The State of Vulkan on Apple Devices

Why you should be developing for the Apple ecosystem with Vulkan, and how to do it.

Richard S. Wright Jr., LunarG
June 3, 2021

If you are reading this, you likely already know what Vulkan is -- a low-overhead, high-performance, cross-platform, graphics and compute API designed to take maximum advantage of today's highly parallel and scalable graphics and compute hardware. To be sure, there are other APIs with this goal such as Microsoft's DirectX and Apple's Metal API. For a long time OpenGL matured to meet the needs of rapidly evolving hardware and met well the challenges of cross-platform development of high-performance graphics applications. OpenGL will remain viable for a very long time for many applications (and can actually be layered on top of Vulkan), but to take maximum advantage of modern hardware, a new and somewhat reinvented approach is called for. Everyone does not need the level of control that Vulkan provides, but if you do, you can think of Vulkan as a lower-level successor to OpenGL in that it attempts to enable highly portable, yet high-performance, application development across desktop and mobile device platforms.

Writing platform specific code does have its advantages, and vendors such as Microsoft and Apple certainly adore having best of breed titles available exclusively for their customers as it adds value to their ecosystem. For developers though, it is often in our best interests to make our products available as widely as possible. Aside from gaming consoles with their mostly closed development environments, Windows, Linux, Apple, and Android combined represent most of the market for developers of end user software and games today.

In the past, we used OpenGL to help with 3D technical applications and games that we wanted to work across platforms. As graphics and compute hardware has evolved, Vulkan has emerged as the new cross-platform API best suited to take maximum advantage of today's hardware capabilities, and Vulkan is now widely available with native support across a very heterogeneous development landscape and is natively supported on Windows, Linux, and Android operating systems. Just as Microsoft has its own low-level solution in the form of DirectX, Apple's answer to modern low-level, high-performance graphics is its own Metal API. Apple does not support a mechanism for an alternate low-level graphics driver. Unless this changes, or Apple adopts Vulkan directly, there cannot be a native Vulkan driver for Apple platforms, desktop or mobile. This does not mean however that high-performance Vulkan is not available on Apple hardware.

Vulkan on Metal

The Vulkan Portability Initiative is an effort by the Khronos standards organization to promote and standardize Vulkan implementations that are layered on other low-level APIs. This is a common approach when a low-level driver for an API is not available. Vulkan and Apple's Metal API are both very thin abstractions of graphics and compute hardware with minimal CPU overhead. A layered API approach is nothing new as one similar API can often be emulated by another, while still maintaining high-performance. ANGLE for example was used for years to layer a stable OpenGL implementation over Direct X for Microsoft Windows on hardware that native OpenGL may not have been as well supported. Indeed, now, as native OpenGL driver support begins to wane, OpenGL is being implemented by third parties on top of Vulkan since native quality Vulkan implementations are now so widely available. In fact, OpenGL is now marked as deprecated on Apple platforms and is being layered on top of Metal to ensure continued application support moving forward. Vulkan too can be layered over Metal, and both APIs are similarly thin and lightweight, which means this can be done with very little additional overhead. This is exactly where we find ourselves today with MoltenVK, an implementation of Vulkan that is layered over the native Metal API to support Vulkan development on macOS, iOS, and tvOS devices.

You may have heard of MoltenVK already, and heard that it is not a fully conformant implementation of the Vulkan API. This is true, but MoltenVK does cover almost all of the Vukan 1.1 API, and ongoing work continues to bring MoltenVK closer to full conformance. The purpose of the portability initiative, and the corresponding portability extension is to provide a standard API to allow layered Vulkan implementations to indicate in what ways they are not fully conformant with the Vulkan specification. While MoltenVK on Apple devices may be the first widely available non-conformant Vulkan implementation, others are in the works, and non-conformant Vulkan implementations may soon be available and perhaps even commonplace on Windows and Linux. By using the portability extension, you can ensure your application is compliant on both fully compliant and "portable compliant" implementations.

Moreover, using the Vulkan SDK, you can make use of a device simulation layer that will allow you to validate your code against a specific Vulkan hardware device or implementation while doing your development on a fully conformant Vulkan platform such as Windows or Linux.

The big question most developers ask is what is missing from macOS when compared to Vulkan 1.1. Not a great deal really, and to be honest it's a bit of a moving target. MoltenVK is within just a handful of tests of being fully conformant with Vulkan 1.0, and work is ongoing almost daily to close this gap for versions 1.0 and 1.1. Older hardware and Metal versions also have limitations that must be worked around, but these shortcomings are disappearing as Metal matures.

It is always best on all platforms to make sure your applications query for available functionality and either work around it or fail gracefully when they are not available. This is in fact the main goal of the portability initiative, and if you are initially developing on a non-Apple platform you

can use the Vulkan SDK to validate your code against a portability profile using the Vulkan Configurator (more on this later). If you start your development on Apple hardware, then moving up to a conformant Vulkan implementation elsewhere should be relatively trivial. For the latest information about known MoltenVK issues, see the MoltenVK release notes.

The Vulkan SDK

Your first step to using Vulkan on Apple devices is to obtain the Vulkan SDK from the LunarG web site at http://vulkan.lunarg.com. The Vulkan SDK contains a prebuilt version of the MoltenVK library, the Vulkan Loader, headers, libraries, various tools, and the Vulkan Validation Layers.

In the last year alone, the Vulkan SDK for macOS has received some significant updates. MoltenVK has advanced from supporting Vulkan 1.0 to Vulkan 1.1, all SDK binary components are now *Universal Binaries* fully supporting both x86_64 and Apple Silicon hardware, and tvOS was added to Apple device support (to be fair, this was in MoltenVK already, but it was not shipping with the SDK). The Vulkan validation layers have grown from just the Khronos Validation and API Dump layers to now supporting the Device Simulation (with the portability extension!), and Synchronization2 layers as well. The Vulkan SDK as a whole has also gained a newly redesigned Vulkan Configurator application that makes using the Vulkan Validation Layers far easier than ever before.

Linking Directly to MoltenVK

There are multiple ways to include MoltenVK in your application. You can use the Vulkan Loader, or you can link directly to the MoltenVK library. We'll discuss direct linking first, as the loader approach is not available for iOS or tvOS devices. The Vulkan SDK includes MoltenVK as both a .dylib for macOS applications, and an XCFramework that can be used for both desktop and mobile/tvOS development. The .dylib approach will require distributing the library in your application bundle (in the /Frameworks folder), while the XCFramework approach statically links the MoltenVK library to your applications executable. A nice feature of the XCFrameworks is that they support multiple targets for your application automatically by including just one framework. The Vulkan headers are then found either as part of the MoltenVK prebuilt library here:

```
<SDK_install_location>/MoltenVK/include/vulkan
```

or you can use the independent set of SDK vulkan headers located in

```
<SDK_install_location>/macOS/include/vulkan
```

The headers are not bundled with the Framework, but are made available separately so they can also be included when using the Vulkan Loader, which is the recommended way to develop with Vulkan on the desktop (see below).

One reason for bypassing the Vulkan Loader is that you can make use of Metal specific extensions with Vulkan using the **VK_MVK_moltenvk** extension found in the MoltenVK specific headers located here:

```
<SDK_install_location>/MoltenVK/include/MoltenVK
```

Note: These extensions (documented in the MoltenVK repository here) will not be portable to other platforms. If you intend to use your Vulkan rendering code on non-Apple devices, these extensions should be avoided. In addition, using these Metal-specific extensions also makes it impossible to make use of the Vulkan Loader and Validation Layers.

Using The Loader

The recommended and most useful way to use Vulkan on desktop platforms or Android is to link your application to the Vulkan Loader instead of the ICD (in our case MoltenVK) directly. The Vulkan Loader finds Vulkan-capable drivers on your system and will enumerate physical devices that you can select from based on your rendering needs. On macOS, the MoltenVK library can be installed as an ICD (Installable Client Driver - at least as far as the Vulkan Loader is concerned) in a system path, or included in your applications app bundle. Most developers choose to include the MoltenVK library in their well-tested applications bundle rather than adding it as a system library that may be replaced later by another version that could contain bugs or regressions (each SDK update also updates the MoltenVK library in the system folders).

The loader provides all of the Vulkan entry points and will forward them to the Vulkan driver, which in our case is the MoltenVK dynamic library. MoltenVK will translate the calls to appropriate Metal functionality and pass them on to the selected device (you can have more than one Vulkan-capable hardware device at a time). The biggest advantage of using the Vulkan Loader over statically linking is that you can make use of one of Vulkan's premier features -- Validation Layers.

Including MoltenVK in Your Application Bundle

Installing the Vulkan SDK will place the Vulkan Loader (libvulkan.1.*version*.dylib) and the MoltenVK library (libMoltenVK.dylib) where they can be found at runtime by your development environment. This will allow you during development to take maximum advantage of the Vulkan Layers and the Vulkan Configurator for debugging and testing. End users however should not have to install the Vulkan SDK in order to run Vulkan applications! You can link directly to MoltenVK before releasing your application, but you can also ship your application with loader support by including the loader and MoltenVK in your application bundle.

The location of the pertinent files in your application bundle would be as follows:

```
YourAmazingVulkan.app
    Contents
        Frameworks
            libMoltenVK.dylib
            libvulkan.1.[version number].dylib
            libvulkan.1.dylib -> (sym link to .dylib above)
        MacOS
            YourAmazingVulkan
        Resources
            vulkan
                icd.d
                    MoltenVK_icd.json
```

In the past you could include executable code in the /Resources folder and the MoltenVK library could be placed alongside the json file. However, Apple's security policies now prohibit executable code from being placed there. Thus, the proper location for the Vulkan Loader and the MoltenVK library is in the /Frameworks folder. The MoltenVK_icd.json file points to the location of the MoltenVK library and you will need to change the default path in the MoltenVK_icd.json file thus (this is provided in the Vulkan SDK):

```
       "library_path": "../../../Frameworks/libMoltenVK.dylib",
```

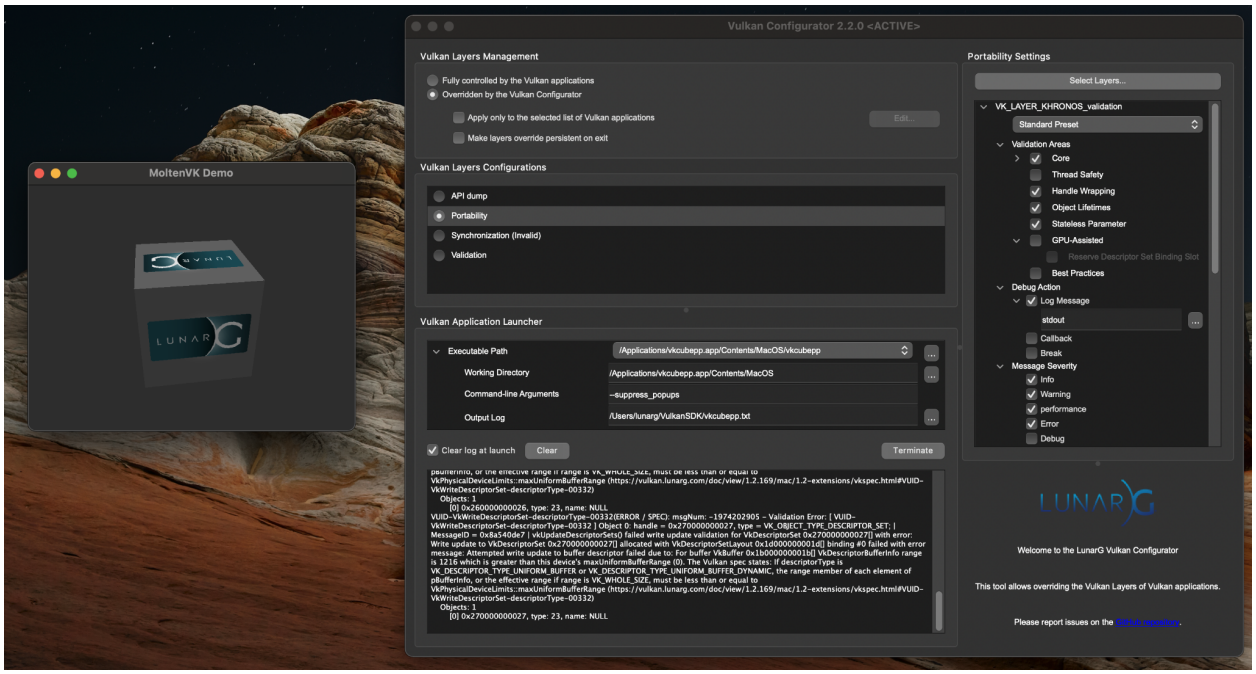
Vulkan Layers

Vulkan layers are a way to inject (or layer) a dynamic library between your Vulkan API calls and the Vulkan driver and device. Some layers will perform API validation for you, some will do logging, and others can add additional functionality to your Vulkan programs. For more information about Vulkan Layers, see the LunarG Vulkan Layers Overview and Configuration document.
Vulkan Layers are supported on the desktop by macOS, but neither the Vulkan Loader nor the Validation layers currently are available on iOS or tvOS. Thus for mobile development, you will need to link directly to the MoltenVK library as described previously.

One useful approach to Vulkan development on Apple mobile devices is to prototype your rendering code in the desktop environment, make use of the validation layers to clean up and validate your code, then deploy your shared rendering code to and do your final performance testing on your iOS or tvOS hardware directly.

The easiest way to make use of Vulkan Layers on any platform is to use the Vulkan Configurator. More information about the Vulkan Configurator can be found here. Validation layers typically send their diagnostic output to *stdout* or *stderr* and the Vulkan Configurator will automatically capture this output when your application is launched from the Vulkan

Configurator application. In addition, any layer settings can be easily inspected and edited in the Vulkan Configurator live (you will have to stop and restart your application when settings change however).



The Vulkan Configurator monitoring layer output on macOS.

Conclusion

The heir to the cross-platform graphics and compute programming throne is Vulkan. Vulkan is a fresh start after the OpenGL/OpenCL years, and is an ideal low-overhead “close to the metal” programming approach that also is highly portable across most platforms popular with today’s developers. Porting *to* Apple involves detecting the presence of the portability extensions and being aware of potentially missing functionality or other platform specific limitations. *Starting* development on Apple means you can “port up” and your applications are going to run as is on Windows, Linux, etc. more easily. Moreover, on Windows or Linux you can simulate Apple hardware capabilities using the device simulation layer and the Vulkan Configurator comes with profiles already set up for common Mac and iOS devices.

If you are already an experienced Vulkan developer, moving to macOS or Apple hardware, then there are really only a few important points to note.

1. MoltenVK provides “nearly conformant” Vulkan 1.1 for macOS, iOS, and tvOS; missing functionality is minor.

2. You cannot use the Vulkan Loader or Validation Layers directly on Apple's mobile hardware.
3. You will likely want to bundle well-tested versions of both the loader and MoltenVK libraries with your application, and not make use of a system-wide Vulkan Loader. This is counter to the norm on Windows and Linux.
4. A good strategy for mobile Vulkan development is to test and debug by using both a desktop and mobile environment simultaneously. Use the Validation Layers on the desktop to trap and correct any Vulkan usage errors or discover best practices violations.
5. Since MoltenVK is a subset, if you start your development on Apple devices, porting to fully conformant Vulkan capable platforms should be relatively painless. Starting your Vulkan project on macOS for example will help ensure maximum portability, while still leaving the option open to use additional Vulkan features to enhance your project on more conformant or capable Vulkan devices.

Vulkan. It’s everywhere. Do the Math.