Efficiency and Development Effort of OpenCL Interoperability in Vulkan and OpenGL Environments: A Comparative Case Study

Piotr Plebański[a], Anna Kelm[b] and Marcin Hajder[c]
Institute of Computer Science, Cardinal Stefan Wyszynski University, Warsaw, Poland

Keywords: GPGPU, API, OpenCL, OpenGL, Vulkan, Interoperability.

Abstract: The increasing demand for high-performance computing has led to the exploration of utilizing General-Purpose Graphics Processing Units (GPGPUs) for non-graphical tasks. In this paper, we present a comparative case study of OpenCL interoperability when paired with two widely used graphics APIs: OpenGL and Vulkan. By implementing an ocean wave simulation benchmark – where OpenCL handles compute-intensive tasks and the graphics API manages real-time visualization – we analyze the impact of API selection on both execution performance and development effort. Our results indicate that Vulkan's low-level control and multi-threaded design deliver marginal performance improvements under minimal rendering loads; however, its increased code verbosity and complex synchronization mechanisms lead to a substantially higher development effort. In contrast, OpenGL, with its more straightforward integration and broad compatibility, provides a practical alternative for compute-first applications. The insights from this case study offer guidance for developers navigating the trade-offs between raw performance and maintainability in GPU-accelerated environments.

1 INTRODUCTION

Since the advent of programmable graphics processing units (GPUs, also known as graphic cards), there has always been interest in utilizing them for non-graphical computational tasks (Harris et al., 2002).

Initially, this had to be done by hacking graphics application programming interfaces (APIs), for example, pretending to render to a texture while in fact using the fragment shader to perform fluid calculations (Harris and Gems, 2004).

This has been addressed by the release of the Compute Unified Device Architecture, CUDA (NVIDIA Corporation, 2025b), a dedicated compute API, which allowed programmers to view their general-purpose GPUs (GPGPUs) as massively parallel processors rather than means for rendering graphics.

Open Computing Language, OpenCL (Khronos Group, 2023a) soon followed as a free and open alternative to CUDA, and is now supported by all major GPGPU vendors.

While direct use of either graphics or compute APIs is considered low-level today, due to the wide availability of rendering engines and software for performing GPU-accelerated calculations (e.g.: PyTorch, RAPIDS, MATLAB), a bespoke solution may still be preferable for performance-sensitive applications.

One of the common use cases is a combination of GPGPU-powered simulation with a real-time visualization of the generated data, e.g., for fluid dynamics (Gunadi and Yugopuspito, 2018; Harris and Gems, 2004). We investigate the case where a compute API is utilized for generating the data, combined with a graphics API for rendering the results in 3D. Specifically, we chose OpenCL as the compute API, and decided to compare the application performance when combining it with either Vulkan or OpenGL as graphics API.

Although modern graphics APIs such as DirectX, Metal, or Vulkan support compute pipelines, their compute shaders are not compatible with each other (Galvan, 2022), so choosing the visualization API to implement the simulation would effectively limit the project to that one solution, and by extension, to the platforms that support it. Separating the compute and graphics API grants the application better portability, as well as flexibility in the choice of visualization. Finally, we assume that the application

[a] https://orcid.org/0009-0000-5973-5720
[b] https://orcid.org/0000-0002-4975-1994
[c] https://orcid.org/0009-0008-2092-8013

Plebański, P., Kelm, A. and Hajder, M.
Efficiency and Development Effort of OpenCL Interoperability in Vulkan and OpenGL Environments: A Comparative Case Study.
DOI: 10.5220/0013529000003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 111-119
ISBN: 978-989-758-757-3; ISSN: 2184-2833

would be primarily focused on generating simulation results, so a compute API is the superior choice for structuring the program around its primary purpose.

We chose OpenCL instead of CUDA (NVIDIA specific) or HIP (AMD specific) because of the wide support of the API across hardware vendors (Khronos Group, 2023a; Acosta et al., 2018). Similarly, we only investigate the interoperability with Vulkan and OpenGL due to their wide support across platforms (Khronos Group, 2025a). The APIs we focus on are all open standards maintained by The Khronos Group.

When considering which Graphics API to choose for an application, performance is assumed to be of high importance - otherwise, the application would likely be based on a rendering engine or a ready-made software that fits the task.

For games and graphically-intensive applications, using the Vulkan API is known to consistently lead to better performance than other graphics APIs (Dyńdo, 2017, and references therein). The complexity cost has been widely proven to be worth it in those applications.

We investigate whether this holds true for applications where the compute component of the workload is significant and is performed independently of the graphics API. Obviously, the less time the application spends rendering graphics with respect to the time spent on other calculations, the less potential gains can be expected from the use of a "faster" graphics API. However, even at this point our preliminary explorations revealed several nuances, such as Vulkan's capacity for yielding very high frames per second (FPS) numbers at very light workloads, which may be significant for some low-latency applications. We examine various aspects of graphics APIs interoperability with OpenCL and discuss the performance implications that follow.

2 GRAPHICS APIs: OpenGL AND VULKAN COMPARED

2.1 OpenGL

OpenGL is a well established graphics API, with mature drivers across many platforms, and is well supported by all GPU vendors. It provides a high level of abstraction, leading to concise and clearly structured programs (Peddie, 2022).

OpenGL traces its origins to Silicon Graphics, Incorporated (SGI) *IRIS GL* API from the 1980s and adopted its modern name with its public release in 1992. Since then it has been maintained as an open standard by The Khronos Group, and in 2004 with version 2.0 received a major upgrade to better suit the needs of modern GPUs.

Up to 2023, the OpenGL standard has been regularly updated to support feature extensions that allow its users to leverage some of the latest advances in GPGPU technology, such as mesh shading (Kubisch et al., 2019).

Support of some modern GPU features is missing, perhaps most notably, there is no ray tracing support planned for OpenGL (Piers, 2019).

2.2 Vulkan

Vulkan is a relatively new graphics API initially unveiled in 2015. Its main focus was enabling better performance through its three main features:

- low-level, detailed API calls, allowing for an unprecedented amount of control over the GPU (Khronos Group, 2025d),
- modern architecture, allowing for multi-threaded operation and diverse GPU-CPU synchronization methods (Khronos Group, 2025b), and
- low driver overhead, allowing for efficient communication with the GPU (Kapoulkine, 2025).

While these features allow the programmer to highly optimize the rendering process, the added benefit is enabling GPU drivers to better tailor the low-level execution details to the intended program's purpose, thanks to the added detail in every API call and its context (Khronos Group, 2025c).

2.2.1 Vulkan *vs* OpenGL

Vulkan's advantages come at a cost of significantly higher program complexity with respect to OpenGL. For most tasks, that can be divided into four areas of increasing difficulty:

- **code size**, the sheer amount of code that has to be understood,
- **code complexity**, in terms of more variables, dependencies, parameters, and size of manipulated objects,
- **execution complexity**, stemming from the complex synchronization mechanisms governing parallel program flow, and
- **program state complexity**, in terms of more API objects that have to be tracked during program lifetime.

Comparatively, OpenGL offers a fairly linear execution model, while in Vulkan running the rendering on

the GPU in parallel with the code on the CPU is effectively mandatory. Linear program flow can be forced in Vulkan by explicitly waiting for the GPU to become ”idle” - the `vkDeviceWaitIdle` function – but is considered a bad practice and leads to very poor performance.

3 OpenCL INTEROPERABILITY WITH GRAPHICS APIs

OpenCL is an API for performing calculations in heterogeneous environments. It can be run directly on the CPU as well as on the GPGPU. It is an open standard maintained by The Khronos Group (Khronos Group, 2023a).

The typical workflow for running OpenCL is first assigning memory buffers for input and output data, then running an OpenCL kernel that reads the inputs, performs calculations in a super-parallel manner, and writes the results to the output buffer. In a real-life application this would likely involve multiple kernels and in/out buffers, with execution staged and synchronized using events to optimize performance, but the end result remains the same. Once the calculations finish, the result is available to be read from the output buffer by the application running on the CPU.

Graphics APIs operate in a similar way, with assets like 3D geometry and textures being loaded into GPU memory by the CPU-side program and then utilized by shaders running on the GPU to efficiently generate images.

Data transfer between CPU and GPU memory is a relatively slow process, limited by the performance of the PCI Express (PCIe) bus, and subject to latency incurred from memory management operations, as well as the overhead incurred from performing bus transfers rather than accessing memory directly. Submitting the data to PCIe also consumes CPU execution time.

Below are the *theoretical* throughputs at which each component operates:

- PCIe v5 16x connector has a theoretical bandwidth of 64 GB/s,
- CPU-side DDR4 Memory running in quad-channel configuration at 2000 MHz goes up to 128 GB/s (4 * 64-bit memory bus),
- CPU-side DDR5 Memory running in dual-channel configuration at 4600 MHz goes up to 147.2 GB/s (2 * 64-bit memory bus),
- GPU-side GDDR6X on a NVIDIA RTX 3090 goes up to 936 GB/s (384-bit memory bus) (Micron Technology, Inc., 2022), and
- on a NVIDIA RTX 5090, its GDDR7 memory goes up to 1792 GB/s (512-bit memory bus) (NVIDIA Corporation, 2025a).

The above numbers demonstrate that copying data from system memory to the GPU is slow relative to GPU's internal transfer speeds. This is where OpenCL's interoperability extensions become important – both OpenGL and Vulkan support extensions that allow OpenCL kernels and graphics API shaders to share GPU memory, which is depicted in Figure 1.

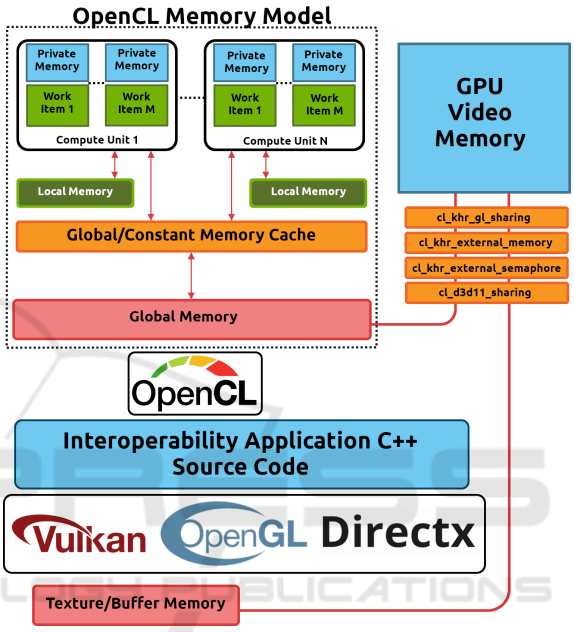

Figure 1: Interoperability relationship between OpenCL Memory Model and Video Memory shared with rendering interfaces, see (Raja et al., 2012; Khronos Group, 2023b).

For OpenGL-OpenCL interoperability, the `cl_khr_gl_sharing` extension provides the functionality. When the extension is available, it allows OpenCL to efficiently share OpenGL resources such as textures and buffer objects without the overhead of transporting GPU data back and forth to the host memory. This setup enables real-time data sharing, where computational results can be rendered almost directly from the shared memory. If the extension is not available, OpenCL calculation results must make the round trip back to the host system and be copied into a dedicated OpenGL buffer.

For Vulkan-OpenCL interoperability the `cl_khr_external_memory` is available. However, it should be used in concert with the `cl_khr_external_semaphore` extension to optimize data sharing between the two APIs.

The `cl_khr_external_memory` extension enables memory sharing between Vulkan and OpenCL

by directly accessing Vulkan's memory objects and allows direct access to the memory associated with OpenCL objects to Vulkan shaders.

The `cl_khr_external_semaphore` extension manages synchronization, ensuring that compute and graphics tasks are well-coordinated, preventing conflicts during read/write operations on shared resources. This highlights the difference between the approaches taken by OpenGL and Vulkan, where the former takes care of the synchronization, while the latter allows the programmer to specify exactly at what point during execution should the resources be available to the other API's programs.

It should be noted that using APIs, even low-level ones such as Vulkan, it is merely communicated to the device and its driver what the expected data flow should be. It is up to the GPGPU to orchestrate the shader execution and data transfers as per the API contract. The programmer shall not make any assumptions pertaining to execution details not guaranteed by the Vulkan specification.

4 IMPLEMENTATION AND METHODOLOGY

In order to measure the impact of graphics API choice on application performance and complexity, we implemented two conceptually equivalent benchmarks, one using OpenGL for visualization, the other based on the Vulkan API.

Our benchmark concept, inspired by (Flügge, 2017), utilizes simulation outputs to drive the vertex shader as a height map (Dempski, 2002). The simulation is based on the Cooley-Tukey fast Fourier transform (FFT) algorithm, which optimizes discrete Fourier transform (DFT) computations. This algorithm decomposes complex wave equations into simpler components, transforming them into a grid format to enhance computational efficiency. The innovation in (Flügge, 2017) lies in adapting the DFT technique to GPU architecture, significantly reducing the computation time required for ocean wave simulations.

4.1 Simulation Details

At a high level both OpenGL and Vulkan implementations of our benchmark operate in the same manner. Both the OpenCL kernels and the code driving the computation have intentionally been kept as similar as possible in both variants.

During the initialization phase, the contexts for both the compute and graphics APIs are created. Context creation involves enumerating available devices, verifying they meet minimum requirements, and ensuring necessary interoperability extensions are supported.

Once the APIs contexts are ready, OpenCL preparation is nearly the same for both benchmark variants. The compute kernels are compiled and setup for operation, memory buffers are allocated, configured for interoperability and seeded with initial data. The first round of kernel execution further processes this startup data to produce optimized wave textures for running the simulation.

The two notable differences are the extensions used for interoperability between OpenCL and the graphics API, and the kernel file format. For OpenGL-OpenCL interoperability, the shaders are supplied as GLSL (OpenGL Shading Language) programs which are compiled during program runtime. When working with Vulkan, the shaders must be supplied as SPIR-V (Standard Portable Intermediate Representation) binaries, which have to be manually prebuilt – from similar GLSL files – before starting the program. The GLSL source remains mostly the same for both versions.

Preparing the graphics context for rendering is somewhat different between the two benchmarks. However, similarities can still be outlined at a high level. For both graphics APIs we start by setting up various presentation details, such as creating a window for displaying the results, and allocating image buffers for storing rendering output. We then prepare vertex buffers for storing the mesh that will be rendered, and populate them with a flat grid that will later be transformed in the vertex shader, based on the OpenCL simulation output.

The presentation part of the program remains similar for both variants. After the contexts have been initialized, the program enters the main rendering loop. First – marked in Figure 2 with green blocks – the OpenCL simulation is run to generate values for the next frame, then the results are made available to the graphics API context, and used in submitting commands for rendering the next frame. API-specific operations follow, in both cases resulting in rendering the scene, shown in Figure 2 with a red block.

During rendering, the vertex shader is responsible for transforming the contents of the vertex buffer in two ways.

First, it takes the flat grid we created during init and transforms it into the *ocean surface* shape based on the output of the last OpenCL calculation. Each pixel in the resulting *ocean texture* corresponds to an elevation above the flat plane the grid starts at. For each vertex in the grid, the vertex shader collects the

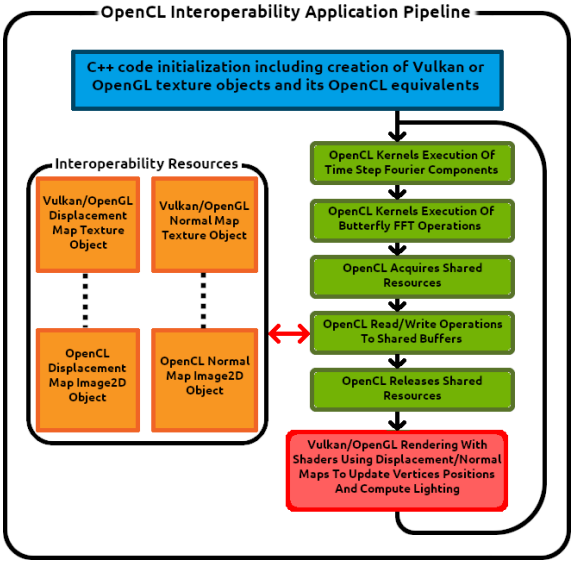

Figure 2: The illustrated diagram depicts the processing pipeline for the OpenCL interoperability applications being compared. In both Vulkan and OpenGL implementation cases, the primary loop of the application can be approximated using identical steps.

corresponding texture value, and displaces the vertices ”up” or ”down” by the adequate amount.

Next, the vertex shader performs the model-view-projection (MVP) matrix transformations in order to align the rendered mesh with the current camera position and adjust perspective.

Then the fragment shader calculates the lighting and Fresnel factor for each pixel where the ocean surface is visible. The latter calculation uses a normal map, which is also generated by the OpenCL kernels. Other than this we intentionally keep the lighting calculations simple, as visual fidelity is not the focus of this work.

4.2 Benchmarks

We implemented both benchmarks in C++, utilizing the CMake build system available in the OpenCL-SDK repository (Group, 2023).

The OpenGL benchmark is modeled after the OpenCL interoperability sample available in the OpenCL-SDK repository. Its base class extends the `cl::sdk::InteropWindow` class, which provides the basic scaffolding for initializing OpenGL and OpenCL contexts alongside each other. This benchmark has a mostly linear flow, with some OpenCL calculations being performed in parallel to program execution on the host, but the execution reconverges as the program waits for the simulation results to become available. The data is then made available to the OpenGL context and commands for rendering the

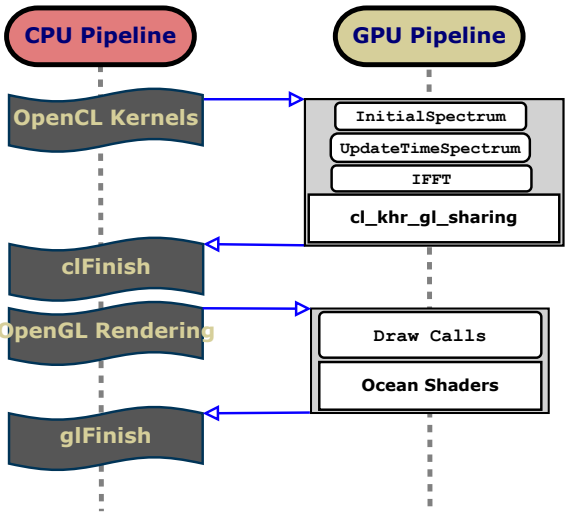

Figure 3: OpenGL benchmark render loop outline. Blue lines with arrows indicate the host blocking execution until OpenCL results are available.

next frame are submitted. Figure 3 shows the render loop execution flow is linear, with various steps executed without overlap.

The Vulkan benchmark is based on the code presented at the end of the Vulkan Tutorial (Overvoorde, 2023). Functionally, it differs in two significant ways from the OpenGL version:

- In addition to using `cl_khr_external_memory`, the Vulkan equivalent of the OpenGL-OpenCL interoperability extension, the benchmark also relies on the `cl_khr_external_semaphore` extension to fully synchronize resource sharing.
- While the program is still bound by waiting for OpenCL calculation results, the rendering work submitted via the Vulkan API runs in parallel to the code executed on the host system. This effectively allows the program to save time by running OpenCL and Rendering jobs in parallel, as the rendering of the previous frame usually completes while we wait for the next frame’s simulation results.

Vulkan API variant of our benchmark is significantly more verbose, its code size over six times larger than the OpenGL implementation. As shown in Figure 4, the program flow is also more complex, with multiple points where synchronization is necessary to ensure resources are free before continuing operation.

Both variants have been instrumented with multiple timestamps being recorded during execution.

We use the standard C++ `chrono` library (cppreference.com, 2025) for calculating time, with the `std::chrono::steady_clock` clock variant being used to record high-precision measurements.

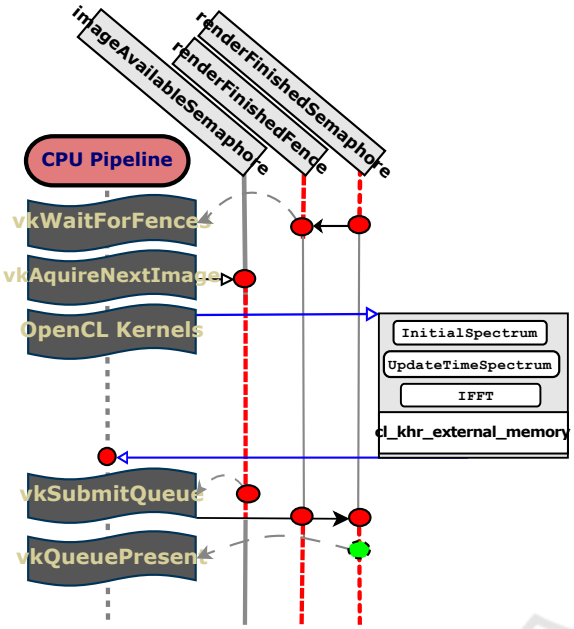

Figure 4: Vulkan benchmark render loop outline. Red circles and dashed lines indicate synchronization. Blue lines with arrows indicate the host blocking execution until OpenCL results are available. Gray dashed lines point to the function waiting for synchronization.

We measured the performance of both benchmark variants on three NVIDIA GPUs and two operating systems (Windows and Linux), and repeated the measurements at three levels of rendering workload. In order to increase the amount of rendering work, we increased the number of rendered ocean meshes, rendering the same mesh once, 64 times, and 256 times.

The GPUs, computers and drivers they operated on are:

- NVIDIA Geforce RTX3090 (24 GB VRAM), AMD Ryzen Threadripper 3970X 32-Core Processor, 128 GB DDR4@4000 RAM, Gigabyte TRX40 AORUS XTREME motherboard, drivers: v550.144.03 (Linux), v560.94 (Windows),
- NVIDIA Geforce RTX2070 Super (8 GB VRAM), AMD Ryzen 9 3900 12-Core Processor, 64 GB DDR4@3600 RAM, MSI X570-A PRO motherboard, drivers: v545.23.08 (Linux), v560.94 (Windows), and
- NVIDIA RTX 4070 (12 GB VRAM), AMD Ryzen 7 5700x 8-Core Processor, 32 GB DDR4@3200 RAM, Asus Prime B550-Plus motherboard, drivers: v565.77 (Linux), v566.36 (Windows).

Using different computers for each card prevented direct comparisons of each card efficiencies, but this is unlikely to significantly affect the comparison of the two GPGPU APIs, the study's primary focus.

AMD Vulkan drivers do not support the `cl_khr_external_memory` extension, thus making interoperability performance the main bottleneck in the application. For this reason we omitted AMD cards from our benchmarks.

Before running each benchmark, we set GPUs fans to run at their maximum speed to increase thermal stability, and ensured no other processes were active that might skew the results. The programs were configured to capture 10,000 frames, generated as fast as the system was capable, without limiting FPS to the display capability. The scene was rendered at 1600x800 resolution.

For analysis, execution times for frame numbers 3,000 to 8,000 were taken, which eliminated measurement instability of the initial fast temperature increase of the GPUs. The frame times in the results were recorded with 100 ns precision.

5 RESULTS AND DISCUSSION

Our analysis consists of two major directions. First, we assess the development cost of both graphics APIs, then we focus on performance measurements to evaluate possible benefits related to each of implementations.

5.1 Development Cost

Both applications, utilizing either Vulkan or OpenGL, follow identical functional flow paths. This similarity limits the number of software complexity measures that can provide a meaningful comparison. For instance, it excludes McCabe's cyclomatic complexity (McCabe, 1976) from consideration. Metrics that focus on the properties of the source code alone appear to be more suitable for this specific scenario.

The simplest available metric, NLOC (number of lines of code), quickly reveals the difference between the two approaches. After isolating only the code pertinent to the use of a particular graphics API, OpenGL code fits in just 261 lines of code, while the distilled Vulkan API implementation spans 1,711 lines, over 6.5 times more.

Analyzing the program flow also reveals a significantly increased complexity for Vulkan. At a glance, Figures 3 and 4 demonstrate that the Vulkan API is substantially more complex, featuring multiple synchronization points.

Taken together, these factors make it significantly more challenging – and thus more costly – to work on programs implemented using the Vulkan API. Not only are they harder to reason about, but the software

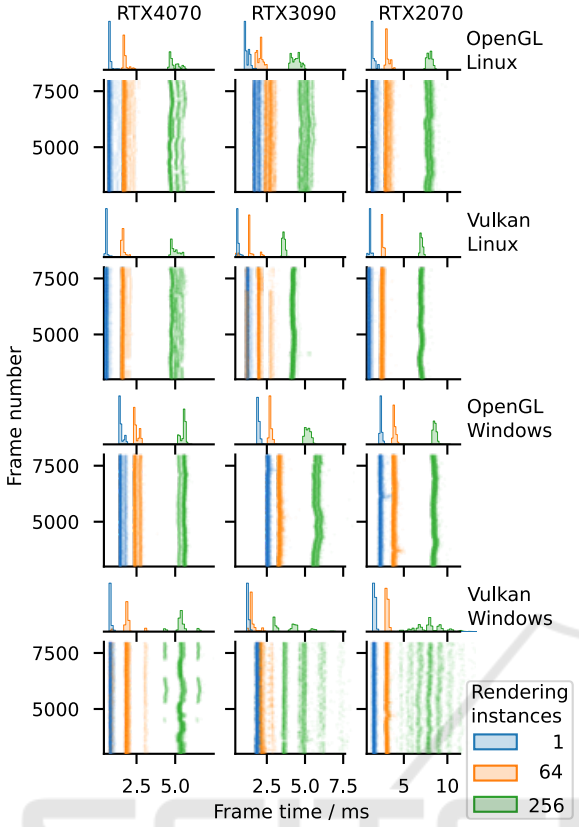

Figure 5: Time traces and histograms for all tested combinations of hardware, OS, and rendering workload.

developer must also gain an understanding of a substantially larger code base.

5.2 Performance Analysis

We performed our measurements on three GPUs, on Windows and Linux, for each combination running the benchmark with three increasingly larger rendering workloads. Figure 5 shows time traces and histograms of the measured frame times.

In most of the time traces, there is a visible spread of the measured values into roughly discrete intervals. Undoubtedly, the strongest spread can be observed for the *Vulkan Windows* frame times, clearly differing from the *Vulkan Linux*, which suggest it might be associated with the differences in Windows and Linux driver implementation for NVIDIA GPUs.

The effects of GPU temperature fluctuations that present as continuous small deviations in the traces, visible especially for 256 rendering instances workload, had negligible effect on the calculated mean frame times as the temperature remained stable during the measurement except for the few hundreds initial warm-up frames not shown in the plot.

The graph shows no obvious increase of the performance of Vulkan over OpenGL. The *Vulkan Windows* frame times are spread into roughly discrete intervals. However, both the histograms on top of the plots, and averaging the samples (Figure 6) reveal the performance to be similar to that seen on other graphs.

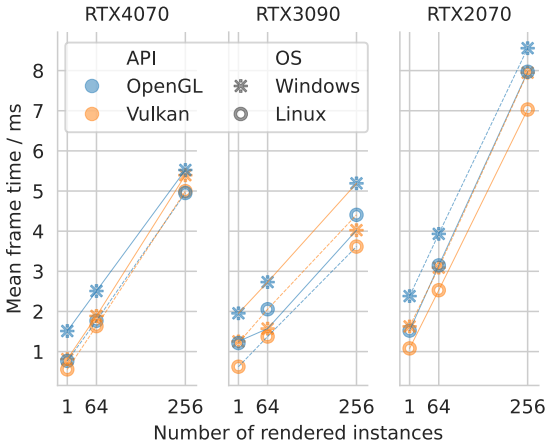

Figure 6: Mean performance of Vulkan vs OpenGL as rendering workload increases.

The observed Vulkan execution times are consistently lower across all combinations of hardware, OS and workloads. The difference appears to be a fixed amount which we attribute to the overall gain coming from lower API overhead and rendering in parallel with program execution.

The total rendering time increases linearly with increased rendering workload with frame times being consistently better on Linux than on Windows, especially for the Vulkan API.

For small rendering workloads (1 instance) Vulkan can yield significantly more FPS than OpenGL, starting at much lower frame times. For larger workloads, the performance benefit of using Vulkan becomes relatively smaller, not increasing significantly from the observed fixed amount, while frame times increase.

5.3 Vulkan *vs* OpenGL Performance

The observed performance benefits from using Vulkan API over OpenGL in a compute-first application fail to provide us with any clear answers, confirming only that rendering API choice is not obvious.

Even though it may seem that for performance-sensitive applications Vulkan should be the first choice, the observed speedups were barely significant when the rendering workload was increased. Counter-intuitively, it seems that Vulkan might be a good fit for relatively simple visualizations, where it allows the rendering part of the application to keep up with

the compute part running in OpenCL, keeping visualization latency very low.

As the rendering workload increased, the relative speed gain from using Vulkan over OpenGL decreased rapidly, making the extra effort required for using the complex API a questionable choice.

It is worth noting that the case we investigated is not indicative of all potential use-cases for OpenCL+visualization applications. For example, an application with a highly complex visualization rendering pipeline might exhibit higher gains from a Vulkan implementation. On the other hand, such highly complex cases might benefit from using a 3D rendering engine instead of interacting with a graphics API directly.

An important consideration is driver support for needed extensions. We found it surprising that AMD Vulkan drivers do not support the `cl_khr_external_memory` extension.

If very high FPS (over 2,000) is not required and the application does not need ray tracing or other unsupported functionalities, OpenGL is definitely a safer choice, and is likely to work on a wide variety of hardware and operating systems in the foreseeable future. While implementing the visualization in Vulkan might bring marginal performance gains, those would likely not be worth the extra development and maintenance effort that choice would incur.

5.4 Comparison with Previous Studies

Comparing our results and findings to (Gunadi and Yugopuspito, 2018) and (Lujan et al., 2019), we see they fall somewhere between them. (Lujan et al., 2019) also finds that Vulkan is capable of much higher FPS at very low loads, far exceeding results observed in our benchmarks. We attribute this to our use of OpenCL, which added a small delay to each generated frame, which became the dominant component of each cycle as rendering times went down. In (Gunadi and Yugopuspito, 2018) OpenGL is noticeably faster for some simulation parameters, which is something we did not observe in our experiments. While we noted some marginal cases where OpenGL frame times were faster than those on Vulkan, they were not significant in our benchmark results. Apart from that, the results from (Gunadi and Yugopuspito, 2018) are in agreement with ours, in that Vulkan can be consistently faster than OpenGL, but the difference, while noticeable, is not very large.

6 CONCLUSIONS

The rise of general-purpose GPU computing has transformed scientific computing and real-time visualization applications, making the choice of programming interfaces increasingly critical for developers. While modern GPUs can efficiently handle both computational tasks and graphics rendering, the diversity of available APIs – from vendor-specific to open standards – presents complex trade-offs in terms of performance, portability, and development complexity. The growing demand for applications that combine intensive computations with real-time visualization has made it particularly important to understand how different compute and graphics APIs interact and perform together.

This research investigated the performance implications of combining OpenCL (a compute API) with different graphics APIs (Vulkan and OpenGL) in applications that require both intensive computational tasks and real-time visualization. The analysis reveals that using Vulkan significantly increases development complexity and cost, with Vulkan implementations requiring over six times the lines of code compared to OpenGL and involving more intricate synchronization. Performance measurements show that Vulkan offers marginal frame time improvements primarily at low rendering workloads, but these benefits diminish as the workload increases, making the complexity unjustifiable for larger tasks. Furthermore, Vulkan's advantages are inconsistent across different operating systems and hardware, partly due to limited driver support for certain extensions and general immaturity of still developing drivers. Consequently, for most applications where extremely high frame rates or advanced rendering features are not essential, OpenGL emerges as the more practical and cost-effective choice.

While our study provides insights into selected efficiency characteristics and development effort comparisons between OpenGL and Vulkan, a comprehensive comparison of these two APIs requires further research.

We did not investigate the possible extent of optimization that is available for Vulkan and OpenGL applications. We study a fairly straightforward application and it may be worth investigating whether our findings hold true for more optimized programs, both in terms of potential performance gains, and the effort required to achieve them.

We also used a fairly straightforward visualization in our benchmarks. A more complex example may provide more opportunities for optimization, as well as present interesting performance bottlenecks related

to OpenCL-graphics API interoperation.

Finally, the case we presented does not involve significant data throughputs in the application, an area where Vulkan's multi-threaded nature, flexible memory management options, and elaborate synchronization mechanisms might allow it to display significant performance gains, or rapidly raise code complexity. This case would also present an interesting opportunity to compare Vulkan's API-provided mechanisms to more conventional means which might be necessary to achieve similar results in OpenGL.

ACKNOWLEDGEMENTS

This research was funded by the Institute of Computer Science of Cardinal Stefan Wyszyński University in Warsaw, Poland. We want to express our sincere gratitude to the University that has supported our research efforts.

We sincerely thank Robert Kłopotek, PhD, Eng. for valuable remarks on the manuscript.

REFERENCES

Acosta, A., Merino, C., and Totz, J. (2018). Analysis of opencl support for mobile gpus on android. In *Proceedings of the international workshop on openCL*, pages 1–6.

cppreference.com (2025). Date and time utilities. [Aaccessed: 2025-02-07].

Dempski, K. (2002). *Real-Time Rendering Tricks and Techniques in DirectX*. The Premier Press game development series. Premier Press, 1 edition.

Dyńdo, D. (2017). Vulkan i opengl – porównanie wydajności na przykładach. Master's thesis, Uniwersytet Wrocławski, Wydział Matematyki i Informatyki, Instytut Informatyki, Wrocław.

Flügge, F.-J. (2017). Realtime GPGPU FFT ocean water simulation. Master's thesis, Hamburg University of Technology, Institute of Embedded Systems, Hamburg, Germany. Research Project Thesis.

Galvan, A. (2022). A review of shader languages. https://alain.xyz/blog/a-review-of-shader-languages. Accessed: 2025-02-06.

Group, K. (2023). Opencl-sdk. https://github.com/KhronosGroup/OpenCL-SDK. Accessed: 2025-02-07.

Gunadi, S. I. and Yugopuspito, P. (2018). Real-time gpu-based sph fluid simulation using vulkan and opengl compute shaders. In *2018 4th International Conference on Science and Technology (ICST)*, pages 1–6. IEEE.

Harris, M. J., Coombe, G., Scheuermann, T., and Lastra, A. (2002). Physically-based visual simulation on graphics hardware. In *Graphics Hardware*, volume 2002, pages 1–10.

Harris, M. J. and Gems, G. (2004). Chapter 38: Fast fluid dynamics simulation on the gpu.

Kapoulkine, A. (2025). Reducing vulkan api call overhead. https://gpuopen.com/learn/reducing-vulkan-api-call-overhead/. Accessed: 2025-02-05.

Khronos Group (2023a). Opencl. Accessed: 2025-02-05.

Khronos Group (2023b). Opencl programming model. GitHub repository. Chapter in *OpenCL Guide*, https://github.com/KhronosGroup/OpenCL-Guide/blob/main/chapters/opencl_programming_model.md, accessed on 2025-02-06.

Khronos Group (2025a). Conformant products. https://www.khronos.org/conformance/adopters/conformant-products#opengl. Accessed: 2025-02-06.

Khronos Group (2025b). Multithreading render passes sample - vulkan samples. https://docs.vulkan.org/samples/latest/samples/performance/multithreading_render_passes/README.html. Accessed: 2025-02-05.

Khronos Group (2025c). Vulkan essentials - vulkan samples. https://docs.vulkan.org/samples/latest/samples/vulkan_basics.html. Accessed: 2025-02-05.

Khronos Group (2025d). Vulkan specification: Introduction chapter. https://docs.vulkan.org/spec/latest/chapters/introduction.html. Accessed: 2025-02-05.

Kubisch, C., Brown, P., Uralsky, Y., Smith, T., and Knowles, P. (2019). *GL_NV_mesh_shader*. NVIDIA.

Lujan, M., Baum, M., Chen, D., and Zong, Z. (2019). Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 777–781. IEEE.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Micron Technology, Inc. (2022). GDDR6X Infographic. https://www.micron.com/content/dam/micron/global/public/infographics/gddr6x-infographic.pdf. Accessed: 2025-02-05.

NVIDIA Corporation (2025a). Geforce rtx 5090 graphics card. Accessed: 2025-02-05.

NVIDIA Corporation (2025b). Nvidia cuda zone. https://developer.nvidia.com/cuda-zone. Accessed: 2025-02-05.

Overvoorde, A. (2023). Vulkan tutorial. https://vulkan-tutorial.com/. Accessed: 2025-02-07.

Peddie, J. (2022). *The History of the GPU-Eras and Environment*. Springer.

Piers, D. (2019). Opengl / opengl es update. https://youtu.be/1fU4w2ZGxH4?si=c6CnJdIwFO8V5tcE&t=8930. Presentation at SIGGRAPH 2019. Accessed: 2025-02-05.

Raja, C., Balasubramanian, S., and Raghavendra, P. (2012). Heterogeneous highly parallel implementation of matrix exponentiation using gpu. *International Journal of Distributed and Parallel systems*, 3.