



Vulkan®: The Future of Embedded Graphics

Introduction

The Khronos Group released their new 3D rendering and compute API named Vulkan on February 16th, 2016. Tools, tests and validation layers, as well as a conformant open-source Linux driver were released the same day. Vulkan is a high efficiency, royalty-free API that provides easy cross-platform access to modern GPUs. Purpose-built for 3D real-time graphics applications, Vulkan is suitable for a wide range of platforms from mobile phones to embedded applications. Built on components of AMD's Mantle API, Vulkan was initially branded as the primary successor to OpenGL®, but has been completely redesigned from the ground up with direct control over GPU acceleration, as well as support for modern GPUs and programming practices in mind.

This white paper will discuss the benefits of Vulkan, its differences as compared to OpenGL, and considerations when deciding between the two.

How Does Vulkan Compare to OpenGL?

OpenGL and Vulkan APIs each have their benefits, and it is key to consider the use case before deciding which type of API suits the application at hand. It's important to note that although OpenGL is 25+ years old, it will continue to be developed since it is for now, in some ways, easier to use than a low-level API such as Vulkan. However, GPU vendors are expected to eventually shift their focus from supporting OpenGL to supporting Vulkan.

OpenGL does not require developers to deal with memory, or with synchronizing the different blocks within a GPU, making it usable for all levels of programmers. Since OpenGL is tied to high level contexts, much of the functionality is hidden inside the driver to allow abstraction across hardware platforms. The developer tells the driver what they want done, and how they would like it done, and the driver takes care of the intricate details and checks, including the proper use of API calls, and deciding whether the state is properly prepared. The driver also provides the developer with feedback on any issues. In addition, each individual command in OpenGL contains more functionality than low-level APIs, and many of the functions can be completed easily. However, this ease of use is a trade off for less flexibility and control over GPU management as compared to a low-level API.

Vulkan is more complex to use than OpenGL and offers much more control over the GPU. The Vulkan developer must be far more involved in all the details, writing more code than is required with OpenGL. Vulkan communicates commands directly to the hardware, and while memory and error management still need to be pushed to the application, the application has the flexibility to optimize these processes, which can result in much higher performance.

Figure 1 details Khronos' summary of the differences between OpenGL and Vulkan APIs.



Ground-up Explicit API Redesign



	
Originally architected for graphics workstations with direct renderers and split memory	Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering
Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance	Explicit API – the application has direct, predictable control over the operation of the GPU
Threading model doesn't enable generation of graphics commands in parallel to command execution	Multi-core friendly with multiple command buffers that can be created in parallel
Syntax evolved over twenty years – complex API choices can obscure optimal performance path	Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance
Shader language compiler built into driver. Only GLSL supported. Have to ship shader source	SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability
Despite conformance testing developers must often handle implementation variability between vendors	Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability

Figure 1: Comparing OpenGL and Vulkan APIs ¹

Why Vulkan?

Vulkan is an object-based API with no global state. While Vulkan shares OpenGL's graphics pipeline stages and nomenclature, it has shed old layers of abstraction, resulting in simplified protocol roots, , and minimized graphical driver overhead. Vulkan is a thinner and wider API than OpenGL; although there are more API functions in Vulkan, each specific function tends to do less work on the CPU and incurs less overhead. This flexibility affords the application the possibility to setup exactly the right amount of state it needs, and in exactly the way it wants to set it, giving the application the option to optimize work and reduce CPU overhead in the process. In this way, the greatest difference between OpenGL and Vulkan is that the OpenGL implementation needs to make educated guesses about what the application intends to do - for example it needs to choose the best memory heap for a given allocation - whereas the Vulkan implementation is explicitly told by the application.

Figure 2 and Figure 3 below demonstrate the difference in CPU usage between OpenGL and Vulkan.

¹ Retrieved from https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

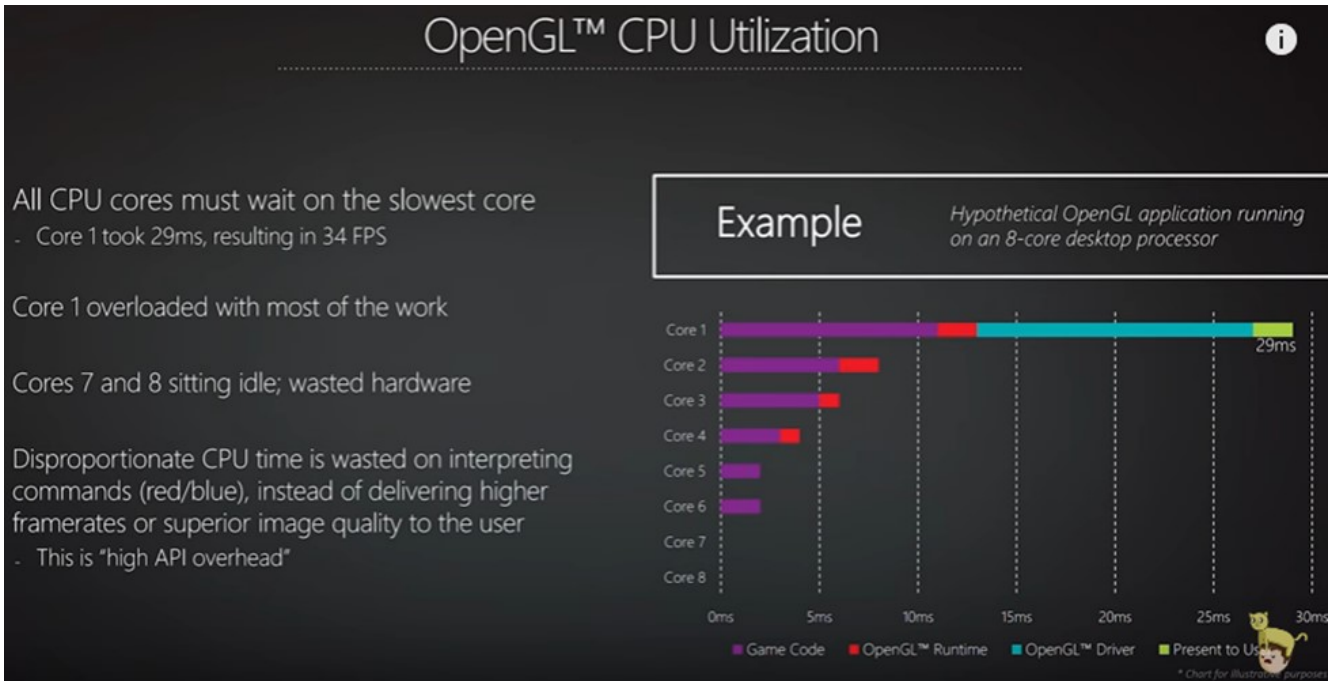


Figure 2: OpenGL CPU Utilization ²

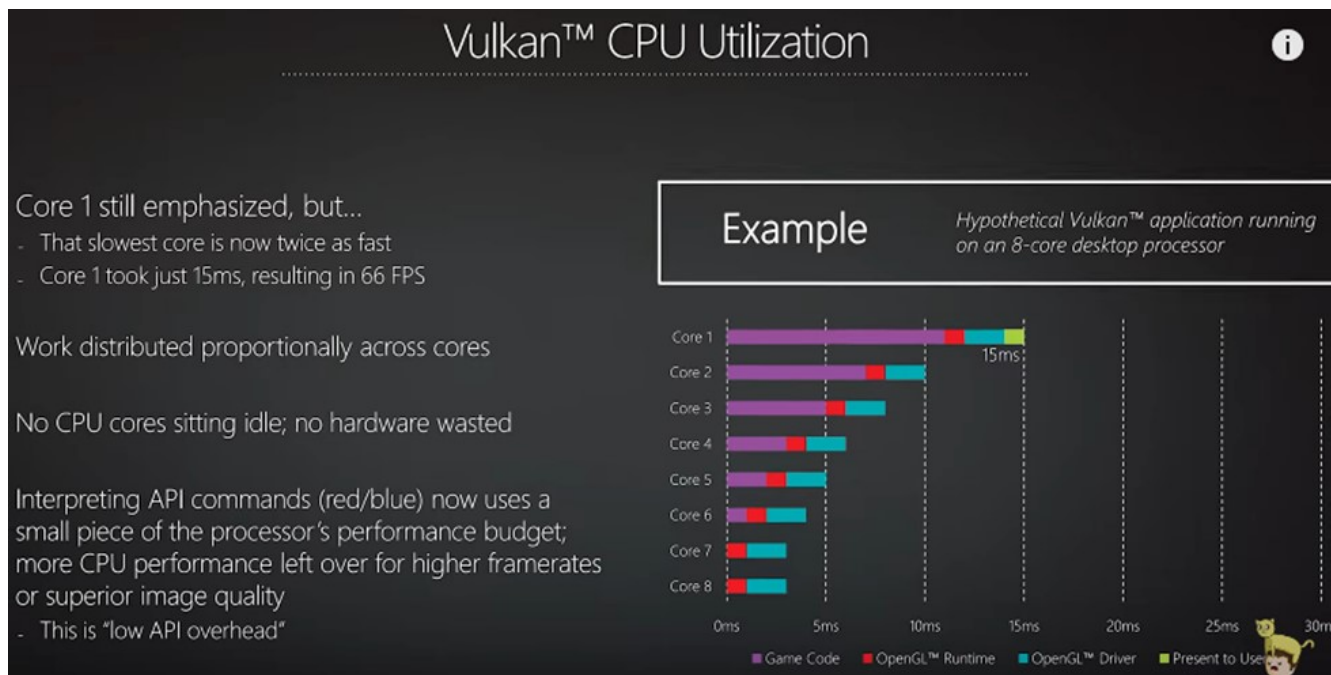


Figure 3: OpenGL Vulkan CPU Utilization ³

² Retrieved from https://www.youtube.com/watch?v=r0fgEVEgK_k (data supplied by AMD)

³ Retrieved from https://www.youtube.com/watch?v=r0fgEVEgK_k (data supplied by AMD)



Vulkan API eliminates the need for separate compute and graphics APIs. Vulkan's capabilities allow it to receive either graphics or compute commands and allocate them to the correct execution unit in the GPU. This is much more efficient and less complex than running a mixed OpenGL and OpenCL™ environment.

In contrast to OpenGL which, due to its high-level encapsulation of the graphics workload was not an API that lent itself easily to multi-threaded environments, Vulkan enables better scaling on multicore CPUs. For example, in multi-core CPUs running OpenGL, one core is typically responsible for managing time sensitive tasks. Vulkan can, however, tap previously unused hardware resources to split this workload up between many cores. Vulkan drivers do no error checking, saving significant CPU usage time; however, Vulkan has an optional parameter checking layer that may be used, or if the application has strict data checking, this layer may not be required. Parallel buffer generation ensures all cores are used and allows developers to get the maximum performance out of their hardware.

GPU hardware today is much more standardized than past hardware, and developers want visibility into the GPU. Vulkan is specifically designed for modern GPUs and allows more balanced GPU usage and more direct control over the GPU than OpenGL. With Vulkan, developers can see what the GPU is doing, as texture, memory management, formats, etc. are all developer controlled, but enough details remain hidden to maintain cross-platform compatibility.

OpenGL and Vulkan expose the GPU's programmable pipelines to the application. OpenGL uses the GLSL shader language while Vulkan uses SPIR-V. SPIR-V instructions resemble assembly instructions. Many Vulkan application developers write their shaders first in GLSL and then use a GLSL->SPIR-V converter offline, before providing the SPIR-V binaries to Vulkan online.

Should I use Vulkan or OpenGL?

The decision to use Vulkan or OpenGL APIs is highly dependent on your specific platform and configuration. The following scenarios are some examples where you may want to reconsider using Vulkan:

- Your application needs compatibility to pre-Vulkan platforms
- Your application is heavily GPU-bound
- Your application is heavily CPU bound due to non-graphics requirements
- Your application is single threaded, and this is not likely to change
- Your application can target middleware and avoid direct 3D graphics

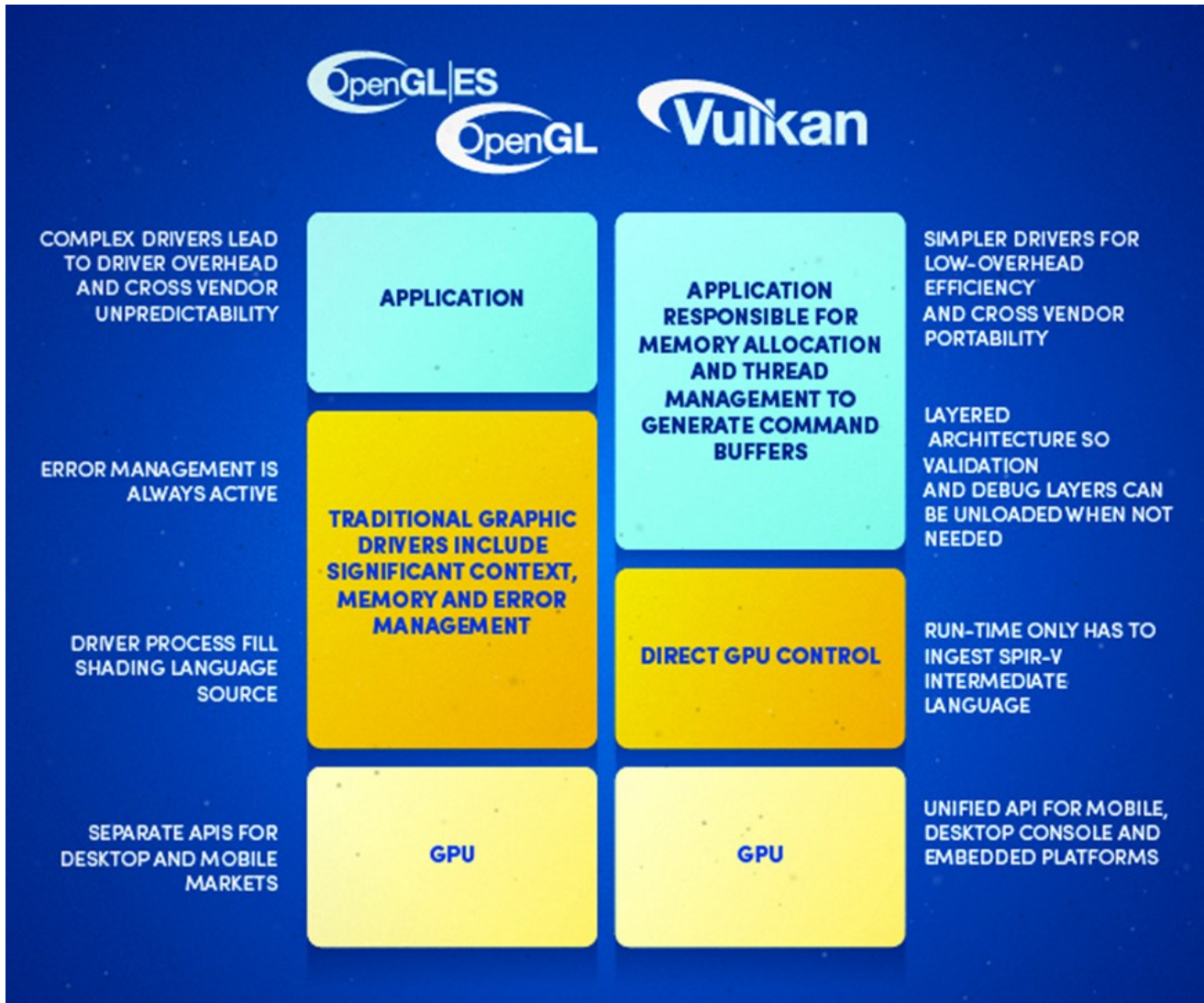


Figure 4: Comparison of OpenGL and Vulkan APIs⁴

⁴ Taken from <https://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api> Oct 30, 2017



VkCore™ SC Safety Critical API

To bring Vulkan's state of the art capabilities to embedded markets, CoreAVI has developed a safety critical version of the Vulkan API called VkCore SC. VkCore SC is designed and developed from the ground up for high performance and flexibility and offers the option for RTCA DO-178C/EUROCAE ED-12C certification up to DAL A. Legacy applications are also supported through OpenGL SC 1.0.1 and Open GL SC 2.0 libraries called VkCoreGL SC 1 and VkCoreGL SC 2 running on top of Vulkan, allowing legacy OpenGL applications to take advantage of the advanced capabilities of Vulkan while transitioning from OpenGL to Vulkan. This allows for performance improvements and the addition of differentiated features to existing applications.

VkCore SC is available with the DO-178C/ED-12C safety certification packages required for avionics applications, as well as an ISO 26262 Accredited Safety Assessment Certificate for automotive platforms. To learn more, read our [VkCore SC datasheet](#) or contact Sales@coreavi.com.

Author



Mary Beth Barrans

Director of Marketing

Mary Beth Barrans joined CoreAVI in 2017. As the Director of Marketing, she is responsible for the product positioning and customer-focused power messaging for CoreAVI's safety certifiable graphics and hardware IP product lines, as well as strategic partnerships and events. She previously worked for Curtiss-Wright Defense Solutions as a Senior Marketing Product Specialist. Mary Beth has a Bachelor of Social Sciences, a Bachelor of Education, a Masters of Arts, and a Technical Writing designation.