Article

Creating threads and threadgroups

Learn how Metal organizes compute-processing workloads.

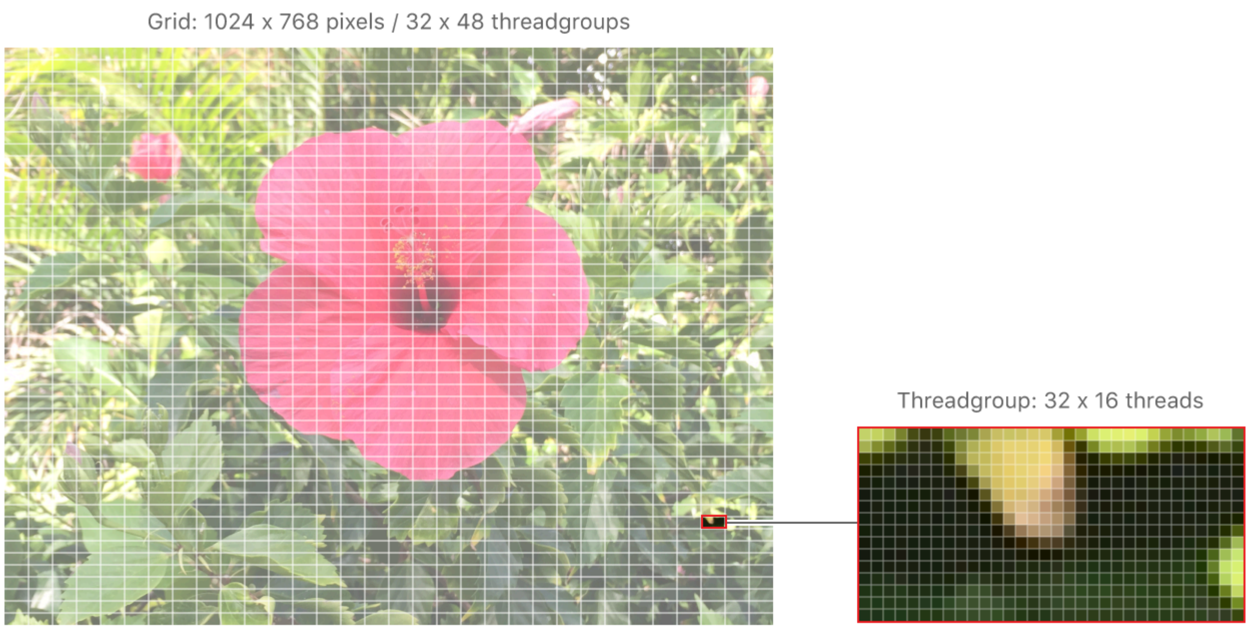
Overview

Recall from Processing a texture in a compute function that when you dispatch your compute pass, Metal executes your kernel function over a 1D, 2D, or 3D grid. Each point in the grid represents a single instance of your kernel function, referred to as a *thread*. For example, in image processing, the grid is typically a 2D matrix of threads—representing the entire image—with each thread corresponding to a single pixel of the image being processed.

Threads are organized into *threadgroups* that are executed together and can share a common block of memory. While sometimes kernel functions are designed so that threads run independently of each other, it's also common for threads in a threadgroup to collaborate on their working set.

Identification of threads by position in grid

Figure 1 shows how an image being processed by a compute kernel is divided into threadgroups and how each threadgroup is composed of individual threads. Each thread processes a single pixel.

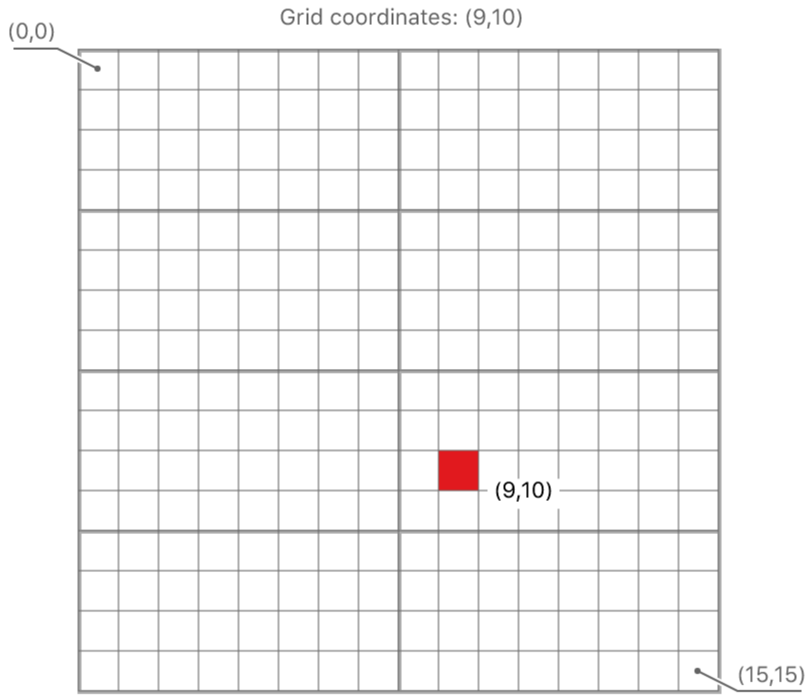

Grid: 1024 x 768 pixels / 32 x 48 threadgroups
Threadgroup: 32 x 16 threads

A thread can be identified by its position in the grid; it's this unique position that allows your kernel function to do something different for each thread. The sample kernel function in Processing a texture in a compute function, below, shows how a thread's position in the grid is passed into the function as a parameter. In this case, the parameter, `gid`, is a vector representing 2D coordinates and is used to both read from and write to a particular location in a texture.

```
kernel void
grayscaleKernel(texture2d<half, access::read>  inTexture  [[texture(AAPLTextureInde
                texture2d<half, access::write> outTexture [[texture(AAPLTextureInde
                uint2                          gid        [[thread_position_in_grid
{
    if((gid.x >= outTexture.get_width()) || (gid.y >= outTexture.get_height()))
    {
        return;
    }
    half4 inColor  = inTexture.read(gid);
    half  gray     = dot(inColor.rgb, kRec709Luma);
    outTexture.write(half4(gray, gray, gray, 1.0), gid);
}
```

`[[thread_position_in_grid]]` is an *attribute qualifier*. Attribute qualifiers, identifiable by their double square-bracket syntax, allow kernel parameters to be bound to resources and built-in variables, in this case the thread's position in the grid to the kernel function.

For example, given a grid of 16 x 16 threads partitioned into 2 x 4 threadgroups of 8 x 4 threads, a single thread (shown in Figure 2 in red) has a position in the grid of (9,10):

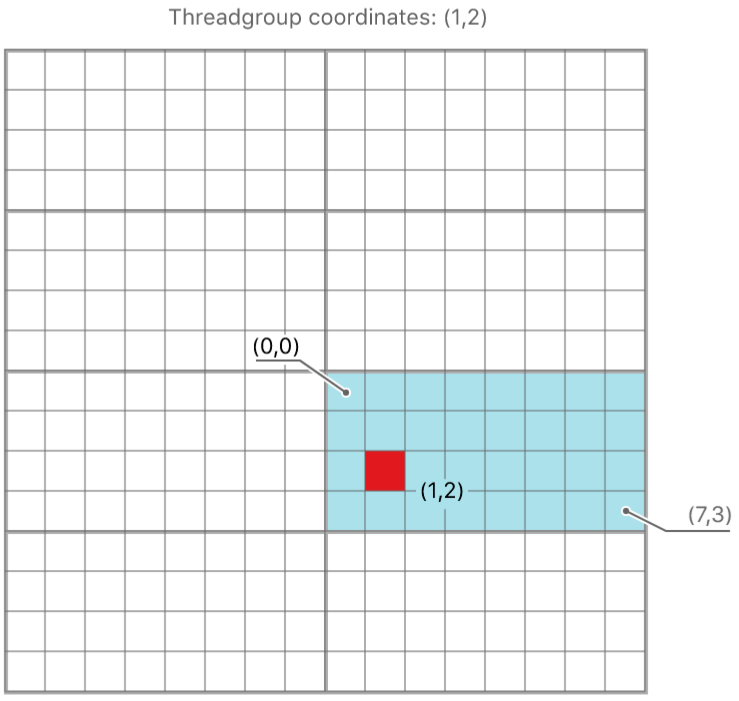


Identification of threads by position in threadgroup

A thread's position in its threadgroup is also available as the attribute qualifier `[[thread_position_in_threadgroup]]`, and a threadgroup's position in the grid is available as `[[threadgroup_position_in_grid]]`.

Depending on the shape of the grid, these position attributes are either a scalar value, or a two- or three-element vector. In the case of a 2D grid, position attributes are two-element vectors, with the origin at the top-left.

The thread identified in Figure 2 is in the threadgroup with a position in the grid of (1,2), and its position in that threadgroup is (1,2), as shown in Figure 3:



Using the following code, you can also calculate a thread's position in the grid based on its position in its threadgroup and that threadgroup's size and position in the grid:

```
kernel void
myKernel(uint2 threadgroup_position_in_grid   [[ threadgroup_position_in_grid ]],
         uint2 thread_position_in_threadgroup [[ thread_position_in_threadgroup ]],
         uint2 threads_per_threadgroup        [[ threads_per_threadgroup ]])
{

    uint2 thread_position_in_grid =
        (threadgroup_position_in_grid * threads_per_threadgroup) +
        thread_position_in_threadgroup;
}
```

SIMD groups

The threads in a threadgroup are further organized into single-instruction, multiple-data (SIMD) groups, also known as *warps* or *wavefronts*, that execute concurrently. The threads in a SIMD group execute the same code. Avoid writing code that could cause your kernel function to *diverge*; that is, to follow different code paths. A typical example of divergence is caused by using an *if* statement. Even if a single thread in a SIMD group takes a different path from the others, all threads in that group execute both branches, and the execution time for the group is the sum of the execution time of both branches.

The division of threadgroups into SIMD groups is defined by Metal. It remains constant 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.

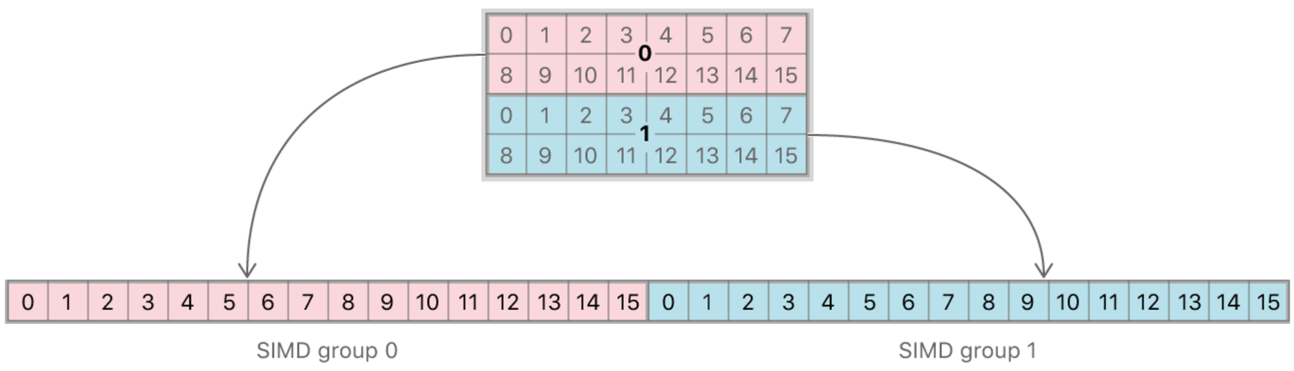
The number of threads in a SIMD group is returned by the `threadExecutionWidth` of your compute pipeline state object. Attribute qualifiers allow you to access a SIMD group's scalar index within a threadgroup, and a thread's scalar index within a SIMD group:

`[[simdgroup_index_in_threadgroup]]`
: The unique scalar index of a SIMD group in its threadgroup.

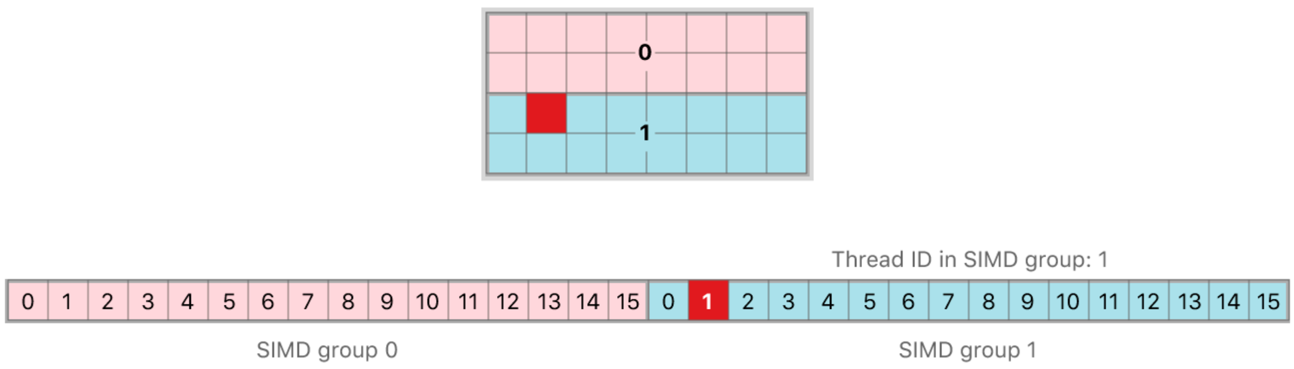
`[[thread_index_in_simdgroup]]`
: The unique scalar index of a thread in its SIMD group, also known as the *lane ID*.

Although threadgroups can be multidimensional, SIMD groups are 1D. Therefore, a thread's position within a SIMD group is a scalar value for all threadgroup shapes. The SIMD group size remains constant and is unaffected by the threadgroup size.

For example, using the same 16 x 16 grid shown in Figure 2, with a thread execution width of 16, a single 8 x 4 threadgroup consists of 2 SIMD groups. Because a SIMD group contains 16 threads, each SIMD group constitutes 2 rows in the threadgroup:



The thread shown in red in Figure 5 has a `[[simdgroup_index_in_threadgroup]]` value of 1 and a `[[thread_index_in_simdgroup]]` value of 1:



See Also

Encoding a compute pass

Calculating threadgroup and grid sizes

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

`protocol MTL4ComputeCommandEncoder`

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

`protocol MTLComputeCommandEncoder`

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