Article

Porting your Metal code to Apple silicon

Create a version of your Metal app that runs on both Apple silicon and Intel-based Mac computers.

Overview

How you port a Metal app to a Mac with Apple silicon depends on whether your project supports Apple family GPUs. If your project supports iOS or tvOS, it also supports Apple family GPUs. If your codebase is macOS-only, you may find situations where Apple family GPUs behave differently from the GPUs in Intel-based Macs. This article describes how to port your app so it works across GPU families.

Before you port your app, test run it under Rosetta translation. When you run an app linked against macOS 10.15 or earlier under Rosetta translation, Metal supports backward compatibility through software workarounds for common programming errors; these workarounds trade some GPU performance for behavior that's more consistent with Intel-based Macs. To expedite fixing your app, this article describes common porting issues that you may encounter. For some issues, Metal doesn't automatically apply a workaround because the performance penalty is too large. If you expect your app to run under translation, fix these issues first.

When you're ready to port your app, turn on Metal diagnostic tools in Xcode to search for problems in your code. After you fix these problems, recompile your app using Xcode 12, as described in Building a universal macOS binary. Then, address the list of porting issues described in this article to ensure that your code accounts for these. Once your app runs correctly, analyze its performance and adjust your rendering strategy to take advantage of Apple family GPUs.

Identify and Correct Potential Porting Issues

Xcode can help validate whether your app uses Metal correctly. Although Xcode can't find every problem, fixing common problems before porting your app makes the porting process easier. Once you start porting your app, use Xcode to verify code changes and understand how your code runs on Apple family GPUs.

- Use API validation to find common Metal framework programming errors. When API validation is on, Xcode stops in the debugger when you incorrectly call the Metal API. For more information, see Metal developer workflows
- Use shader validation to recompile your shaders with additional debugging added. For example, shader validation stops in the debugger with an out-of-bounds error if you attempt to access memory outside your buffers. For more information, see Metal developer workflows.
- Capture a GPU trace using Metal Debugger to analyze a set of related Metal calls, such as all commands used to render a single frame of animation. Metal Debugger collects data from Metal API calls, analyzes the resulting GPU trace, and provides suggestions to fix errors or improve performance. For example, modern game engines require dozens of passes and thousands of GPU commands to render complex scenes. Metal Debugger can find dependencies between passes and uncover places where you incorrectly generate or store data. Not all of Metal Debugger's suggestions are problems you must fix. You decide whether or not to act on each suggestion. For more information, see Metal debugger.

Test for GPU Features Your App Uses

Metal collects sets of GPU features into *GPU families*, so you can test for a group of features using a single query. When capabilities vary and features aren't common enough to be part of a specific family, Metal provides additional queries to get detailed information about a GPU. You might need to test multiple GPU families, as well as specific Metal device object properties, to learn the capabilities of a GPU. Use that data to decide at runtime which strategy your app uses to leverage the GPU.

Previously, Apple GPUs and Mac GPUs belonged to distinct families, and each GPU only supported one family, so unless you designed a cross-platform app, you only checked for members of a single family. The GPU in a Mac with Apple silicon is a member of both GPU families, and supports both Mac family 2 and Apple family feature sets. Now, to support both Apple silicon and Intel-based Mac computers, test for both families in your app. Use a Mac family test to determine the major feature set that the computer supports. Test for an Apple GPU family where you can take advantage of features supported only on an Apple family GPU, or to apply optimizations specific to tile-based rendering. When a device object has a specific method or property to determine whether a feature is available, use that method or property instead of testing for GPU families.

Always use availability methods and properties to determine features, and don't rely on the GPU name or other hardcoded information. GPU names may change on future hardware, so the `name` property of the device object is an unreliable indicator of the feature set. Similarly, hard coding values for specific GPUs rather than using device queries can cause your app to crash, or prevent your app from taking advantage of new features.

For more information, see Detecting GPU features and Metal software versions. For a list of specific queries, see `MTLDevice`.

Set Load and Store Actions on Your Render Passes

Every render target in a render pass has a load and store action that indicates what you expect the texture contents to be at the start and end of the render pass, respectively. When you compile your app for macOS 11, you must choose the correct load and store actions for your render targets.

On Intel-based Macs, setting these actions incorrectly often has no effect, because the GPUs in those systems render directly to memory, and don't have dedicated stages that copy texture data between memory and the GPU. It's possible that you may have these actions incorrectly configured in your app.

Apple-family GPUs use tile memory inside the GPU to temporarily hold texture contents during the render pass. The load action determines whether the GPU copies a texture's existing contents into tile memory, and the store action determines whether it stores the tile contents back to memory. If you set either action to *don't care*, Apple family GPUs skip these memory operations to improve performance. If you do this unintentionally, you'll see visual artifacts in your rendered content.

> **Note**
>
> If your app is linked against macOS 10.15 or earlier, is running under Rosetta translation, and you set a load action to `MTLLoadAction.dontCare`, Metal forces the GPU to load the contents into tile memory, trading performance for behavior more consistent with Intel-based Macs.

For more information on load and store actions, see Setting load and store actions.

Make Vertex Shader Positions Invariant

Many rendering techniques require multiple render passes, where earlier passes generate data that's consumed by later render passes. Each pass uses a different vertex shader to perform its specific calculations. Such techniques often rely on different vertex shaders calculating vertex positions the same way. For example, an early pass might write depth values to a depth texture, and a later pass might test the positions it calculates against data stored in the depth texture. If the two shaders calculate the position data differently, those tests might fail.

If your app uses such a rendering technique, add the `invariant` attribute to the position output of every vertex shader used in the rendering technique, as shown in the code below:

```
typedef struct{
    float4 position [[position, invariant]];
    float4 color;
…
} RasterizerData;
```

Then, compile the shaders with the `preserve–invariance` flag set. If you instead compile shaders at runtime, specify the `preserveInvariance` option when creating your `MTLLibrary` object. Metal compiles these shaders more conservatively, ensuring that the GPU calculates positions marked with the `invariant` attribute the same way.

When Metal compiles vertex shaders, the compiler optimizes the shader for performance, rather than strictly enforcing consistency and precision across different shaders. The compiler may emit different instruction sequences or merge floating-point operations together, which can slightly change how the GPU calculates each result; this compiler behavior depends on the complete source code in each shader. The compiler for Apple family GPUs applies optimizations aggressively, so mark the position outputs as invariant, or you may see visual artifacts in places that you don't typically see on other GPUs.

> **Note**
>
> If your app is linked against macOS 10.15 or earlier and is running under Rosetta translation, Metal compiles vertex positions as if the `invariant` keyword were present, trading performance for behavior more consistent with Intel-based Macs.

Bind Textures Once For Write Access

Metal doesn't permit you to bind the same texture to multiple arguments of a shader if any those arguments can write to the texture (`access::write` or `access:: read_write`). If you need both read and write access, you must bind the texture to a single argument with the `access:: read_write` keyword, and synchronize access to the texture.

Similarly, if you assign a depth texture to a render pass and update its contents within that render pass, you can't bind the same texture as a shader input during the same render pass to read from the depth texture. If you do, you'll see inconsistent results across different GPUs.

To detect code that incorrectly binds textures, turn on Metal API validation in Xcode and run your app. Revise that code to perform the work in separate render passes, so updates to textures are complete before you read from them.

Note

If your app is linked against macOS 10.15 or earlier and is running under Rosetta translation, Metal takes snapshots of depth data before a render pass, redirecting any memory reads to the snapshot. This behavior comes with a cost in performance.

Check Depth and Stencil Texture Formats

Only some Mac family GPUs support combined depth and stencil formats. To test whether a particular device supports this format, read the `isDepth24Stencil8PixelFormatSupported` property on the MTLDevice.

Keep Memory Accesses Within Memory Boundaries

When you access device or threadgroup memory in a shader, you must stay within the bounds of the memory you're accessing. For example, when you access data in a buffer, you can't access memory before the start or past the end of the buffer. Metal doesn't define a specific GPU behavior when you access memory outside these boundaries, so if your app incorrectly accesses memory, you may see different behavior when you run your app on Apple family GPUs. An Apple family GPU can treat an incorrect memory access as a hardware fault, terminating the command buffer that caused the exception.

To test for memory access errors, turn on shader validation in Xcode and run your app. For more information, see Metal developer workflows.

Determine the SIMD Group Size at Runtime

In a compute shader, the SIMD group size, also called the thread execution width, is the number of threads that run together on the GPU. Metal Shading Language specifies some operations that are specific to SIMD groups, while other operations apply to the larger threadgroups; to implement some shaders efficiently, you need to know the SIMD group size.

The size of a SIMD group varies between different GPUs, particularly Mac GPUs. Don't assume the size of SIMD groups. At runtime, after you create a compute pipeline state object, read its `threadExecutionWidth` property to get the SIMD group size for that compute pipeline. To get the SIMD group size in your shader, declare an argument with the `threads_per_simdgroup` attribute, instead.

Synchronize Memory Operations in Shaders

GPUs execute memory accesses on different threads in an unpredictable order. If a shader reads and writes the same memory locations, and you need the GPU to execute those actions in a specific order, you must include barriers between the memory operations. Metal requires barriers even in cases where the threadgroup size is the same as SIMD group size. Apple GPUs aggressively optimize memory operations to improve performance; if you don't have proper synchronization in your shaders, you may corrupt memory or get incorrect results.

Call the `threadgroup_barrier` function in your shader to force all threads in a threadgroup to reach that function call before allowing any threads to continue past it. Specify a flag parameter to determine how the GPU synchronizes memory operations that your shader previously submitted. For example, the following call prevents threads in a threadgroup from moving past the barrier until all device memory accesses from the threadgroup have completed.

```
threadgroup_barrier(mem_device);
```

If you only need to synchronize threads in a SIMD group, instead use the `simdgroup_barrier` function. The following call prevents threads in each SIMD group from moving past the barrier until all previous threadgroup memory accesses made by the SIMD group have completed.

```
simd_barrier(mem_threadgroup);
```

For more information, see section 6.8.1 of Metal Shading Language Specification.

Synchronize Concurrent Compute Dispatches

If you configure a `MTLComputeCommandEncoder` for concurrent dispatch, Metal doesn't perform any automatic synchronization between commands that the encoder encodes. Apple family GPUs can aggressively execute commands at the same time, so if you aren't correctly synchronizing commands, the GPU may run commands in the wrong order, which can corrupt memory or cause a fault.

For more information on how to synchronize concurrent commands, see `memoryBarrier(_:)`.

Synchronize Untracked Heaps

When you use an untracked heap, Metal doesn't track resource dependencies or synchronize commands that access resources on that heap. Apple family GPUs take advantage of this behavior to improve performance, so if you aren't synchronizing commands correctly, the GPU may run commands in the wrong order, which can corrupt memory or cause a fault.

For more information on how to use heaps or fences to synchronize commands, see Resource synchronization.

Profile Your App

After you’ve got your app running correctly, profile and tune its performance for the Apple GPU.

When you design an app to run on Apple GPUs, it’s a good practice to do more work in each render pass. Small render passes that render just a few primitives can be inefficient because the GPU must repeatedly copy pixel data between tile memory and system memory. Coalesce multiple logical rendering passes into a single Metal rendering pass if a series of sequential render passes share a common set of render targets, so the GPU only needs to copy the data once.

Coalescing render passes is possible on Apple family GPUs where it might not on other GPUs, such as when writing a deferred renderer or when performing a series of post-processing effects to the same texture. In these cases, use programmable blending inside your fragment shaders to blend new pixel data with the current pixel data. The pixel data stays in tile memory until the end of the render pass. Apple GPUs also support tile shading to directly process pixel data stored in the tile using a compute shader that runs as part of a render pass. The combination of programmable blending and tile shading mean you can perform complex render operations while copying data between the GPU and system memory as little as possible.

For examples of these techniques, see Rendering a scene with deferred lighting in Objective-C and Rendering a scene with forward plus lighting using tile shaders.