```

(See section 4.3.1 for a description of the `texture` attribute.)

A texture type can also be used 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 but which do not use `access::read` or `access:sample` attributes 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; // legal
    texture2d<float, access::read> y = imgB; // legal
    texture2d<float, access::write> z; // illegal
    …
}
```

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. A sampler object can also be described in the program source instead of in the API. For these cases we only allow a subset of the sampler state to be specified: the addressing mode, filter mode, normalized coordinates and comparison function.

Table 7 describes the list of supported sampler state enums and their associated values (and defaults). These states can be specified when a sampler is initialized in Metal program source.

Table 7 Sampler State Enumeration Values

Enum Name	Valid Values	Description
`coord`	`normalized` (default) `pixel`	Specifies whether the texture coordinates when sampling from a texture are normalized or unnormalized 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.

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

Enum Name	Valid Values	Description
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 that is used by the sample_compare and gather_compare texture functions.

clamp_to_border is only available on macOS. 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).

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 specified as the address mode for one or more texture coordinates, the other texture coordinates can use an address mode of clamp_to_border if and only if the border color is transparent_black. Otherwise the behavior is undefined.

The enumeration types used by the sampler data type as described in Table 7 are specified as follows:

```
enum class coord       { normalized, pixel };
enum class filter      { nearest, linear };
enum class min_filter  { nearest, linear };
enum class mag_filter  { nearest, linear };
enum class mip_filter  { none, nearest, linear };

enum class s_address   { clamp_to_zero, clamp_to_edge, clamp_to_border
                         repeat, mirrored_repeat };
enum class t_address   { clamp_to_zero, clamp_to_edge, clamp_to_border
                         repeat, mirrored_repeat };
enum class r_address   { clamp_to_zero, clamp_to_edge, clamp_to_border
```

```
                          repeat, mirrored_repeat };
enum class address      { clamp_to_zero, clamp_to_edge, clamp_to_border
                          repeat, mirrored_repeat };

enum class compare_func { none, less, less_equal, greater, greater_equal,
                          equal, not_equal };

enum class border_color { transparent_black, opaque_black, opaque_white };
```

`border_color` is only valid if the `address` mode is `clamp_to_border`.

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, the following types can also be specified with a sampler:

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

Metal implements the sampler objects as follows:

```
struct sampler {
      public:
            // full version of sampler constructor
            template<typename… Ts>
            constexpr sampler(Ts… sampler_params){};
      private:
};
```

`Ts` must be the enumeration types listed above that can be used by the sampler data type. If the same enumeration type is declared multiple times in a given sampler constructor, the last listed value takes effect.

The following Metal program source illustrates several ways to declare samplers. (The attributes (`sampler(n)`, `buffer(n)`, and `texture(n)`) that appear in the code below are explained in section 4.3.1.). Note that samplers or constant buffers declared in program source do not need these attribute attributes.

```
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));
```

NOTE: Samplers that are initialized in the Metal shading language source are declared with `constexpr`.

2.10 Imageblocks

The imageblock is a 2-dimensional data structure (represented by width, height, and number of samples) allocated in threadgroup memory that is designed as an efficient mechanism for processing 2-dimensional image data. The data layout of the imageblock is opaque. The elements in the imageblock can be accessed using an (x, y) coordinate and optionally the sample index. The elements in the imageblock associated with a specific (x, y) are referred to as the per-thread imageblock data or as just the imageblock data.

Imageblocks are only supported for `ios-metal2.0`. Imageblocks are only used with fragment and kernel functions.

For fragment functions, the imageblock dimensions are derived from the tile size. For kernel functions, the developer specifies the imageblock dimensions, which are typically derived from the threadgroup size. For fragment functions, only the fragment's imageblock data (i.e., identified by the fragment's pixel position in the tile) can be accessed. For kernel functions, all threads in the threadgroup can access the imageblock.

The imageblock data is described as a struct. Each element of the struct can be a scalar or vector integer or floating-point data type, pixel data types (specified in Table 6 in section 2.6), an array of these types, or structs built using these types.

Built-in functions for imageblocks are listed in section 5.15.

An imageblock *slice* refers to a region in the imageblock that describes the values of a given element in the imageblock data struct 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 8.

Table 8 Imageblock Data Type and Compatible Target Texture Format

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

Pixel Storage Type	Compatible Texture Formats
`rgb10a2<T>`	`RGB10A2Unorm`
`rg11b10f<T>`	`RG11B10Float`
`rgb9e5<T>`	`RGB9E5Float`

Sections 2.10.1 and 2.10.2 describe how an imageblock is accessed in a fragment or kernel function, respectively.

2.10.1 Imageblocks in Fragment Functions

In a fragment function, the imageblock can be accessed in two ways:

- as a color attachment where the storage layout of the imageblock is not known in the fragment function (*implicit imageblock layout*). An implicit imageblock layout uses the existing color attachment attribute. See section 2.10.1.1.
- or as a struct used to declare the imageblock data when the storage layout of the imageblock is explicitly specified in the fragment function (*explicit imageblock layout*). See section 2.10.1.2.

2.10.1.1 Implicit Imageblock Layout for Fragment Functions

The imageblock data (i.e., all the data members in the imageblock associated with a pixel) can be accessed 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 (i.e., the types used to perform computations in the fragment function). The Metal runtime defines the actual storage (i.e., pixel) format to be used.

When accessing the imageblock data as color attachments, the pixel storage types described in section 2.6 cannot be declared in the imageblock slice struct.

For an imageblock data implicit layout of type `T`, `T` is a struct where each member satisfies one of the following:

- have a color attachment (see the `[[color(m)]]` attribute in Table 12 of section 4.3.4). The color index `m` must be unique for each member (and sub-member) of `T`.
- be a struct type with members that satisfy the constraint on the list.

2.10.1.2 Explicit Imageblock Layout for Fragment Functions

The imageblock data with explicit layout (i.e., the imageblock layout is declared in the shading function, not via the runtime as done for color attachments) is declared as a struct.

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 struct built with these types. The data members of the imageblock struct use the appropriate alignment rules for each data member type declared in the struct to determine the actual struct 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. The input and output imageblock data to a fragment function can be declared as a struct. The input and output imageblock structs can be the fully explicit imageblock struct (referred to as the master explicit imageblock struct) or be a subset of the master explicit imageblock struct (referred to as the imageblock view struct). For the latter case, the `[[imageblock_data(type)]]` attribute must be used with the input and output imageblock data struct specified on a fragment function, where `type` specifies the fully explicit imageblock data struct.

If the `[[imageblock_data(type)]]` attribute is specified on the input argument or output struct element without type, the fragment function is assumed to use the master explicit imageblock data struct 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 struct.

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 4.7.3.

2.10.2 Imageblocks 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 user functions 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 struct, and
- each member of `T` may 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 struct with members that are one of the types on this list

For an imageblock with implicit layout (`imageblock_layout_implicit`), each member of the struct may have a color attachment (see the `[[color(m)]]` attribute in Table 12 of section 4.3.4). The color index `m` must be unique for each member (and sub-member) of `T`.

If no imageblock layout is specified by the user, the compiler deduces the layout based on `T`. If `T` is not compatible with an implicit or explicit imageblock, a compiler error is generated.

Both explicit and implicit imageblocks can be passed as arguments to a kernel function. This also makes it easy to share explicit and implicit imageblock structs 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 4.7.3.

2.11 Aggregate Types

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

Do not specify a struct 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. (See section 4.2 for details on address spaces.)

Unless used in an Argument Buffer (see section 2.12), the following restriction is applicable:

- An `array<T, N>` type used to declare an array of textures (see section 2.11.1) cannot be declared in a struct.

2.11.1 Arrays of Textures and Samplers

An array of textures is declared as:

`array<typename T, size_t N>` or

`const array<typename T, size_t N>`

`T` must be a texture type described in section 2.8 declared with the `access::read` or `access::sample` attribute.

An array of samplers is declared as:

`array<sampler, size_t N>` or

`const array<sampler, size_t N>`

An array of textures or an array of samplers can be passed as an argument to a function (graphics, kernel, or user function) or be declared as a local variable inside a function. An array of samplers can also be declared in program scope.

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. 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 Element Access

Elements of an array of textures or an array of samplers can be accessed using the `[]` operator. The following variants of the `[]` operator are available:

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

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

Elements of a templated type `array_ref<T>` can also be accessed using the following variant of the `[]` operator:

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

2.11.1.2 Capacity

```
constexpr size_t size();

constexpr size_t size() const;
```

Returns the number of elements in the array or the `array_ref<T>`.

Examples:

```
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:

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

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

```
}

kernel void
my_kernel_B(const array<texture2d<float>, 20> src [[texture(0)]],
            texture2d<float, access::write> dst [[texture(10)]],
             …)
{
    float4 clr = foo(src);
    …
}
```

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

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

2.12 Argument Buffers

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

```
struct Foo {
     texture2d<float, access::write> a; depth2d<float> b;
     sampler sam c;
     texture2d<float> d;
     device float4* e;
     texture2d<float> f;
     int g;
};
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 example below, index 0 is automatically assigned to `foo1`. The `[[id(n)]]` attribute specifies the index offsets for the `t1` and `t2` struct members. Since no index is specified for `foo2`, it is automatically assigned the next index, 4, which is determined by adding 1 to the maximum ID used by the previous struct 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 the `[[id]]` attribute is omitted, an ID is automatically assigned according to the following rules:

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

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.

Argument buffers can be accessed 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, resources can be copied between structs 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

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; i.e., 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 = texList[i].sample(s, float2(coord.x, coord.y));
     …;
     texOutput[i].write(color, coord);
}
```

If the variable `i` has the same value for all threads (i.e., 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 (i.e., boolean, integer, and floating point types) 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 = texList[i].sample(s, float2(coord.x, coord.y));
```

```
    …;
    texOutput[i].write(color, coord);
}
```

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 = …;
intz=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 (i.e., when converting to floating-point, round to nearest even; 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:

```
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 will have 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

This chapter lists and describes the Metal operators.

3.1 Scalar and Vector Operators

1. The arithmetic operators, add (+), subtract (-), multiply (*) and divide (/), operate on scalar and vector, integer and floating-point data types. All arithmetic operators return a result of the same built-in type (integer or floating-point) as the type of the operands, after operand type conversion. After conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
- One operand is a scalar, and the other is a vector. In this case, 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.
- The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

Division on integer types that results 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 ±infinity or NaN, as prescribed by the IEEE-754 standard. (For details about numerical accuracy of floating-point operations, see section 7.)

2. The operator modulus (%) operates on scalar and vector integer data types. All arithmetic operators return a result of the same built-in type as the type of the operands, after operand type conversion. The following cases are valid:

- The two operands are scalars. In this case, the operation is applied, and the result is a scalar.
- One operand is a scalar, and the other is a vector. In this case, 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.
- The two operands are vectors of the same size. In this case, the operation is performed component-wise, which results in a same size vector.

The resulting value is undefined for any component computed with a second operand that is zero, while results for other components with non-zero operands remain defined. If both operands are non-negative, the remainder is non-negative. If one or both operands are negative, results are undefined.

3. The arithmetic unary operators (+ and -) operate on scalar and vector, integer and floating-point types.

4. The arithmetic post- and pre-increment and decrement operators (-- and ++) operate on scalar and vector integer types. All unary operators work component-wise on their operands. The result is the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 to the contents of the expression they operate on, 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 they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.
5. The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate on scalar and vector, integer and floating-point types. The result is a Boolean (`bool` type) scalar or vector. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a `bool`.
- One operand is a scalar, and the other is a vector. In this case, 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 Boolean vector.
- The two operands are vectors of the same type. In this case, the operation is performed component-wise, which results in a Boolean vector.

The relational operators always return `false` if either argument is a NaN. To test a relational operation on any or all elements of a vector, use the `any` and `all` built-in functions (defined in section 5.4) in the context of an `if(…)` statement.

6. The equality operators, equal (==) and not equal (!=), operate on scalar and vector, integer and floating-point types. All equality operators result in a Boolean (`bool` type) scalar or vector. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a `bool`.
- One operand is a scalar, and the other is a vector. In this case, 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, resulting in a Boolean vector.
- The two operands are vectors of the same type. In this case, the operation is performed component-wise resulting in a same size Boolean vector.

All other cases of implicit conversions are illegal. If one or both arguments is "Not a Number" (NaN), the equality operator equal (==) returns `false`. If one or both arguments is "Not a Number" (NaN), the equality operator not equal (!=) returns `true`.

7. The bitwise operators and (&), or (|), exclusive or (^), not (~) operate on all scalar and vector built-in types except the built-in scalar and vector floating-point types. For built-in vector types, the 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 operation is performed component-wise resulting in a same size vector.

8. The logical operators and (&&), or (||) operate on two Boolean expressions. The result is a scalar or vector Boolean.
9. The logical unary operator not (!) operates on a Boolean expression. The result is a scalar or vector Boolean.
10. The ternary selection operator (?:) operates on three expressions (`exp1` ? `exp2` : `exp3`). This operator evaluates the first expression `exp1`, which must result in a scalar Boolean. If the result is `true`, it selects to evaluate the second expression; otherwise it evaluates the third expression. Only one of the second and third expressions is evaluated. The second and third expressions can be any type, as long their types match, or there is a conversion in section 2.11 that can be applied to one of the expressions to make their types match, or one is a vector and the other is a scalar in which case the scalar is widened to the same type as the vector type. This resulting matching type is the type of the entire expression.
11. The ones' complement operator (~). The operand must be of a scalar or vector integer type, and the result is the ones' complement of its operand.

The operators right-shift (>>), left-shift (<<) operate on all scalar and vector integer types. For built-in vector types, the operators are applied component-wise. For the right-shift (>>), 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 `log2(N)` least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the data type of `E1`, if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. The vacated bits are filled with zeros.

The result of `E1 >> E2` is `E1` right-shifted by `log2(N)` least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the data type of `E1`, if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. 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.

NOTE:

- **Operators not described above that are supported by C++14 (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 (*) operates on:

- a scalar and a matrix,
- a matrix and a scalar,
- a vector and a matrix,
- a matrix and a vector,
- or 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 – matrix, matrix – vector and matrix – matrix multiplication, the number of columns of the left operand is required to be equal to the number of rows of the right operand. The multiply operation does a linear algebraic multiply, yielding a vector or a matrix that has the same number of rows as the left operand and the same number of columns as the right operand.

The examples below presume these vector, matrix, and scalar variables are initialized:

```
float3 v;
float3x3 m;
float a = 3.0f;
```

The following 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 following 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 following matrix-to-vector multiplication

```
float3 u = m * v;
```

is equivalent to:

```
u = v.x * m[0];
u += v.y * m[1];
u += v.z * m[2];
```

The following matrix-to-matrix multiplication

```
float3x3   m, n, r;
r = m * n;
```

is equivalent to:

```
r[0] = m[0] * n[0].x;
r[0] += m[1] * n[0].y;
r[0] += m[2] * n[0].z;

r[1] = m[0] * n[1].x;
r[1] += m[1] * n[1].y;
r[1] += m[2] * n[1].z;

r[2] = m[0] * n[2].x;
r[2] += m[1] * n[2].y;
r[2] += m[2] * n[2].z;
```

NOTE: The order of partial sums for the vector-to-matrix, matrix-to-vector and matrix-to-matrix multiplication operations described above is undefined.

4 Function and Variable Declarations

This chapter describes how functions, arguments, and variables are declared. It also details how attributes are often used with functions, arguments, and variables to specify restrictions.

4.1 Function Specifiers

Metal supports the following function specifiers that restrict how a function may be used:

- `kernel` - A data-parallel function that is executed over a 1-, 2- or 3-dimensional grid.
- `vertex` - A vertex function that is executed for each vertex in the vertex stream and generates per-vertex output or a post-tessellation vertex function that is executed for each surface sample on the patch produced by the fixed-function tessellator.
- `fragment` – A fragment function that is executed for each fragment in the fragment stream and their associated data and generates per-fragment output.

A function specifier is used at the start of a function, before its return type. The following example shows the syntax for a compute function.

```
kernel void
my_kernel(…)
{…}
```

For functions declared with the `kernel` specifier, the return type must be `void`.

Only a graphics function can be declared with one of the `vertex` or `fragment` specifiers. For graphics functions, the return type identifies whether the output generated by the function is either per-vertex or per-fragment. The return type for a graphics function may be `void` indicating that the function does not generate output.

Functions that use a `kernel`, `vertex` or `fragment` function specifier cannot call functions that also use these specifiers, or a compilation error results.

Functions that use a `kernel`, `vertex` or `fragment` function specifier can be declared within a namespace.

4.1.1 Post-Tessellation Vertex Functions

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:

- the per-patch data,
- the patch control point data, and
- 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 specifier.

4.1.1.1 Patch Type and Number of Control Points Per-Patch

The `[[patch]]` specifier is required for the post-tessellation vertex function.

For macOS, the `[[patch(patch-type, N)]]` specifier 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(…)
{…}
```

4.1.2 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 is accumulated. 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. Threads may be launched 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.

A tile function is only supported for `iOS-metal2.0`. iOS GPUs always process each tile and each dispatch to completion. All draws and dispatches for a tile will launch in submission before the next tile is processed.

Tile functions have access to 32KB of threadgroup memory that may be divided between imageblock storage and threadgroup storage. (For details on the `threadgroup_imageblock` address space, see section 4.2.3.) The imageblock size is dependent on the tile width, tile height, and the bit depth of each sample. The bit depth is determined either by the render pass attachments (see implicit imageblock layout in section 2.10.1.1) or in function-declared structures (see explicit imageblock layout in section 2.10.1.2). For a detailed description of how the `threadgroup_imageblock` address space is used in kernel functions, refer to section 4.2.3.

4.1.3 Fragment Function Specifier

The `[[early_fragment_tests]]` function specifier can be used with a fragment function to request that fragment tests be performed before fragment function execution.

Below is an example of a fragment function that uses this specifier:

```
[[early_fragment_tests]]

fragment float4

my_fragment( … )

{…}
```

NOTE:

- **It is an error if the return type of the fragment function declared with the `[[early_fragment_tests]]` specifier includes a depth value i.e. 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]]` specifier with any function that is not a fragment function i.e. not declared with the `fragment` specifier.**

4.2 Address Space Attributes for Variables and Arguments

The Metal shading language implements address space attributes to specify the region of memory where a function variable or argument is allocated. These attributes describe disjoint address spaces for variables:

- `device` (for more details, see section 4.2.1)

- `threadgroup` (see section 4.2.2)
- `threadgroup_imageblock` (see section 4.2.3)
- `constant` (see section 4.2.4)
- `thread` (see section 4.2.5)

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 4.2.2.)

```
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.

4.2.1 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 struct. The actual size of the buffer memory object is determined when the memory object is allocated via appropriate Metal API calls in the host code.

Some examples are:

```
// an array of a float vector with 4 components
device float4 *color;

struct Foo {
      float a[3];
      int b[2];
}
```

```
// an array of Foo elements
device Foo *my_info;
```

Since texture objects are always allocated from the device address space, the `device` address attribute is not needed for texture types. The elements of a texture object cannot be directly accessed. Functions to read from and write to a texture object are provided.

4.2.2 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. Variables declared in the `threadgroup` address space **cannot** be used in graphics functions.

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 4.3.1.)

```
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];
      …
}
```

4.2.2.1 SIMD-groups and Quad-groups

Within a threadgroup, threads can be divided into **SIMD-groups** in an implementation- defined fashion. Each SIMD-group is a collection of threads that executes 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 non-uniform 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.

SIMD-groups are only supported for `macos–metal2.0`. Quad-groups are only supported on

`ios–metal2.0`.

For kernel function attributes SIMD-groups and quad-groups, see section 4.3.4.6. SIMD- group functions are described in section 5.13. Quad-group functions are described in section 5.14.

4.2.3 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 4.1.2). 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 is executing 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.

4.2.4 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.

NOTE: 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.

4.2.5 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;
    …
}
```

4.3 Function Arguments and Variables

Most inputs and outputs to a graphics and kernel functions are passed as arguments. (Exceptions are initialized variables in the constant address space and samplers declared in program scope.) Arguments to graphics (vertex and fragment) and kernel functions can be one of the following:

- device buffer – a pointer or reference to any data type in the `device` address space (see section 2.6)
- constant buffer – a pointer or reference to any data type in the `constant` address space (see section 2.6)
- `texture` object (see section 2.8) or an array of textures
- `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.
- imageblock (see section 2.10)
- argument buffer (see section 2.12)
- A struct with elements that are either buffers or textures.

Buffers (device and constant) specified as argument values to a graphics or kernel function cannot alias; i.e., a buffer passed as an argument value cannot overlap another buffer passed to a separate argument of the same graphics or kernel function.

Arguments to graphics and kernel functions cannot be declared to be of type `size_t`, `ptrdiff_t`, or a struct 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 4.3.1),
- built-in variables that support communicating data between fixed-function and programmable pipeline stages (see section 4.3.3),
- which data is sent down the pipeline from vertex function to fragment function (see section 4.3.5).

4.3.1 Attributes to Locate Buffers, Textures and Samplers

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)]]`
- texture and an array of textures – `[[texture(index)]]`
- sampler – `[[sampler(index)]]`
- threadgroup buffer – `[[threadgroup(index)]]`

The `index` value is an unsigned integer that identifies the location of a buffer, texture or sampler argument that is being assigned. The proper syntax is for the attribute to follow the argument/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)]])
{…}
```

NOTE: For buffers (device, constant and threadgroup), textures or samplers the `index` value that is used to specify the buffer, texture or sampler index must be unique. 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 4.10.1).

4.3.1.1 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 4.2.1). All the `render_vertex` function arguments are specified with the `buffer(0)`, `buffer(1)`, `buffer(2)`, and `buffer(3)` attributes (introduced in section 4.3.1). The `position` attribute shown in this example is discussed in section 4.3.3.

```
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 position to a buffer
      xform_output[v_id] = v_out;
}
```

4.3.1.2 Raster Order Groups

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 struct 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. So 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.load(coord);
      // do operations on clr
      clr = …;
      texA.write(clr, coord);
}
```

For an argument buffer, the `[[raster_order_group(index)]]` attribute can be used on a buffer or texture member in a struct.

4.3.2 Struct of Buffers and Textures

Arguments to a graphics, kernel or user function can be a struct or a nested struct whose members are buffers, textures or samplers only. Such a struct must be passed by value. Each member of such a struct passed as the argument type to a graphics or kernel function can have an attribute to specify its location (as described in section 4.3.1).

Example of a struct passed as an argument:

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

kernel void
my_kernel(Foo f)
{…}
```

Below are some examples of invalid use cases that should result in a compilation error.

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

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

```
kernel void
my_kernel(MyResources r) // illegal use
{…}
```

Nested structs are also supported as shown by 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)
{…}
```

4.3.3 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, per-vertex inputs can also be passed as an argument to a vertex function by declaring them with the `[[stage_in]]` attribute. For per-vertex inputs 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)]]`.

The `index` value is an unsigned integer that identifies the vertex input location that is being assigned. The proper syntax is for the attribute to follow the argument/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 example below shows how vertex attributes can be assigned to elements of a vertex input struct 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 patch control-point data is specified in the post-tessellation vertex function as the following templated type:

`patch_control_point<T>`

where `T` is a user defined struct. Each element of `T` must specify an attribute location using `[[attribute(index)]]`.

Member functions are supported by the `patch_control_point<T>` type are:

`constexpr size_t size() const;`

Returns the number of control-points in the patch

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

Returns the data for a specific patch control point identified by `pos`.

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 ]}, …)
{…}
```

4.3.4 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.

4.3.4.1 Vertex Function Input Attributes

Table 9 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 9 Attributes for Vertex Function Input Arguments

Attribute	Corresponding Data Types	Description
`[[vertex_id]]`	`ushort` or `uint`	The per-vertex identifier. The per-vertex identifier includes the base vertex value if one is specified.
`[[instance_id]]`	`ushort` or `uint`	The per-instance identifier. The per-instance identifier includes the base instance value if one is specified.
`[[base_vertex]]`	`ushort` or `uint`	The base vertex value added to each vertex identifier before reading per-vertex data.
`[[base_instance]]`	`ushort` or `uint`	The base instance value added to each instance identifier before reading per-instance data.

Notes on vertex function input attributes:

- 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.

4.3.4.2 Post-Tessellation Vertex Function Input Attributes

Table 10 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 10 Attributes for Post-Tessellation Vertex Function Input Arguments

Attribute	Corresponding Data Types	Description
[[patch_id]]	ushort or uint	The patch identifier.
[[instance_id]]	ushort or uint	The per-instance identifier. The per-instance identifier includes the base instance value if one is specified.
[[base_instance]]	ushort or uint	The base instance value added to each instance identifier before reading per-instance data.
[[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.

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.

4.3.4.3 Vertex Function Output Attributes

Table 11 lists the built-in attributes that can be specified for a return type of a vertex function or the members of a struct that are returned by a vertex function (and the corresponding data types with which they can be used).

Table 11 Attributes for Vertex Function Return Type

Attribute	Corresponding Data Types
[[clip_distance]]	float or float[n] n must be known at compile time
[[point_size]]	float
[[position]]	float4

Attribute	Corresponding Data Types
[[render_target_array_index]]	uchar, ushort or uint The render target array index. This refers to the face of a cubemap, 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.
[[viewport_array_index]]	The viewport (and scissor rectangle) index value of the primitive.

[[viewport_array_index]] enables specifying one viewport or scissor rectangle from multiple active viewports and scissor rectangles. For details about [[viewport_array_index]], see section 4.11.

[[viewport_array_index]] and [[render_target_array_index]] are only available for macOS.

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 struct, it must include an element declared with the [[position]] attribute.

The example below describes a vertex function called process_vertex. The function returns a user-defined struct 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;
}
```

NOTE: Post-tessellation vertex function outputs are the same as a regular vertex function.

4.3.4.4 Fragment Function Input Attributes

Table 12 lists the built-in attributes that can be specified for arguments of a fragment function (and their corresponding data types).

NOTE: 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 struct, it must include an element declared with the `[[position]]` attribute.

Table 12 Attributes for Fragment Function Input Arguments

Attribute	Corresponding Data Types	Description
`[[color(m)]]`	`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 indicating where within a point primitive the current fragment is located. They range from 0.0 to 1.0 across the point.
`[[position]]`	`float4`	Describes the window relative coordinate (x, y, z, 1/w) values for the fragment.
`[[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]]	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]] specifier must be used on the fragment function; otherwise the compilation fails.
[[render_target_ array_index]]	uchar, ushort or uint	The render target array index. This refers to the face of a cubemap, 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.
[[viewport_array_ index]]	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 4.5.)

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 [[color(m)]] attribute is only supported in iOS.

The [[sample_mask]] attribute can only be declared once for a fragment function input.

The [[render_target_array_index]] and [[viewport_array_index]] attributes are only available for macOS. For more details about [[viewport_array_index]], see section 4.11.

Table 13 lists attributes that can be specified for tile arguments that are input to a fragment function in iOS. The data types used to declare [[pixel_position_in_tile]] and [[pixels_per_tile]] must match.

Table 13 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.

[[tile_index]] is a value from [0, n), where n is the number of tiles in the render target.

4.3.4.5 Fragment Function Output Attributes

The return type of a fragment function describes the per-fragment output. A fragment function can output one or more render-target color values, a depth value, and a coverage mask, which must be identified by using the attributes listed in Table 14. 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 14 Attributes for Fragment Function Return Types

Attribute	Corresponding Data Types
[[color(m)]] [[color(m), index(i)]]	floatn, halfn, intn, uintn, shortn or ushortn 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)]]	float
[[sample_mask]]	uint

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 12). 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 4.6 and 4.7.

If index(i) is not specified in the attribute, an index of 0 is assumed. 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
```

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;
}
```

NOTE: If a color attachment index is used both as an input to and output of a fragment function, the data types associated with the input argument and output declared with this color attachment index must match.

4.3.4.6 Kernel Function Input Attributes

When a kernel 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 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, the total number of threads in the threadgroup should be 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 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, or the number of threads in the grid and the threadgroup size are specified. For example, consider a kernel submitted for execution that uses a 2-dimensional 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 (i.e., `threadgroup_position_in_grid`) and `(lx, ly)` be the position of each thread in the threadgroup (i.e., `thread_position_in_threadgroup`).

The thread position in the grid (i.e., `thread_position_in_grid`) is:

```
(gx, gy) = (wx * Sx + lx, wy * Sy + ly)
```

The grid size (i.e., `threads_per_grid`) is:

```
(Gx, Gy) = (Wx * Sx, Wy * Sy)
```

In most cases (other than a tile function), the thread index in the threadgroup (i.e., `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).

Threadgroups may be multi-dimensional, but a SIMD-group is 1-dimensional. Any given thread in a SIMD-group can query its SIMD lane ID and which SIMD-group it is a member of. The number of SIMD-groups (given by `[[simdgroups_per_threadgroup]]`) is computed by the total number of threads in threadgroup / SIMD-group size (i.e., the thread execution width). The `[[dispatch_simdgroups_per_threadgroup]]` is computed as the number of threads in the the threadgroup size specified at dispatch / SIMD-group size.

SIMD-groups execute concurrently within a given threadgroup and make independent forward progress with respect to each other, even 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 the SIMD-group (given by `[[thread_index_in_simdgroup]]`) is a value between 0 and SIMD-group size – 1, inclusive.

Similarly, the number of quad-groups (given by `[[quadgroups_per_threadgroup]]`) is the total number of threads in threadgroup divided by 4, which is the thread execution width of a quad-group. `[[dispatch_quadgroups_per_threadgroup]]` is computed as the number of threads in the threadgroup size specified at dispatch divided by 4. The thread index in a quad-group (given by `[[thread_index_in_quadgroup]]`) is a value between 0 and 3, inclusive.

Table 15 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 15 Attributes for Kernel Function Input Arguments

Attribute	Corresponding Data Types
`[[thread_position_in_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[thread_position_in_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[thread_index_in_threadgroup]]`	`ushort` or `uint`
`[[threadgroup_position_in_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[threads_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[threads_per_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[dispatch_threads_per_threadgroup]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[threadgroups_per_grid]]`	`ushort`, `ushort2`, `ushort3`, `uint`, `uint2` or `uint3`
`[[thread_execution_width]]`	`ushort` or `uint`
`[[threads_per_simdgroup]]`	`ushort` or `uint`
`[[thread_index_in_simdgroup]]`	`ushort` or `uint`
`[[thread_index_in_quadgroup]]`	`ushort` or `uint`
`[[simdgroup_index_in_threadgroup]]`	`ushort` or `uint`

Attribute	Corresponding Data Types
[[quadgroup_index_in_threadgroup]]	ushort or uint
[[simdgroups_per_threadgroup]]	ushort or uint
[[quadgroups_per_threadgroup]]	ushort or uint
[[dispatch_simdgroups_per_threadgroup]]	ushort or uint
[[dispatch_quadgroups_per_threadgroup]]	ushort or uint

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]] atriabute 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]] and [[threads_per_threadgroup]], and [[dispatch_threads_per_threadgroup]] must match.
- If [[thread_position_in_threadgroup]] is declared to be of type uint, uint2 or uint3, then [[thread_index_in_threadgroup]] must be declared to be of 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.

4.3.5 stage_in Attribute

The per-fragment inputs to a fragment function are generated using the output from a vertex function and the fragments generated by the rasterizer. The per-fragment inputs are identified using the `[[stage_in]]` attribute.

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, per-vertex inputs can also be passed as arguments to a vertex function by declaring them with the `[[stage_in]]` attribute.

A kernel function reads per-thread inputs by indexing into a 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, per-thread inputs can also be passed as arguments to a kernel function by declaring them with the `[[stage_in]]` attribute.

Only one argument of the vertex, fragment or kernel function can be declared with the `[[stage_in]]` attribute. For a user-defined struct declared with the `[[stage_in]]` attribute, the members of the struct can be:

- a scalar integer or floating-point value or
- a vector of integer or floating-point values.

NOTE: Packed vectors, matrices, structs, references or pointers to a type, and arrays of scalars, vectors, matrices and bitfields are not supported as members of the struct declared with the `stage_in` attribute.

4.3.5.1 Vertex Function Example that Uses the stage_in Attribute

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;
}
```

4.3.5.2 Fragment Function Example that Uses the stage_in Attribute

An example in section 4.3.3 previously introduces the `process_vertex` vertex function, which returns a `VertexOutput` struct 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 5.10.3).

```
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;
}
```

4.3.5.3 Kernel Function Example that Uses the stage_in Attribute

The following example shows how to pass per-thread inputs using the `stage_in` attribute. Using 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]])
{…}
```

4.4 Storage Class Specifiers

Metal supports the `static` and `extern` storage class specifiers. Metal does not support the `thread_local` storage class specifiers.

The `extern` storage-class specifier can only be used for functions and variables declared in program scope or variables declared inside a function. The `static` storage-class specifier is only for device variables declared in program scope (see section 4.2.3) and is not for variables declared inside a graphics or kernel function. In the following example, the `static` specifier is incorrectly used for the variables `b` and `c` declared inside a kernel function.

```
extern constant float4 noise_table[256];
static constant float4 color_table[256] = { … };  // 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;     // static is an error.
     static float c;               // static is an error.

     …
     my_foo(img);
     …
     my_bar(ptr);
     …
}
```

4.5 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`.

The sampling attribute variants (`sample_perspective` and `sample_no_perspective`) interpolate 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.

The following example is user-defined struct that specifies how data in certain members are interpolated:

```
struct FragmentInput {
    float4 pos [[center_no_perspective]];
    float4 color [[center_perspective]];
    float2 texcoord;
    int index [[flat]];
    float f [[sample_perspective]];
};
```

4.6 Per-Fragment Function vs. Per-Sample Function

The fragment function is typically executed per-fragment. The sampling attribute identifies if any fragment input is to be interpolated at per-sample vs. per-fragment. Similarly, the `[[sample_id]]` attribute is used to identify the current sample index and the `[[color(m)]]` attribute is used to identify the destination fragment color or sample color (for a multisampled color attachment) value. If any of these attributes are used 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 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 is executed 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);
}
```

4.7 Imageblock Attributes

This section describes several attributes that are used with imageblocks. The `[[imageblock_data(type)]]` attribute that specifies input and output imageblock with an explicit imageblock layout on a fragment function is described in section 2.10.

4.7.1 user Attribute for Matching Data Members of Master and View Imageblocks

The `[[user(name)]]` attribute can be used to specify an attribute name for a data member of the imageblock data type for a fragment function. If the imageblock struct specified in a fragment function is a subset of the master explicit imageblock struct, the following rules are used to match data members declared in the imageblock struct used with a fragment function with corresponding data members declared in the master explicit imageblock struct:

- Every attribute name given by [[user(name)]] must be unique for each data member in the imageblock.
- If the attribute name given by `[[user(name)]]` is specified for a data member, the attribute name given by name must match with a data member declared in the master explicit imageblock struct. In addition, the 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 struct must match. Additionally, the data member cannot be within a nested struct that is either within the view imageblock struct or within the master imageblock struct.

The example below shows master and view imageblock structs with data members that are declared with the `[[user(name)]]` attribute:

Example:

```
// The explicit layout imageblock data master struct.
struct IM {
     rgba8unorm<half4> a [[user(my_a), raster_order_group(0)]];
     rgb9e5f<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 struct for input.
struct IVIn {
      rgb9e5f<float4> x [[user(my_b)]]; // Maps to IM::b
      float y [[user(my_d)]]; // Maps to IM::d
};

// The explicit layout imageblock data view struct for output.
struct IVOut {
      int z [[ user(my_c) ]]; // Maps to IM::c
};

// The fragment return struct.
struct FragOut {
      // IVOut is a view of the master IM.
      IVOut i [[ imagelock_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 example below shows master and view imageblock structs with data members that are declared without the `[[user(name)]]` attribute:

```
struct IM {
      rgba8unorm<half4> a [[raster_order_group(0)]];
      rgb9e5f<float4> b [[raster_order_group(0)]];
      int c [[raster_order_group(0)]];
```

```
    float d [[raster_order_group(0)]];
};

struct IVIn {
    rgb9e5f<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;
}
```

Nested structs can be declared in the master imageblock and view imageblock structs. Below is an example that shows how nested structs in an imageblock can be used with data members declared with the `[[user(name)]]` attribute:

```
struct A {
    rgba8unorm<half4> a [[user(A_a)]];
    rgb9e5f<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
    rgb9e5f<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 struct must correspond to exactly one master struct field. A master struct field can refer to a top-level struct field as well as a field within a nested struct. It is illegal for two or more view struct fields to alias the same master struct 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)]])
{…}
```

NOTE: Explicit imageblock types cannot have data members declared with the [[color(n)]] attribute.

4.7.2 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, the `[[raster_order_group(index)]]` attribute must be specified with data members of the master explicit imageblock data struct.

If the master explicit imageblock struct contains data members that are structs, the `[[raster_order_group(index)]]` attribute can be specified for all data members in the nested struct or just the nested struct. If the `[[raster_order_group(index)]]` attribute is specified on the nested struct, then it applies to all data members of the nested struct, and no data member in the nested struct can have the `[[raster_order_group(index)]]` attribute declared.

The `[[raster_order_group(index)]]` may be optionally specified with data members of an imageblock view struct, but the `[[raster_order_group(index)]]` must match the same `[[raster_order_group(index)]]` specified on the data member of the master explicit imageblock struct.

The following example shows how the `[[raster_order_group(index)]]` attribute can be specified for data members of a master imageblock. Since the `[[raster_order_group(index)]]` attribute is used on the `S` struct member of the `gBufferData` struct, the attribute cannot be used on any members of the `S` struct.

```
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 the `[[raster_order_group(index)]]` attribute can be specified for data members of a master imageblock that are declared as an array of a struct 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 declared as an array of a struct of type `S` with members that specify raster order groups that will 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]; // compilation error
```

```
    rgb11b10f<half3> lighting [[raster_order_group(2)]];
};
```

4.7.3 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, the following attribute can also be specified with 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)`.

NOTE: Accessing data members of an implicit imageblock that are aliased 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;
    rgb9e5f<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;
```

```
}
```

4.7.4 Imageblocks and Function Constants

Use of `[[function_constant(name)]]` is unsupported with data members of an imageblock struct that is either passed as inputs to or returned as output from a fragment or kernel function.

4.8 Programmable Blending

The fragment function can be used to perform per-fragment or per-sample programmable blending. The color attachment index identified by the `[[color(m)]]` attribute can be specified as an argument to a fragment function.

Below is an OpenGL ES programmable blending example that describes how to paint grayscale onto what is below.

The GLSL version is:

```
#extension GL_APPLE_shader_framebuffer_fetch : require
void main()
{
    // RGB to grayscale
    mediump float lum = dot(gl_LastFragData[0].rgb, vec3(0.30,0.59,0.11));
    gl_FragColor = vec4(lum, lum, lum, 1.0);
}
```

The Metal version equivalent is:

```
fragment half4
paint_grayscale(half4 dst_color [[color(0)]])
{
    // RGB to grayscale
    half lum = dot(dst_color.rgb, half3(0.30h, 0.59h, 0.11h));
    return half4(lum, lum, lum, 1.0h);
}
```

4.9 Graphics Function – Signature Matching

A graphics function signature is a list of parameters that are either input to or output from a graphics function.

4.9.1 Vertex – Fragment Signature Matching

There are two kinds of data that can be passed between a vertex and fragment function: user-defined and built-in variables.

The per-instance input to a fragment function is declared with the `[[stage_in]]` attribute. These are output by an associated vertex function.

Built-in variables are declared with one of the attribute attributes defined in section 4.3.3. These are either generated by a vertex function (such as `[[position]]`, `[[point_size]]`, `[[clip_distance]]`), are generated by the rasterizer (such as `[[point_coord]]`, `[[front_facing]]`, `[[sample_id]]`, `[[sample_mask]]`) or refer to a framebuffer color value (such as `[[color]]`) passed as an input to the fragment function.

The built-in variable `[[position]]` must always be returned. The other built-in variables (`[[point_size]]`, `[[clip_distance]]`) generated by a vertex function, if needed, must be declared in the return type of the vertex function but cannot be accessed by the fragment function.

Built-in variables generated by the rasterizer or refer to a framebuffer color value may also declared as arguments of the fragment function with the appropriate attribute attribute.

The attribute `[[user(name)]]` syntax can also be used to specify an attribute name for any user-defined variables.

A vertex and fragment function are considered to 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 for a matching type to be associated with elements of the vertex return type. For elements that are user-generated outputs, the following rules apply:
- If the attribute name given by `[[user(name)]]` is specified 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 the `[[user(name)]]` attribute name is not specified, then the argument name and types must match.

Below 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(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;
}
```

`my_vertex` and `my_fragment` or `my_vertex` and `my_fragment2` can be used together to render a primitive.

Below is another 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;
}
```

Below is another example of compatible signatures:

```
struct VertexOutput {
    float4 position [[position]];
    float3 normal;
    float2 texcoord;
};

struct FragInput {
    float4 position [[position]];
    float2 texcoord;
};

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 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:

```
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:

```
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;
}
```

4.10 Program Scope Function Constants

Metal shading language version 1.2 introduced support for function constants, which generates multiple variants of a function. Before version 1.2, developers could compile one function many times with different pre-processor macro defines to enable different features (a.k.a., an *ubershader*). Using pre-processor 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.

The Metal shading language allows you to use function constants with the same ease of use as using pre-processor macros but moves the generation of the specific variants to the creation of the pipeline state. You do not have to compile the variants offline.

4.10.1 Specifying Program Scope Function Constants

The Metal shading language is extended to allow program scope variables to be declared with the following attribute:

```
[[function_constant(index)]]
```

Program scope variables declared with the `[[function_constant(index)]]` attribute or program scope variables initialized with variables declared with this attribute are referred to as function constants.

These function constants are not initialized in the Metal shader source but instead their values are specified when the render or compute pipeline state is created. `index` is used to specify a location index that can be used to 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)]];
```

Variables in program scope declared in the constant address space can also be initialized using a function constant(s).

Examples:

```
constant int a [[function_constant(0)]];

constant bool b [[function_constant(2)]];

constant bool c = ((a == 1) && b);

constant int d = (a * 4);
```

The value of function constants `a` and `b` are specified when the render or compute pipeline state is created.

The following built-in function can be used to determine if a function constant is available; i.e., it has been defined when the render or compute pipeline state is created. `name` must refer to a function constant variable.

```
bool is_function_constant_defined(name)
```

`is_function_constant_defined(name)` returns `true` if the function constant variable is defined when the render or compute pipeline state is created and `false` otherwise.

If a function constant variable value is not defined when the render or compute pipeline state is created 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 returned by `is_function_constant_defined(name)` is `false`.

Function constants can be used in Metal:

- To control code paths that get compiled,

- To specify the optional arguments of a function (graphics, kernel or user functions), or
- To specify optional elements of a struct that is declared with the `[[stage_in]]` attribute.

4.10.1.1 Function Constants to Control Code Paths to Compile

Consider the following function which uses pre-processor 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;
}
```

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.

4.10.1.2 Function Constants when Declaring the Arguments of Functions

Arguments to a graphics, kernel or user functions can be declared with the `[[function_constant(name)]]` attribute attribute to identify that the argument is optional. `name` refers to a function constant variable. If the value of the function constant variable given by `name` is `non-zero` or `true` (determined when the render or compute pipeline state is created), the argument is considered to be declared in the function signature. If the value of the function constant variable given by `name` is `0` or `false`, the argument is **not** considered to be declared in the function signature. If `name` refers to a function constant variable that has not been defined (determined when the render or compute pipeline state is created), the behavior is the same as if `is_function_constant_defined(name)` is used and its value is `false`.

Consider the following fragment function that uses pre-processor 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]])
```

The corresponding fragment function now written using function constants is:

```
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(inputBufferDefined)]],
                     device int *output [[buffer(1)]],
                     uint tid [[thread_position_in_grid]])
{
     if (hasInputBuffer)
          output[tid] = inputA[0] * tid;
     else
          output[tid] = tid;
}
```

4.10.1.3 Function Constants for Elements of a [[stage_in]] Struct

Elements of a struct declared with the `[[stage_in]]` attribute passed to a graphics function (or post-tessellation vertex function) can also be declared with the `[[function_constant(name)]]` attribute to identify that the element is optional. `name` refers to a function constant variable. If the value of the function constant variable given by `name` is `non-zero` or `true` (determined when the render or compute pipeline state is created), the element in the struct is considered to be 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 struct. If `name` refers to a function constant variable that has not been defined (determined when the render or compute pipeline state is created), the behavior is the same as if `is_function_constant_defined(name)` is used and its value is `false`.

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;
}
```

4.10.1.4 Function Constants for Resource Bindings

Starting in Metal 2.0, arguments to a graphics or kernel functions that are a resource (buffer, texture, or sampler) can have their binding number be specified using a function constant. 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)]], …)
{…}
```

4.10.1.5 Function Constants for Color Attachments and Raster Order Groups

Starting in Metal 2.0, the `[[color(n)]]` and `[[raster_order_group(index)]]` indices can also be a function constant. The function constant specified as indices for color and raster order group attributes 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)]];
constant int group1 [[function_constant(3)]];

struct FragmentOutput {
    float4 color0 [[color(colorAttachment0)]];
    float4 color1 [[color(colorAttachment1)]];
```

```
};

fragment FragmentOutput
my_fragment(texture2d<float> texA [[texture(0),
raster_order_group(group0)]], …)
{…}
```

4.10.1.6 Function Constants with Elements of a Struct

Starting in Metal 2.0, to identify that an element of a struct is optional, the `[[function_constant(name)]]` attribute can be declared with elements of a struct that are declared as a return type of a graphics or user function, or 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 4.10.1.3.

`name` refers to a function constant variable. If the value of the function constant variable given by `name` is non-zero or `true` (determined when the render or compute pipeline state is created), the element in the struct is considered to be 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 struct. If `name` refers to a function constant variable that has not been defined, the behavior is the same as if `is_function_constant_defined(name)` is used and its value is `false`.

4.11 Per-Primitive Viewport and Scissor Rectangle Index Selection

The `[[viewport_array_index]]` attribute supports built-in variables as both vertex output and fragment input. With this attribute, vertex function output is used to specify which viewport or scissor rectangle is used for rasterization from an array specified by the `setViewports:count:` or `setScissorRects:count:` framework calls. `[[viewport_array_index]]` is only available for macOS.

The input value of `[[viewport_array_index]]` in the fragment function is the same as the value written to `[[viewport_array_index]]` in the vertex function, even if the value is out of range. If the vertex output does not specify `[[viewport_array_index]]`, the default index value is 0. Specifying a value for `[[viewport_array_index]]` that is larger than the number of viewports passed in by `setViewports:count:` (or in the case of `setViewport:`, larger than 0) is treated as if the `[[viewport_array_index]]` value is 0. Hardware that does not support this feature acts as if the maximum permitted viewport and scissor rectangle count is 1, and the maximum allowed `[[viewport_array_index]]` is 0. This effectively means that every primitive is rendered to viewport/scissor rectangle 0, regardless of the passed value.

Specifying `[[viewport_array_index]]` in a post-tessellation vertex function is allowed. `[[viewport_array_index]]` cannot be specified in the tessellation factor buffer.

Specifying [[viewport_array_index]] as fragment function input counts against the number of varyings available and reduces the number of components that can be passed from vertex function to fragment function.

In case the value of [[viewport_array_index]] differs for different vertices of the primitive, the value specified at the provoking vertex is used in the vertex function and is also passed to the fragment function. The same behavior applies to primitives generated by tessellation.

4.12 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 a buffer(s), 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 struct type, a reference or a pointer to a type.
- The number of inputs to a fragment function declared with the stage_in attribute can be a maximum of 128 scalars. (This number does not include the built-in variables declared with one of the following attributes: [[color(m)]], [[front_facing]], [[sample_id]] and [[sample_mask]].) If an input to a fragment function is a vector then this counts as n 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.

5 Metal Standard Library

This chapter describes the functions supported by the Metal standard library.

5.1 Namespace and Header Files

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

5.2 Common Functions

The functions in Table 16 are in the Metal standard library and are defined in the header <metal_common>. `T` is one of the scalar or vector floating-point types.

Table 16 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 return 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 variants is how NaNs are handled. 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 6.2) is used to select the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these common functions.

5.3 Integer Functions

The integer functions in Table 17 are in the Metal standard library and 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.

Table 17 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 component type of `x`, if `x` is a vector.
`T extract_bits(T x, uint offset, uint bits)`	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. The result is undefined if the sum of `offset` and `bits` is greater than the number of bits used to store the operand.
`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)`	Returns the insertion of the `bits` least-significant bits of `insert` into `base`. The result will have bits [`offset`, `offset+bits-1`] taken from bits [`0`, `bits-1`] of `insert`, and all other bits taken directly from the corresponding bits of `base`. If `bits` is zero, the result is `base`. The result is undefined if the sum of `offset` and `bits` is greater than the number of bits used to store the operand.
`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 min(T x, T x)`	Returns `y` if `y < x`, otherwise it returns `x`.
`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`.

Built-in integer functions	Description
`T reverse_bits(T x)`	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.

5.4 Relational Functions

The relational functions in Table 18 are in the Metal standard library and 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 18 Relational Functions in the Metal Standard Library

Built-in relational functions	Description
`bool all(`T_b` x)`	Returns true only if all components of x are true.
`bool any(`T_b` x)`	Returns true only if any component of x are true.
T_b `isfinite(T x)`	Test for finite value.
T_b `isinf(T x)`	Test for infinity value (positive or negative).
T_b `isnan(T x)`	Test for a NaN.
T_b `isnormal(T x)`	Test for a normal value.
T_b `isordered(T x, T y)`	Test if arguments are ordered. `isordered()` takes arguments x and y and returns the result `(x == x) && (y == y)`.
T_b `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 and false otherwise.

Built-in relational functions	Description
T_b not(T_b x)	Returns the component-wise logical complement of x.
T select(T a, T b, T_b c) T_i select(T_i a, T_i b, T_b c)	For each component of a vector type, result[i] = c[i] ? b[i] : a[i] For a scalar type, result = c ? b : a
T_b signbit(T x)	Test for sign bit. Returns true if the sign bit is set for the floating-point value in x and false otherwise.

5.5 Math Functions

The math functions in Table 19 are in the Metal standard library and are defined in the header <metal_math>. T is one of the scalar or vector floating-point types. T_i refers only to the scalar or vector integer types.

Table 19 Math Functions in the Metal Standard Library

Built-in math functions	Description
T acos(T x)	Arc cosine function.
T acosh(T x)	Inverse hyperbolic cosine.
T asin(T x)	Arc sine function.
T asinh(T x)	Inverse hyperbolic sine.
T atan(T y_over_x)	Arc tangent function.
T atan2(T y, T x)	Arc tangent of y over x.
T atanh(T x)	Hyperbolic arc tangent.
T ceil(T x)	Round 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.
T cosh(T x)	Compute hyperbolic cosine.
T cospi(T x)	Compute cos(πx).

Built-in math functions	Description
`T exp(T x)`	Compute the base- e exponential of x.
`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 is less than or equal to y.
`T floor(T x)`	Round 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 it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.
`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 fmod(T x, T y)`	Returns x – y * trunc(x/y).
`T fract(T x)`	Returns the fractional part of x which is greater than or equal to 0 or less than 1.
T `frexp(T x,` T_i `&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.
T_i `ilogb(T x)`	Return the exponent as an integer value.
`T ldexp(T x,` T_i `k)`	Multiply x by 2 to the power k.
`T log(T x)`	Compute natural logarithm.
`T log2(T x)`	Compute a base 2 logarithm.
`T log10(T x)`	Compute a base 10 logarithm.

Built-in math functions	Description
`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 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.
`T rsqrt(T x)`	Compute inverse square root.
`T sin(T x)`	Compute sine.
`T sincos(T x, T &cosval)`	Compute sine and cosine of x. The computed sine is the return value and compute cosine is returned in cosval.
`T sinh(T x)`	Compute hyperbolic sine.
`T sinpi(T x)`	Compute `sin(πx)`.
`T sqrt(T x)`	Compute square root.
`T tan(T x)`	Compute tangent.
`T tanh(T x)`	Compute hyperbolic tangent.
`T tanpi(T x)`	Compute `tan(πx)`.
`T trunc(T x)`	Round to integral value using the round to zero rounding mode.

For single precision floating-point, Metal supports two variants of the math functions listed in Table 19: the precise and the fast variants. The `ffast-math` compiler option (refer to section 6.2) is used to select the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide developers a way to explicitly 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 –ffast-math is specified as
                  // compile option.
float b = fast::sin(x);// use fast version of sin()
float c = precise::cos(x); // use precise version of cos()
```

Table 20 lists available symbolic constants with values of type `float` that are accurate within the precision of a single-precision floating-point number.

Table 20 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 21 lists available symbolic constants with values of type `half` that are accurate within the precision of a half-precision floating-point number.

Table 21 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.
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

5.6 Matrix Functions

The functions in Table 22 are in the Metal standard library and are defined in the header <metal_matrix>. `T` is `float` or `half`.

Table 22 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);
```

5.7 Geometric Functions

The functions in Table 23 are in the Metal standard library and 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 (i.e., `float` if `T` is `floatn` and `half` if `T` is `halfn`).

Table 23 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`, i.e., `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`, i.e., `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`, i.e., `sqrt(x[0]`^{2} `+ x[1]`^{2} `+ …)`
T_s `length_squared(T x)`	Return the square of the length of vector x, i.e., `(x[0]`^{2} `+ x[1]`^{2} `+ …)`
`T normalize(T x)`	Returns 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`, returns the reflection direction: `I – 2 * dot(N, I) * N` In order to achieve the desired result, `N` must be normalized.
`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`. The `ffast-math` compiler option (refer to section 6.2) is used to select the appropriate variant when compiling the Metal source. In addition, the `metal::precise` and `metal::fast` nested namespaces are also available and provide developers a way to explicitly select the fast or precise variant of these geometric functions.

5.8 Compute Functions

The functions in section 5.8 and its subsections can only be called from a `kernel` function and are defined in the header `<metal_compute>`.

5.8.1 Threadgroup and SIMD-group Synchronization Functions

Table 24 lists supported threadgroup and SIMD-group synchronization functions.

Table 24 Synchronization Compute Function in the Metal Standard Library

Built-in threadgroup function	Description
`void threadgroup_barrier(mem_flags flags)` `void threadgroup_barrier(mem_flags flags, memory_scope scope)`	All threads in a threadgroup executing the kernel must execute this function before any thread is allowed to continue execution beyond the `threadgroup_barrier`.
`void simdgroup_barrier(mem_flags flags)` `void simdgroup_barrier(mem_flags flags, memory_scope scope)`	All threads in a SIMD-group executing the kernel must execute this function before any thread is allowed to continue execution beyond the `simdgroup_barrier`.

Refer to section 5.12.2 for valid values of `scope`.

The `threadgroup_barrier` function acts as an execution and memory barrier. The `threadgroup_barrier` function must be encountered by all threads in a threadgroup executing the kernel. The `threadgroup_barrier` function also supports a variant that specifies the memory scope. For the `threadgroup_barrier` variant that does not take a memory scope, the default `scope` is `memory_scope_threadgroup`.

If `threadgroup_barrier` is inside a conditional statement and if any thread enters the conditional statement and executes the barrier, then all threads in the threadgroup must enter the conditional and execute the barrier.

If `threadgroup_barrier` is inside a loop, for each iteration of the loop, all threads in the threadgroup must execute the `threadgroup_barrier` before any threads are allowed to continue execution beyond the `threadgroup_barrier`.

The `threadgroup_barrier` function can also queue a memory fence (reads and writes) to ensure correct ordering of memory operations to threadgroup or device memory.

The `simdgroup_barrier` function acts as an execution and memory barrier. The `simdgroup_barrier` function must be encountered by all threads in a SIMD-group executing the kernel. The `simdgroup_barrier` function also supports a variant that specifies the memory scope. For the `simdgroup_barrier` variant that does not take a memory scope, the default `scope` is `memory_scope_simdgroup`.

If `simdgroup_barrier` is inside a conditional statement and if any thread enters the conditional statement and executes the barrier, then all threads in the SIMD-group must enter the conditional and execute the barrier.

If `simdgroup_barrier` is inside a loop, for each iteration of the loop, all threads in the SIMD-group must execute the `simdgroup_barrier` before any threads are allowed to continue execution beyond the `simdgroup_barrier`.

The `simdgroup_barrier` function can also queue a memory fence (reads and writes) to ensure correct ordering of memory operations to threadgroup or device memory

The `mem_flags` argument to `threadgroup_barrier` and `simdgroup_barrier` is a bitfield and can be one or more of the following values, as described in Table 25.

Table 25 mem_flags Enum Values for Barrier Functions

mem_flags	Description
`mem_none`	In this case, 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`	Ensure correct ordering of memory operations to texture memory for threads in a threadgroup.

The enumeration types used by `mem_flags` are specified as follows:

```
enum class mem_flags {mem_none, mem_device, mem_threadgroup, mem_texture};
```

The `scope` argument specifies whether the memory accesses of threads in the threadgroup to memory address space(s) identified by `flags` become visible to all threads in the threadgroup or the device.

5.9 Graphics Functions

This section and its subsections list the set of graphics functions that can be called by a fragment and vertex functions. These are defined in the header `<metal_graphics>`.

5.9.1 Fragment Functions

The functions in this section (listed in Table 26, Table 27, and Table 28) can only be called inside a fragment function (a function declared with the `fragment` function specifier) 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.

5.9.1.1 Fragment Functions – Derivatives

Metal includes the functions in Table 26 to compute derivatives. `T` is one of `float`, `float2`, `float3`, `float4`, `half`, `half2`, `half3` or `half4`.

NOTE: Derivatives are undefined within non-uniform control flow.

Table 26 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`; i.e., `fabs(dfdx(p)) + fabs(dfdy(p))`

5.9.1.2 Fragment Functions – Samples

Metal includes the following per-sample functions in Table 27. `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 27 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 indx)`	Returns the normalized sample offset (x, y) for a given sample index `indx`. Values of x and y are in [0.0 … 1.0).

5.9.1.3 Fragment Functions – Flow Control

The Metal function in Table 28 is used to terminate a fragment.

Table 28 Fragment Flow Control Function in the Metal Standard Library

Built-in fragment functions	Description
`void discard_fragment(void)`	Marks the current fragment as being terminated, and the output of the fragment function for this fragment is discarded.

NOTE:

- **Writes to a buffer or texture from a fragment thread before `discard_fragment` is called 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 following derivative computations (explicit or implicit) is undefined.**

5.10 Texture Functions

The texture member functions are categorized into: sample from a texture, sample compare from a texture, read (sampler-less read) from a texture, gather from a texture, gather compare from a texture, write to a texture, texture query and texture fence functions.

These are defined in the header `<metal_texture>`.

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 that is 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.

Overloaded variants of texture `sample` and `sample_compare` functions for a 2D texture, 2D texture array, 3D texture, and cube are available and allow the texture to be sampled using a bias that is applied to a mip-level before sampling or with user-provided gradients in the x and y direction.

NOTE:

- **The texture `sample`, `sample_compare`, `gather` and `gather_compare` functions require that the texture is declared with the `sample` access attribute.**
- **The texture `sample_compare` and `gather_compare` functions are only available for depth texture types.**
- **The texture `read` functions require that the texture is declared with the `sample`, `read` or `read_write` access attribute.**
- **The texture `write` functions require that the texture is declared with the `write` or `read_write` access attribute.**
- **The `sample` and `sample_compare` functions that do not take an explicit LOD or gradients and the `gather` and `gather_compare` functions when called from kernel or vertex functions assume an implicit LOD of 0.**
- **For the `gather` and `gather_compare` functions, the four samples that would contribute to filtering are placed into xyzw in counter clockwise order starting with the sample to the lower left of the queried location. This is the same as nearest sampling with un-normalized texture coordinate deltas at the following locations: (-,+), (+,+),(+,-),(-,-), where the magnitude of the deltas are always half a texel.**

5.10.1 1D Texture

The following member functions can be used to sample from a 1D texture.

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

T_v is a 4-component vector type based on the templated type <T> used to declare the texture type. If T is float, T_v is float4. If T is half, T_v is half4. If T is int, T_v is int4. If T is uint, T_v is uint4. If T is short, T_v is short4 and if T is ushort, T_v is ushort4.

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

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

The following member functions can be used to 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)
```

The following 1D texture query member functions are provided. 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
```

5.10.22 1D Texture Array

The following member functions can be used to sample from a 1D texture array.

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

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

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

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

The following member functions can be used to 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)
```

The following 1D texture array query member functions are provided. 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
```

5.10.3 2D Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
```

```
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`.

The following member functions can be used to perform sampler-less reads from a 2D texture:

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

The following member functions can be used to write to a 2D texture.

```
void write(Tv color, uint2 coord, uint lod = 0)
void write(Tv color, ushort2 coord, ushort lod = 0)
```

The following member functions can be used to do a gather of four samples that would be used 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 2D texture query member functions are provided.

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

5.10.3.1 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 mip-level
clr = tex.sample(s, coord, level(lod));

// sample with an offset
clr = tex.sample(s, coord, offset);

// sample using a mip-level and an offset
clr = tex.sample(s, coord, level(lod), offset);
```

5.10.4 2D Texture Array

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`.

The following member functions can be used to 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
```

The following member functions can be used to write to a 2D texture array.

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

void write(Tv color, ushort2 coord, ushort array, ushort lod = 0)
```

The following member functions can be used to do a gather of four samples that would be used 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 2D texture array query member functions are provided.

```
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
```

5.10.5 3D Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
level(float lod)
gradient3d(float3 dPdx, float3 dPdy)
```

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient3d`.

The following member functions can be used to perform sampler-less reads from a 3D texture:

```
Tv read(uint3 coord, uint lod = 0) const
Tv read(ushort3 coord, ushort lod = 0) const
```

The following member functions can be used to write to a 3D texture.

```
void write(Tv color, uint3 coord, uint lod = 0)
void write(Tv color, ushort3 coord, ushort lod = 0)
```

The following 3D texture query member functions are provided.

```
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
```

5.10.6 Cube Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following member functions can be used to sample from a cube texture.

```
Tv sample(sampler s, float3 coord) const
Tv sample(sampler s, float3 coord, lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`.

Table 29 describes the cube face and the number used to identify the face.

Table 29 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 functions can be used to do a gather of four samples that would be used for bilinear interpolation when sampling a cube texture.

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

The following member functions can be used to 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
```

The following member functions can be used to write to a cube texture.

```
void write(Tv color, uint2 coord, uint face, uint lod = 0)
void write(Tv color, ushort2 coord, ushort face, ushort lod = 0)
```

The following cube texture query member functions are provided.

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

5.10.7 Cube Array Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
level(float lod)
gradientcube(float3 dPdx, float3 dPdy)
```

The following member functions can be used to sample from a cube array texture.

```
Tv sample(sampler s, float3 coord, uint array) const
Tv sample(sampler s, float3 coord, uint array, lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`.

The following member functions can be used to do a gather of four samples that would be used for bilinear interpolation when sampling a cube array texture.

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

The following member functions can be used to perform sampler-less reads from a cube array texture:

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

The following member functions can be used to write to a cube array texture.

```
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)
```

The following cube array texture query member functions are provided.

```
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
```

5.10.8 2D Multisampled Texture

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

```
Tv read(uint2 coord, uint sample) const
T read(ushort2 coord, ushort sample) const
```

The following 2D multisampled texture query member functions are provided.

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

5.10.9 2D Depth Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
level(float lod)
gradient2d(float2 dPdx, float2 dPdy)
```

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`.

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`. `T` must be a `float` type.

NOTE: `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.

The following member functions can be used to 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
```

The following built-in functions can be used to do a gather of four samples that would be used 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 be used do a gather of four samples that would be used 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 2D depth texture query member functions are provided.

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

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

```
uint  get_num_mip_levels() const
```

5.10.10 2D Depth Texture Array

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`.

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradient2d`. `T` must be a `float` type.

The following member functions can be used to 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
```

The following member functions can be used to do a gather of four samples that would be used 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 functions can be used do a gather of four samples that would be used 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 2D depth texture array query member functions are provided.

```
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
```

5.10.11 Cube Depth Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
```

```
level(float lod)
```

```
gradientcube(float3 dPdx, float3 dPdy)
```

The following member functions can be used to sample from a cube depth texture.

```
T sample(sampler s, float3 coord) const
```

`T sample(sampler s, float3 coord, lod_options options) const`

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`.

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`. `T` must be a `float` type.

The following member functions can be used to 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
```

The following member functions can be used to do a gather of four samples that would be used for bilinear interpolation when sampling a cube depth texture.

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

The following member functions can be used do a gather of four samples that would be used for bilinear interpolation when sampling a cube texture and comparing these samples with a specified comparison value.

T_v `gather_compare(sampler s, float3 coord, float compare_value) const`

`T` must be a `float` type.

The following cube depth texture query member functions are provided.

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

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

```
uint get_num_mip_levels() const
```

5.10.12 Cube Array Depth Texture

The following data types and corresponding constructor functions are available to specify various sampling options:

```
bias(float value)
```

```
level(float lod)
```

```
gradientcube(float3 dPdx, float3 dPdy)
```

The following member functions can be used to sample from a cube array depth texture.

```
T sample(sampler s, float3 coord, uint array) const
T sample(sampler s, float3 coord, uint array, lod_options options) const
```

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`.

The following member functions can be used to 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
```

`lod_options` must be one of the following types: `bias`, `level` or `gradientcube`. `T` must be a `float` type.

The following member functions can be used to 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
```

The following member functions can be used to do a gather of four samples that would be used for bilinear interpolation when sampling a cube depth texture.

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

The following member functions can be used do a gather of four samples that would be used for bilinear interpolation when sampling a cube 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 cube depth texture query member functions are provided.

```
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
```

5.10.13 2D Multisampled Depth Texture

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

```
T read(uint2 coord, uint sample) const
T read(ushort2 coord, ushort sample) const
```

The following 2D multisampled depth texture query member functions are provided.

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

5.10.14 Texture Fence Functions

The following member function is supported by texture types that can be declared with the `access::read_write` attribute:

```
void  fence()
```

The texture `fence` function ensures that writes to the texture by a thread become visible to subsequent reads from that texture by the same thread (i.e. the thread performing the write).

The following example show how the texture `fence` function can be used 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);
     …
     // fence to ensure that writes by thread become
     // visible to later reads by thread
     texA.fence();

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

```
}
```

5.10.15 Null Texture Functions

The following function can be used to determine if a texture is a null texture.

```
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>);
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>);
```

Returns `true` if the texture is a null texture and `false` otherwise. The behavior of calling any texture member function with a null texture is undefined.

5.11 Pack and Unpack Functions

This section lists the Metal functions for converting a vector floating-point data to and from a packed integer value. The functions are defined in the header `<metal_pack>`. Refer to section 7.7 for details on how to convert from a 8-bit, 10-bit or 16-bit signed or unsigned integer value to a normalized single- or half-precision floating-point value and vice-versa.

5.11.1 Unpack Integer(s); Convert to a Floating-Point Vector

Table 30 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 30 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 1010102 (10a2), or 565 color value to the corresponding normalized single- or half-precision floating-point vector.

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

Table 31 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 31 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 4-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 4-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 2-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 4- or 3-component vector of normalized single- or half-precision floating-point values to a packed, 1010102 or 565 color integer value.

5.12 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 an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. 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>`.

5.12.1 Memory Order

The enumerated type `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations as defined in section 29.3 of the C++14 specification and may provide for operation ordering. For details on different memory orders, see sections 5.12.1.1, 5.12.1.2, and 5.12.1.3.

For `ios-metal2.0`, all the enumerated values listed in Table 32 are supported with atomic operations.

```
enum memory_order {memory_order_relaxed, memory_order_acquire,
memory_order_release, memory_order_acq_rel, memory_order_seq_cst};
```

For pre-2.0 versions of Metal on iOS and all versions of Metal on macOS, `memory_order_relaxed` is the only `memory_order` supported with atomic operations.

```
enum memory_order {memory_order_relaxed };
```

Table 32 Memory Ordering Enum Values

Memory Order	Description
`memory_order_relaxed`	There are no synchronization or ordering constraints, only atomicity is required of this operation.
`memory_order_acquire`	A load operation with this memory order performs the acquire operation on the affected memory location: prior writes made to other memory locations by the thread that did the release become visible in this thread.
`memory_order_release`	A store operation with this memory order performs the release operation: prior writes to other memory locations become visible to the threads that do an acquire on the same location.
`memory_order_acq_rel`	A load operation with this memory order performs the acquire operation on the affected memory location and a store operation with this memory order performs the release operation.
`memory_order_seq_cst`	Same as `memory_order_acq_rel`, plus a single total order exists in which all threads observe all modifications in the same order.

5.12.1.1 Relaxed Ordering

Atomic operations tagged `memory_order_relaxed` are not synchronization operations. These operations do not order memory, but they guarantee atomicity and modification order consistency.

Typical use for relaxed memory ordering is updating counters, such as reference counters since this only requires atomicity, but neither ordering nor synchronization.

5.12.1.2 Release-Acquire Ordering

If an atomic store in thread A is tagged `memory_order_release` and an atomic load in thread B from the same variable is tagged `memory_order_acquire`, all memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* in thread B. That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.

The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable. Other threads can see a different order of memory accesses than either or both of the synchronized threads.

5.12.1.3 Sequentially Consistent Ordering

Atomic operations tagged `memory_order_seq_cst` order memory the same way as release/acquire ordering (everything that happened-before a store operation in one thread becomes a visible side effect in the thread that performed the load) and also establish a single total modification order of all atomic operations that are so tagged. Sequential ordering may be necessary for multiple producer-multiple consumer situations, where all consumers must observe the actions of all producers occurring in the same order.

Note: as soon as an atomic operation that does not use a memory order of **`memory_order_seq_cst`** is encountered, the sequential consistency is lost.

5.12.2 Memory Scope

The enumerated type `memory_scope` specifies whether the memory ordering constraints given by `memory_order` apply to threads within a SIMD-group, a threadgroup, or threads across threadgroups of a kernel(s) executing on the device. Its enumerated values are as follows:

```
enum memory_scope {memory_scope_simdgroup, memory_scope_threadgroup,
memory_scope_device};
```

The memory scope can be specified when performing atomic operations to `device` memory. Atomic operations to `threadgroup` memory only guarantee memory ordering in the threadgroup, not across threadgroups.

5.12.3 Fence Functions

For iOS, the following fence functions are supported.

```
void atomic_thread_fence(mem_flags flags, memory_order order)
```

```
void atomic_thread_fence(mem_flags flags, memory_order order, memory_scope
scope)
```

`atomic_thread_fence` establishes memory synchronization ordering of non-atomic and relaxed atomic accesses, as instructed by `order`, without an associated atomic function. For example, all non-atomic and relaxed atomic stores that happen before a memory_order_release fence in thread A will be synchronized with non-atomic and relaxed atomic loads from the same locations made in thread B after a memory_order_acquire fence. If there exist atomic functions X and Y, both operating on some atomic object M, such that the fence in thread A is sequenced before X, X modifies M, Y is sequenced before the fence in thread B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence, X would read if it were a release operation.

Depending on the value of order, this operation:

- has no effects, if order == memory_order_relaxed
- is an acquire fence, if order == memory_order_acquire
- is a release fence, if order == memory_order_release
- is both an acquire fence and a release fence, if order == memory_order_acq_rel
- is a sequentially consistent acquire and release fence, if order == memory_order_seq_cst

atomic_thread_fence imposes different synchronization constraints than an atomic store operation with the same memory_order. While an atomic store-release operation prevents all preceding writes from moving past the store-release, an atomic_thread_fence with memory_order_release ordering prevents all preceding writes from moving past all subsequent stores.

5.12.4 Atomic Functions

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by memory_order.

In the atomic functions described in section 5.12.4 and its subsections:

- 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.
- Functions that address device memory and do not have memory_scope argument have the same semantics as the corresponding functions with the memory_scope argument set to device. Functions that address threadgroup memory and do not have memory_scope argument have the same semantics as the corresponding functions with the memory_scope argument set to threadgroup.

Functions listed in section 5.12.4 and its subsections with names that end with _explicit are supported for all versions of Metal: atomic_store_explicit, atomic_load_explicit, atomic_exchange_explicit, atomic_compare_exchange_weak_explicit, atomic_fetch_key_explicit.

Other functions are only supported on ios-metal2.0: atomic_store, atomic_load, atomic_exchange, atomic_compare_exchange_weak, atomic_fetch_key.

5.12.4.1 Atomic Store Functions

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

For all versions of Metal, the following atomic store functions are supported. For all versions of Metal, `memory_order_relaxed` is supported for order. For `ios-metal2.0`, `memory_order_release` and `memory_order_seq_cst` are also supported. Behavior is undefined for `memory_order_acquire` and `memory_order_acq_rel`.

```
void atomic_store_explicit(threadgroup A* object,
                           C desired,
                           memory_order order)
void atomic_store_explicit(volatile threadgroup A* object,
                           C desired,
                           memory_order order)
void atomic_store_explicit(device A* object,
                           C desired,
                           memory_order order)
void atomic_store_explicit(volatile device A* object,
                           C desired,
                           memory_order order)
void atomic_store_explicit(device A* object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
void atomic_store_explicit(volatile device A* object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
```

For `ios-metal2.0`, the following atomic store functions are also supported. `memory_order_seq_cst` is the implied memory order.

```
void atomic_store(threadgroup A* object, C desired)
void atomic_store(volatile threadgroup A* object, C desired)
void atomic_store(device A* object, C desired)
void atomic_store(volatile device A* object, C desired)
```

5.12.4.2 Atomic Load Functions

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

For all versions of Metal, the following atomic load functions are supported. For all versions of Metal, `memory_order_relaxed` is supported for order. For `ios-metal2.0`, `memory_order_acquire` and `memory_order_acq_rel` are also supported. Behavior is undefined for `memory_order_release` and `memory_order_seq_cst`.

```
C atomic_load_explicit(const threadgroup A* object,
                       memory_order order)
C atomic_load_explicit(const volatile threadgroup A* object,
                       memory_order order)
C atomic_load_explicit(const device A* object,
                       memory_order order)
C atomic_load_explicit(const volatile device A* object,
                       memory_order order)
C atomic_load_explicit(const device A* object,
                       memory_order order,
                       memory_scope scope)
C atomic_load_explicit(const volatile device A* object,
                       memory_order order,
                       memory_scope scope)
```

For `ios-metal2.0`, the following atomic load functions are also supported. `memory_order_seq_cst` is the implied memory order.

```
C atomic_load(const threadgroup A* object)
C atomic_load(const volatile threadgroup A* object)
C atomic_load(const device A* object)
C atomic_load(const volatile device A* object)
```

5.12.4.3 Atomic Exchange Functions

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

For all versions of Metal, the following atomic exchange functions are supported. For all versions of Metal, `memory_order_relaxed` is supported for `order`. For `ios-metal2.0`, all `memory_order` values are available.

```
C atomic_exchange_explicit(threadgroup A* object,
                           C desired,
                           memory_order order)
C atomic_exchange_explicit(volatile threadgroup A* object,
```

```
                               C desired,
                               memory_order order)
C atomic_exchange_explicit(device A* object,
                               C desired,
                               memory_order order)
C atomic_exchange_explicit(volatile device A* object,
                               C desired,
                               memory_order order)
C atomic_exchange_explicit(device A* object,
                               C desired,
                               memory_order order,
                               memory_scope scope)
C atomic_exchange_explicit(volatile device A* object,
                               C desired,
                               memory_order order,
                               memory_scope scope)
```

For `ios-metal2.0`, the following atomic exchange functions are also supported. `memory_order_seq_cst` is the implied memory order.

```
C atomic_exchange(threadgroup A* object, C desired)
C atomic_exchange(volatile threadgroup A* object, C desired)
C atomic_exchange(device A* object, C desired)
C atomic_exchange(volatile device A* object, C desired)
```

5.12.5.3 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));
```

For all versions of Metal, the following atomic exchange functions are supported. If the comparison is `true`, memory access is affected according to the value of `success`, and if the comparison is `false`, memory access is affected according to the value of `failure`.

For all versions of Metal, `memory_order_relaxed` is supported for success and failure. For `ios-metal2.0`, all `memory_order` values are available for `success` and `failure` with the following restrictions. The `failure` argument cannot be `memory_order_release` or `memory_order_acq_rel`. The `failure` argument cannot be stronger than the `success` argument.

```
bool atomic_compare_exchange_weak_explicit(threadgroup A* object,
                                            C *expected,
                                            C desired,
                                            memory_order success,
                                            memory_order failure)
bool atomic_compare_exchange_weak_explicit(volatile threadgroup A* object,
                                           C *expected,
                                           C desired,
                                           memory_order success,
                                           memory_order failure)
bool atomic_compare_exchange_weak_explicit(device A* object,
                                           C *expected,
                                           C desired,
                                           memory_order success,
                                           memory_order failure)
bool atomic_compare_exchange_weak_explicit(volatile device A* object,
                                           C *expected,
                                           C desired,
                                           memory_order success,
                                           memory_order failure)
bool atomic_compare_exchange_weak_explicit(device A* object,
                                           C *expected,
                                           C desired,
                                           memory_order success,
                                           memory_order failure,
                                           memory_scope scope)
bool atomic_compare_exchange_weak_explicit(volatile device A* object,
```

```
C *expected,
C desired,
memory_order success,
memory_order failure,
memory_scope scope)
```

For `ios-metal2.0`, the following atomic compare-and-exchange functions are also supported. `memory_order_seq_cst` is the implied memory order.

```
bool atomic_compare_exchange_weak(threadgroup A* object, C *expected, C
desired)
```

```
bool atomic_compare_exchange_weak(volatile threadgroup A* object, C
*expected, C desired)
```

```
bool atomic_compare_exchange_weak(device A* object, C *expected, C desired)
```

```
bool atomic_compare_exchange_weak(volatile device A* object, C *expected, C
desired)
```

5.12.5.4 Atomic Fetch and Modify Functions

The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic type. The key, operator, and computation correspondence is given in Table 33.

Table 33 Atomic Operation Function

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

Atomically replaces the value pointed to by `object` with the result of the computation of the value specified by `key` and `operator`. These operations are atomic read-modify-write operations. For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow. There are no undefined results. Returns the value that `object` held previously.

For all versions of Metal, the following atomic fetch/modify functions are supported. For all versions of Metal, `memory_order_relaxed` is supported for order. For `ios-metal2.0`, all memory_order values are available.

```
C atomic_fetch_key_explicit(threadgroup A* object,
                            M operand,
                            memory_order order)
C atomic_fetch_key_explicit(volatile threadgroup A* object,
                            M operand,
                            memory_order order)
C atomic_fetch_key_explicit(device A* object,
                            M operand,
                            memory_order order)
C atomic_fetch_key_explicit(volatile device A* object,
                            M operand,
                            memory_order order)
C atomic_fetch_key_explicit(device A* object,
                            M operand,
                            memory_order order,
                            memory_scope scope)
C atomic_fetch_key_explicit(volatile device A* object,
                            M operand,
                            memory_order order,
                            memory_scope scope)
```

For `ios-metal2.0`, the following atomic fetch/modify functions are also supported. `memory_order_seq_cst` is the implied memory order.

```
C atomic_fetch_key(threadgroup A* object, M operand)
C atomic_fetch_key(volatile threadgroup A* object, M operand)
C atomic_fetch_key(device A* object, M operand)
C atomic_fetch_key(volatile device A* object, M operand)
```

5.13 SIMD-group Functions

The SIMD-group functions in Table 34 are supported by kernel and fragment functions. These functions allow threads in a SIMD-group to share data without the use of threadgroup memory or require any synchronization operations such as a barrier. 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.

SIMD-groups and SIMD-group functions are only supported for `macos-metal2.0`. SIMD-group functions are defined in the header `<metal_simdgroup>`.

Table 34 SIMD-group Functions in the Metal Standard Library

Built-in SIMD-group functions	Description
`T simd_shuffle(T data, ushort simd_lane_id)`	Returns the value of data specified by 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_broadcast(T data, ushort broadcast_lane_id)`	Broadcast the value of data specified by thread whose SIMD lane ID is `broadcast_lane_id`. `broadcast_lane_id` must be a valid SIMD lane ID and must be the same for all threads in a SIMD-group; otherwise the behavior is undefined.
`T simd_shuffle_up(T data, ushort delta)`	Returns the value of `data` specified by thread whose SIMD lane ID is computed by subtracting delta from the caller's SIMD lane ID. The value of `data` specified by the resulting SIMD lane ID is returned. The computed SIMD lane ID will not wrap around the value of the SIMD-group size so the lower `delta` lanes will 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_down(T data, ushort delta)`	Returns the value of `data` specified by thread whose SIMD lane ID is computed by adding delta to the caller's SIMD lane ID. The value of `data` specified by the resulting SIMD lane ID is returned. The computed SIMD lane ID will not wrap around the value of the SIMD-group size so the upper `delta` lanes will 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)`	Returns the value of `data` specified by thread whose SIMD lane ID is computed by performing a bitwise XOR of the caller's SIMD lane ID and `mask`. The value of `data` specified by the resulting SIMD lane ID is returned. The value of `mask` must be the same for all threads in a SIMD- group; otherwise the behavior is undefined.

`T` is one of the scalar or vector integer or floating-point types.

Let's take a look at examples that start with the following threadgroup:

SIMD Lane ID	**0**	**1**	**2**	**3**	**4**	**5**
`data`	a	b	c	d	e	f

`simd_shuffle_up()` shifts up each threadgroup by the `delta` number of threads. If `delta` is 2, the resulting computed SIMD lane IDs are shifted 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 remain unchanged.

Computed SIMD Lane ID	**-2**	**-1**	**0**	**1**	**2**	**3**
valid	0	0	1	1	1	1
`data`	a	b	a	b	c	d

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 are shifted 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 remain unchanged.

Computed SIMD Lane ID	**2**	**3**	**4**	**5**	**6**	**7**
valid	1	1	1	1	0	0
`data`	c	d	e	f	e	f

Below is an example of how these SIMD functions can be used 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 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=s/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);
}
```

5.14 Quad-group Functions

A quad-group function is a SIMD-group function (see section 5.13) with an execution width of 4. Quad-group functions (listed in Table 35) are supported by kernel and fragment functions.

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

Quad-group functions are only supported for `ios-metal2.0`.

Table 35 Quad-group Functions in the Metal Standard Library

Built-in SIMD-group functions	Description
`T quad_shuffle(T data, ushort quad_lane_id)`	Returns the value of data specified by 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_broadcast(T data, ushort broadcast_lane_id)`	Broadcast the value of data specified by thread whose quad lane ID is `broadcast_lane_id`. `broadcast_lane_id` must be a valid quad lane ID and must be the same for all threads in a quad-group; otherwise the behavior is undefined.
`T quad_shuffle_up(T data, ushort delta)`	Returns the value of `data` specified by thread whose quad lane ID is computed by subtracting delta from the caller's quad lane ID. The value of `data` specified by the resulting quad lane ID is returned. The computed quad lane ID will not wrap around the value of the quad-group size so the lower `delta` lanes will 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_down(T data, ushort delta)`	Returns the value of `data` specified by thread whose quad lane ID is computed by adding delta to the caller's quad lane ID. The value of `data` specified by the resulting quad lane ID is returned. The computed quad lane ID will not wrap around the value of the quad-group size so the upper `delta` lanes will 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)`	Returns the value of `data` specified by thread whose quad lane ID is computed by performing a bitwise XOR of the caller's quad lane ID and `mask`. The value of `data` specified by the resulting quad lane ID is returned. The value of `mask` must be the same for all threads in a quad- group; otherwise the behavior is undefined.

`T` is one of the scalar or vector integer or floating-point types.

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

Let's take a look at examples that start with the following threadgroup:

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

`quad_shuffle_up()` shifts up each threadgroup by the `delta` number of threads. If `delta` is 2, the resulting computed quad lane IDs are shifted 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 resulting computed quad lane IDs are shifted 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

5.15 Imageblock Functions

This section lists the Metal member functions for imageblocks. (For more on the imageblock data type, see section 2.10.)

The following member functions query information about the imageblock:

```
ushort get_width() const;
ushort get_height() const;
ushort get_num_samples() const;
```

The following member function is used 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 return the color coverage mask (i.e., 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`.

5.15.1 Functions for Imageblocks with Implicit Layout

The following functions can be used to read/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);
```

The following member function can be used to read/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);
}
```

The following member function can be used to write an imageblock with a color coverage mask. This member function must be used when writing to an imageblock at color rate.

```
void write(T data, ushort2 coord, ushort color_coverage_mask);
```

The following member functions are used to get a region of a slice for a given data member in the imageblock. This is used 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 struct type specified in `imageblock<T>` with the `[[color(n)]]` attribute where `n` is `color_index`. `origin` is the (x, y) coordinate in the slice. `size` is the actual size of the slice to be copied.

```
const imageblock_slice<E, imageblock_layout_implicit> slice(ushort
color_index) const;
```

```
const imageblock_slice<E, imageblock_layout_implicit> slice(ushort
color_index, ushort2 origin, ushort2 size) const;
```

The `slice(…)` member function that does not take origin and size as arguments specifies the region to copy with an origin of (0,0) and the width and height of the imageblock.

5.15.2 Functions for Imageblocks with Explicit Layout

The following member functions are used to get a reference to the imageblock data for a specific location given by an (x, y) coordinate inside the imageblock. These member functions should be used 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;
```

The following member functions are used 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. These member functions should be used 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 multi-sampled 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;
     …
}
```

The following `write` member function can be used to write an imageblock with a color coverage mask. This member function must be used when writing to an imageblock at color rate.

```
void write(T data, ushort2 coord, ushort color_coverage_mask);
```

The following `slice` member functions are used to get a region of a slice for a given data member in the imageblock struct. This is used to write data associated with a specific data member described in the imageblock struct for all threads in the threadgroup to a specified region in a texture.

`data_member` is a data member declared in the struct type specified in `imageblock<T>`. `origin` is the (x, y) coordinate in the slice. `size` is the actual size of the slice to be copied.

```
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 origin, ushort2
size ) const;
```

The `slice(…)` member function that does not take `origin` and `size` as arguments specifies the region to copy with an origin of (0,0) and the width and height of the imageblock.

5.15.3 Writing an Imageblock Slice to a Region in a Texture

The following `write(…)` member function in these texture types are used to write pixels associated with a slice in the imageblock to a texture starting at location given by `coord`.

For 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 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 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 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 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 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 cube array texture:

```
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 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 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 have finished 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 pixels in imageblock - update the elements in slice
    process_pixels_in_imageblock(img_blk, gid, lid);

    // a barrier to make sure all threads have finished writing to the
    elements in the imageblock
    threadgroup_barrier(mem_flags::mem_threadgroup_imageblock);

    // write a specific element in imageblock to output image
    // only 1 thread in the threadgroup performs the imageblock write
    if (lid.x == 0 && lid.y == 0)
        dst.write(img_blk.slice(f->a), gid);
}
```

5.16 Diagnostics

The header `<metal_assert>` provides a macro for documenting Metal function assertions and a mechanism for disabling the assertion checks.

```
#if defined(NDEBUG)
#define assert(condition) ((void)0)
#else
#define assert(condition) /* implementation defined */
#endif
```

The behavior of the `assert` macro depends on whether the debug macro is defined or not. If `NDEBUG` is defined as a macro name at the point in the source code where `<metal_assert>` is included, then `assert` does nothing.

If `NDEBUG` is not defined, then `assert` checks if its argument, which must have scalar type, compares equal to zero. If it does, `assert` outputs implementation-specific diagnostic information on the standard error output and aborts the function execution. The diagnostic information is required to include the text of expression, as well as the values of the standard macros `__FILE__`, `__LINE__`, and the standard variable `__func__`.

6 Compiler Options

The Metal compiler can be used online (i.e. using the appropriate APIs to compile Metal sources) or offline. Metal sources compiled offline can be loaded as binaries, using the appropriate Metal APIs.

This chapter explains the compiler options supported by the Metal compiler, which are categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. The online and offline Metal compiler support these options.

6.1 Pre-Processor Compiler Options

These 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 may receive multiple options, which are processed in the order in which they appear. This option allows developers to compile Metal code to change which features are enabled or disabled.

```
-I dir
```

Add the directory *dir* to the list of directories to be searched for header files. This option is only available for the offline compiler.

6.2 Math Intrinsics Compiler Options

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

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

These options enable (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.
- 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 zero argument or result as insignificant.
- Allow Reciprocal – Allow optimizations to use the reciprocal of an argument rather than perform division
- Fast – Allow algebraically equivalent transformations i.e. re-associate floating-point operations that may dramatically change results in floating-point.

6.3 Compiler Options Controlling the Language Version

The following option controls the version of unified graphics/compute language that the compiler accepts.

```
-std=
```

Determine the language revision to use. A value for this option must be provided. This can only be:

- `ios-metal1.0` – support the unified graphics / compute language revision 1.0 programs for iOS 8.0.
- `ios-metal1.1` – support the unified graphics / compute language revision 1.1 programs for iOS 9.0.
- `ios-metal1.2` – support the unified graphics / compute language revision 1.2 programs for iOS 10.0.
- `ios-metal2.0` – support the unified graphics / compute language revision 2.0 programs for iOS 11.0.
- `osx-metal1.1` – support the unified graphics / compute language revision 1.1 programs for macOS 10.11.
- `osx-metal1.2` – support the unified graphics / compute language revision 1.2 programs for macOS 10.12.
- `osx-metal2.0` – support the unified graphics / compute language revision 2.0 programs for macOS 10.13.

6.4 Compiler Options to Request or Suppress Warnings

The following options are available.

```
-Werror
```

Make all warnings into errors.

```
-W
```

Inhibit all warning messages.

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-precision and half-precision floating-point numbers.

NaNs must be supported for single-precision 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-precision or half-precision floating-point numbers passed as input to or produced as the output of single-precision 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 precision 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 36 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 36 Minimum Accuracy of Single-Precision Floating-Point Operations and Functions

Math Function	Min Accuracy - ULP values
x + y	Correctly rounded
x - y	Correctly rounded
x * y	Correctly rounded
1.0 / x	Correctly rounded

Math Function	Min Accuracy - ULP values
x / y	Correctly rounded
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

Math Function	Min Accuracy - ULP values
log2	<= 4 ulp
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 37 describes the minimum accuracy of single-precision floating-point arithmetic operations given as ULP values with fast math enabled (which is the default unless `-ffast-math-disable` is specified as a compiler option).

Table 37 Minimum Accuracy of Single-Precision Operations and Functions with Fast Math Enabled

Math Function	Min 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}
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 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 atan(y / x) for x > 0, atan(y / x) + M_PI_F for x < 0 and y > 0, atan(y / x) – M_PI_F for x < 0 and y < 0 and is undefined if y = 0 and x = 0.
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)	The maximum relative error is <= 2^{-17}.
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	0 ulp
fract	Correctly rounded
frexp	0 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <= 3 ulp if x > 0; otherwise the results are undefined.

log2(x)	For x in the domain [0.5, 2], the maximum absolute error is <= 2^{-22}; otherwise the maximum error is <= 2 ulp if x > 0; 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)	The maximum relative error is <= 2^{-17}.
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)	The maximum relative error is <= 2^{-17}.
trunc	Correctly rounded

Table 38 describes the minimum accuracy of half-precision floating-point basic arithmetic operations and math functions given as ULP values. Table 38 only applies to iOS with A11 hardware.

Table 38 Minimum Accuracy of Half Precision Floating-Point Operations and Functions

Math Function	Min Accuracy - ULP values
x + y	Correctly rounded
x - y	Correctly rounded
x * y	Correctly rounded

Math Function	Min Accuracy - ULP values
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
ilogb	0 ulp
ldexp	Correctly rounded
log(x)	<= 1 ulp
log2(x)	<= 1 ulp

Math Function	Min Accuracy - ULP values
log10(x)	<= 1 ulp
modf	0 ulp
pow(x, y)	<= 2 ulp
powr(x, y)	<= 2 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

NOTE: Even though the precision of individual math operations and functions are specified in Tables 36, 37, and 38, the Metal compiler, in fast math mode, may re-associate floating-point operations that may dramatically change results in floating-point. Re-association 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 used by 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 denormals 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 (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 (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. The round to nearest even or round to zero rounding mode is used 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. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may not be flushed to zero.

When converting a floating-point type to an integer type, if the floating-point value is NaN, the integer result 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, the texture coordinate must refer to a region inside the texture for the texture `read` and `write` functions.

In the sections that follow, we discuss conversion rules that are applied when reading and writing textures in a graphics or kernel function. When a multisample resolve operation is performed, the conversion rules described in this section do not apply.

7.7.1 Conversion Rules for Normalized Integer Pixel Data Types

In this section we discuss 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-bit, 10-bit or 16-bit normalized unsigned integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit 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-bit or 16-bit normalized signed integer pixel values, the texture sample and read functions convert the pixel values from an 8-bit 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 39. The precision of the conversion rules are guaranteed to be <= 1.5 ulp except for the cases described in the third column.

Table 39 Rules for 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-bit, 10-bit 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-bit or 16-bit unsigned integer.

For textures that have 8-bit 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-bit 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 40.

Table 40 Rules for 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)` $i_{0:0}$ = int_{RTNE}(x)
2-bit normalized unsigned integer	`x = min(max(f * 3.0, 0.0), 3.0)` $i_{1:0}$ = int_{RTNE}(x)
4-bit normalized unsigned integer	`x = min(max(f * 15.0, 0.0), 15.0)` $i_{3:0}$ = int_{RTNE}(x)
5-bit normalized unsigned integer	`x = min(max(f * 31.0, 0.0), 31.0)` $i_{4:0}$ = int_{RTNE}(x)
6-bit normalized unsigned integer	`x = min(max(f * 63.0, 0.0), 63.0)` $i_{5:0}$ = int_{RTNE}(x)
8-bit normalized unsigned integer	`x = min(max(f * 255.0, 0.0), 255.0)` $i_{7:0}$ = int_{RTNE}(x)
10-bit normalized unsigned integer	`x = min(max(f * 1023.0, 0.0), 1023.0)` $i_{9:0}$ = int_{RTNE}(x)
16-bit normalized unsigned integer	`result = min(max(f * 65535.0, 0.0), 65535.0)` $i_{15:0}$ = int_{RTNE}(x)
8-bit normalized signed integer	`result = min(max(f * 127.0, -127.0), 127.0)` $i_{7:0}$ = int_{RTNE}(x)
16-bit normalized signed integer	`result = min(max(f * 32767.0, -32767.0),32767.0)` $i_{15:0}$ = int_{RTNE}(x)

In Metal Shading Language 2.0, the following restriction has been removed:

The GPU may choose to approximate the rounding mode used in the conversions from floating-point to integer value described in the table above. If a rounding mode other than round to nearest even is used, the absolute error of the implementation dependent rounding mode vs. the result produced by the round to nearest even rounding mode must be <= 0.6.

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.
- Denorms may be flushed to zero.
- All other values must be preserved.

7.7.4 Conversion Rules for 11-bit and 10-bit Floating-Point Pixel Data Type

The floating-point formats use 5 bits for the exponent, 5 bits of mantissa for the 10-bit floating-point types and 6-bits of mantissa for the 11-bit floating-point types with an additional hidden bit for both types. There is no sign bit. The 10-bit and 11-bit floating-point types preserve denorms.

These floating-point formats use the following rules:

- If exponent = 0 and mantissa = 0, the floating-point value is 0.0.
- If exponent = 31 and mantissa != 0, the resulting floating-point value is a NaN.
- If exponent = 31 and mantissa = 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 exponent = 0 and mantissa != 0, the floating-point value is a denormal value given as 2 ^ (exponent – 14) * (mantissa / N)

N is 32 if mantissa is 5-bits and is 64 if mantissa is 6-bits.

Conversion of a 11-bit or 10-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

11-bit or 10-bit floating-point value must be <= 0.5 ULP. Any operation that would result 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 use 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 is 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).

- Components r, g and b are first clamped (in the process, mapping NaN to zero) as follows:
  $r_c = max(0, min(sharedexp_{max}, r)$
  $g_c = max(0, min(sharedexp_{max}, g)$
  $b_c = max(0, min(sharedexp_{max}, b)$
  where $sharedexp_{max} = ((2^N - 1)/2^N) * 2(E_{max} - B)$
- The largest clamped component max_c, is determined:
  $max_c = max(r_c, g_c, b_c)$

- A preliminary shared exponent exp_p is computed:
  $exp_p = max(-B - 1, floor(log_2(max_c)) + 1 + B$
- A refined shared exponent exp_s is computed:
  $max_s = floor((max_c / 2^{exp_p - B - N}) + 0.5f)$
  $exp_s = exp_p, \text{ if } 0 <= max_s < 2^N$, and
  $= exp_p + 1, \text{ if } max_s = 2^N.$
- Finally, three integer values in the range 0 to $2^N - 1$ are computed:
  $r_s = floor(r_c / 2^{exp_p - B - N}) + 0.5f)$
  $g_s = floor(g_c / 2^{exp_p - B - N}) + 0.5f)$
  $b_s = floor(b_c / 2^{exp_p - B - N}) + 0.5f)$

Conversion of a half or single precision floating-point color values to the RGB9E5 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 8-bit 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 41.

Table 41 Rules for Conversion Between Integer Pixel Data Types

Convert from	Convert 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 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 `result` is converted back to an un-normalized sRGB value but without rounding to a 8-bit unsigned integer value (call it `r`) and the original sRGB 8-bit unsigned integer color value (call it r_{orig}) is <= 0.5 i.e.

$$fabs(r - r_{orig}) <= 0.5$$

The following are the conversion 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 i.e. c = c * 255.0
convert to integer:

c = c + 0.5
drop the decimal fraction, and the remaining
floating-point(integral) value is converted
directly to an integer.
```

The precision of the above conversion should be such that

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

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010