UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MÌMIR: A REAL-TIME INTERACTIVE VISUALIZATION LIBRARY FOR CUDA PROGRAMS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

FRANCISCO JAVIER CARTER ARAYA

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER
PROFESOR GUÍA 2:
CRISTÓBAL NAVARRO GUERRERO

MIEMBROS DE LA COMISIÓN:
VALENTIN MUÑOZ APABLAZA
IVÁN SIPIRAN MENDOZA
RICARDO BARRIENTOS ROJEL

Este trabajo ha sido parcialmente financiado por ANID/magister nacional/2022-22221291

SANTIAGO DE CHILE
2023

RESUMEN DE LA TESIS PARA OPTAR AL TÍTULO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
POR: FRANCISCO CARTER ARAYA
FECHA: 2023
PROF. GUÍA: NANCY HITSCHFELD
PROF. GUÍA 2: CRISTÓBAL NAVARRO

MÌMIR: UNA LIBRERÍA DE VISUALIZACIÓN INTERACTIVA EN TIEMPO REAL PARA PROGRAMAS EN CUDA

La visualización de experimentos que corren en GPU mediante plataformas como CUDA presenta características distintas en comparación a la de algoritmos tradicionales que utilizan solo CPU y RAM. Dado el volumen de datos involucrado y la velocidad con que se actualizan, a menudo es deseable poder observar gráficamente su estado, ya sea para presentarlos de forma más expresiva, monitorear su evolución o detectar errores en tiempo real. Esto es considerablemente más difícil utilizando soluciones orientadas a datos residentes en RAM, ya que implica transferencia desde y hacia la GPU para generar y actualizar la visualización. Este trabajo presenta el diseño e implementación de una biblioteca C++ que utiliza interoperabilidad entre CUDA y Vulkan para visualizar conjuntos de datos residentes en memoria de GPU en tiempo real.

La solución propuesta permite crear vistas de los recursos en memoria de GPU creados en CUDA de manera que sus modificaciones se actualicen en tiempo real sobre la visualización producida en Vulkan para varios tipos de datos. El esquema de la biblioteca permite generar visualizaciones con mínimas alteraciones al código fuente del experimento original, necesitando únicamente reemplazar las llamadas a las peticiones de memoria GPU que se quiere visualizar. Los experimentos realizados muestran que el rendimiento en visualizar nubes de puntos es hasta 12 veces más rápido en tiempo total de visualización con un *frame rate* hasta 9 veces más alto y con uso de memoria hasta 1.5 veces menor respecto a visualizaciones generadas con librerías orientadas al despliegue de este tipo de datos. También se muestra que la interoperabilidad puede utilizarse para mostrar conjuntos de datos de diversos tamaños sin pérdida de responsividad, y las opciones de sincronización entre CUDA y Vulkan pueden ajustarse para balancear el uso de recursos entre rendering y cómputo según el caso de uso. Por otro lado, se discuten las diferencias entre utilizar OpenGL y Vulkan como *backends* gráficos para generar las visualizaciones.

Abstract

Visualization of experiments running over GPU through platforms such as CUDA presents different challenges compared to traditional algorithms that use only CPU and RAM. Because of the data volume involved and its update speed, it is often desirable to graphically observe its state for ease of presentation, monitoring its evolution, or detecting errors in real time. This is not possible with solutions oriented to data allocated in RAM, as it implies bidirectional GPU memory transfers. This work presents the design and implementation of a C++ library using CUDA/Vulkan interoperability to visualize datasets allocated in GPU memory in real time.

The proposed solution allows to map writeable CUDA resources to Vulkan graphics handles so that modifications on them reflect over the resulting visualization in real time for various data types. The library scheme allows generating visualizations with minimal alteration over the original experiment source, needing only to replace calls to GPU memory allocations that will be displayed. The experiment results show that performance in displaying point cloud datasets is up to 12 times faster in total visualization time with a frame rate up to 9 times higher, while using 1.5 times less GPU memory compared to visualizations generated with libraries oriented to point cloud display. Further analysis also shows interoperability used in this way can be useful to show input sets of various sizes while keeping responsiveness, and CUDA-Vulkan synchronization options can further tune load balancing between rendering and compute tasks. Differences between using OpenGL and Vulkan as rendering backends for interoperability tasks are also discussed.

A mi familia que me ha acompañado durante todo este recorrido, y a Josefina que está próxima a comenzarlo.

Acknowledgments

I would like to thank the Temporal Research Group for their continued support and feedback over the process of this research.

This work was partially funded by the National Agency for Research and Development (ANID) / Scholarship Program / MAGISTER NACIONAL/2022 - 22221291.

TABLE OF CONTENT

List of Figures

List of Listings

Chapter 1

Introduction

Over the past decades, the scientific community has faced a need for processing data in greater amounts than the computing capability needed to handle it. Likewise, these results need to be presented in such a way that the information contained in them is conveyed effectively, which is often in visual form. Scientific visualizations allow researchers to exhibit results, find patterns in large-scale data, detect local anomalies and discover bugs in data generation. The special hardware architecture of graphics processing units (GPUs) is able to handle both needs at once. Visualizations are produced through graphics computing APIs such as OpenGL, Vulkan, DirectX and Metal, while general-purpose programming on graphics cards (GPGPU) is possible through platforms such as OpenCL and CUDA.

While there are multiple existing libraries or applications for scientific applications, few of them can be used directly with datasets residing in GPU memory, either by performance issues for large input sizes or because they cannot read previously allocated GPU memory directly. In such cases, a workaround is transferring the data from GPU memory to RAM (and maybe even exporting to a file in some cases) and load it in a visualization software, which reduces performance greatly. As a result, it is difficult to produce GPU visualizations of GPU data even though the hardware is more than capable of performing both tasks, and it should be possible to do so in a single pipeline. This work attempts to develop a GPGPU framework that would make such a pipeline possible, by designing a library that connects data generated with the CUDA platform to the modern Vulkan graphics API for displaying the results. Such a library would prove useful for visualizing problems such as large-scale simulations, graphics algorithms (e.g. raytracing) and iterative parallel algorithms, and for more specific sub-tasks like result validation and presentation.

1.1 Motivation

There are common GPU computing patterns that can be mapped directly to primitives supported by graphics APIs for producing meaningful visualizations, such as point clouds, meshes, textures and voxel sets. Using these mappings as a base, visualizations can be further augmented through the use of programmable shader programs, for example, to change point shapes, switch texture interpolation mode or modify scene lightning parameters. These advantages are often lost when using a higher-level API or library for display, or are provided through wrappers designed to read display data from RAM instead of GPU memory allocated through CUDA.

A different possibility is to use GPGPU capabilities provided by Vulkan and OpenGL through Compute Shaders, which allow computing in a similar scheme to CUDA kernels and can replace them for such purposes. However, as they are graphics-oriented APIs foremost, they have different programming models and idioms for producing the same result as a compute API like CUDA. Moreover, in terms of features, they don't provide ways to use the latest architectural advancements in CUDA-enabled GPUs such as Tensor Cores and/or Raytracing Cores. From an user standpoint, switching the experiment codebase from CUDA to another API just for enabling visualization capabilities would require a development effort that is likely bigger than the intended benefit. Instead, it would be expected to incorporate display capabilities to the existing code in the form of a library or extension, just as it would be in traditional solutions that read from RAM-residing memory.

A promising alternative consists in connecting the graphics API and the GPU computing platform directly via interoperability between the two. The main graphics APIs can interoperate with computing plataforms via explicit memory mapping operations. However, the programming effort and learning curve required to produce visual output with these graphics APIs is considerable, let alone to produce scientific visualizations with the mentioned interop scheme. The code needed for visualization would end up bloating the original CUDA experiment code, and is unlikely to be reusable due to amount of boilerplate and the specific characteristics of the data to display. Overcoming those problems would need designing a software architecture extensible enough to handle multiples types of visualizations, while keeping performance and customization compared to graphics API code written specifically to a particular CUDA code.

OpenGL and Vulkan are the main graphics API alternatives for fulfilling the above, but there are differences that may affect how they should be used to achieve that. OpenGL is the oldest and has both a tried and true, well-documented interface, and has evolved through the years to increasingly expose the differences between CPU and GPU execution flows from immediate mode to a more flexible programmable pipeline approach. Vulkan is the more recent alternative that already incorporates the lessons learned through the various OpenGL versions, but its API is considerably more verbose. Even though it is more complex and requires a greater development effort compared to OpenGL, but offers greater flexibility in choosing how and when to use features that are hard-coded in an OpenGL workflow and that may be helpful for implementing the proposed interoperability scheme.

1.2 Research questions

- Are dynamic Vulkan visualizations of CUDA-mapped memory more efficient in memory use and frame rate than the alternatives of generating the data in host memory and transferring it for rendering, or using OpenGL as the graphics backend?
- What changes or replacements would need to be made to existing CUDA code to display its results using Vulkan?
- How much can Vulkan code be abstracted from CUDA code while interoperating between both languages?

1.3 Hypothesis

A new high-performance Vulkan visualization library designed specifically for integration with CUDA code will perform better in memory use and frame rate for the interactive display of GPU-generated data than applications or libraries that read input data from RAM.

1.4 Objectives

1.4.1 Main objective

Design and implement an interactive visualization library in Vulkan for displaying 2D and 3D use cases (meshes, point clouds, textures and voxels), which are allocated in GPU memory that can be processed by CUDA kernels.

1.4.2 Specific objectives

- Research rendering algorithms and strategies for meshes, point clouds, textures and voxels in 2D and 3D domains, how to implement them with the Vulkan API and how to optimize them for Vulkan-specific features.
- Design a C++ architecture to visualize multiple CUDA-Vulkan memory mappings.
- Using the designed architecture, implement the visualization pipeline as a library.
- Provide interactive and customization capabilities for the visualizations generated with the library, such as setting colors and camera position.
- Determine how and when is more advantageous to use OpenGL or Vulkan as part of a compute-graphics interoperability setup.
- Measure the improvement of generating mapped memory visualizations over doing explicit memory copies, in terms of memory use and frame rate.

1.5 Contents

Chapter 2 defines relevant terms that are to be used throughout this work, reviews existing alternatives that could be used in absence of interoperability, and discusses relevant platforms for GPU graphics and computing. Chapter 3 states the design goals and details for a library implementation that fulfills the objectives stated above. Chapter 4 describes said implementation, including code structure and organization. Chapter 5 shows the experimental design used to validate the work described in the previous chapters, including the experiment environment and plots containing the results. The achieved results are also discussed at the end of this section. Finally, chapter 6 shows the conclusions obtained from the previous sections and proposes future work based on the presented results.

Chapter 2

Background and related work

2.1 Key concepts

This section describes the terms to be used throughout this work.

GPU: Shorthand for Graphics Processing Unit, they were conceived as a peripheral specialized in massively parallel floating-point arithmetic through a large number of cores in contrast to a CPU, which typically features less but more potent cores in terms of throughput. Though designed for accelerating graphics algorithms and routines, the parallelism that they provide can be used to accelerate general purpose programs (GPGPU), to the point that specialized compute units cannot be used for display have been released. Further hardware developments such as tensor core units further favor multiple uses for GPU, such as accelerating raytracing, machine learning and general purpose algorithms.

Device Refers to the GPU and its resources, such as memory (VRAM). May be a dedicated GPU installed in a PCI slot or integrated into a CPU (iGPU). As the purpose of this work is oriented for dedicated GPUs, *device* will refer to that kind of units (besides being typically slower, iGPUs do not provide as much features as a ”true” GPU).

Host Refers to the CPU and its memory (RAM). Though most of the heavy-duty computing in GPU algorithms is commonly performed on the device, the CPU is still needed to handle device context creation, control flow, work coordination and memory transfers (RAM-VRAM).

Computing API: In the GPU context, it is an *Application Programming Interface* that allows leveraging the compute capabilities of a device for GPGPU applications. A computing API allows GPU memory handling (allocation, copy, free) and execution of special accelerated routines that execute at a high level of parallelism, leading to a different programming model compared to traditional host-based languages and APIs.

Graphics API: Similar to computing APIs, graphics APIs provide low-level access to a device and its properties, such as VRAM and the rendering pipeline. This enables users to develop specialized applications that may offer better performance and/or resource utilization compared to higher-level alternatives, at the cost of greater code complexity and programming effort. The main graphics APIs are OpenGL, Vulkan, Direct3D and Metal, with the OpenGL being the oldest and Vulkan the newest. Others exist as specialized subsets targeting specific platforms, like WebGL for HTML and OpenGL ES for embedded systems.

Graphics resource: A data structure handled through a graphics API that provides information to a rendering pipeline about the content to be displayed. Resources include buffers and textures, which have an opaque, non-linear layout with built-in interpolation. Graphics resources should not be confused with GPU resources, as the last refer to availability of elements such as processing power and VRAM to distribute over current and incoming GPU processes.

Interoperability: Or *interop* for short, is the process of mapping the resources between GPU APIs, in a way that both sides can access that resource as if it were created normally from the respective source. Typically this would happen between a compute API and a graphics API to visualize the computed results, but it is also possible to do interop between graphics APIs, in which the allocations and display are handled from different specifications. As is the case for low-level APIs, an application attempting interop must explicitly handle details such as synchronization, aliasing and cleanup of the linked resource, which an interop API should also provide in the form of a set of specialized functions.

Compute kernel: Also *kernel* for short, is in the context of GPGPU a function or routine optimized for masively parallel execution over a GPU. As the content of kernel will be executed in parallel over a potentially large number of the available threads in a GPU, the programmer must also specify the number of threads and their arrangement.

Graphics pipeline Modern graphics pipelines introduce programmable pipeline stages that are executed in sequence Figure 2.1 shows a high-level overview of the main stages composing a pipeline.

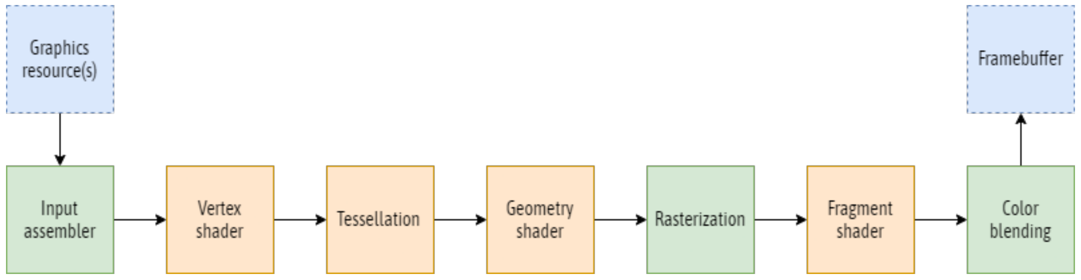


Figure 2.1: Overview of the graphics pipeline. Input/Output are in dashed blue, fixed stages in green and programmable stages in peach.

Shader program: Also referred as *shaders*, they compose the programmable part of the graphics pipeline for modern graphics APIs. Shaders for different pipeline stages can be mixed as long as the output structure of a previous stage matches the expected layout for the next one. **Compute shaders** are an exceptional case as they do not form part of any pipeline stage, and can be used for GPGPU in a similar way to compute kernels.

2.2 Related work

The type of work most relevant to this research are the ones that deal directly with interop between graphics and compute APIs. Three types of applications can be discerned: high-performance simulations that update state continuously through repeated kernel calls, GPU algorithms that use the graphics API for additional specialized compute work, and algorithms of graphical nature. All three types are natural candidates for interop, because visualizing them the traditional way would remove any performance gain or would greatly disrupt the workflow, which is obviously dominated by GPU work.

On the other hand, there is also a number of applications and libraries that support the desired visualization types or have performance-oriented goals, so they may be considered as alternatives to solving the stated problem, and hence are relevant for this work.

2.2.1 Interop uses

The earliest work found referencing interop is a fluid simulation that solves the Navier-Stokes equations in CUDA, displaying the results through interoperability implemented for OpenGL and Direct3D [8]. It serves as a demonstration for the first CUDA interop API (deprecated fully at CUDA 5) that consisted in registering the OpenGL resource index (or handle pointer for Direct3D) for enabling read/write from the CUDA side.

Rossinelli et al. [22] developed a 2D bluff body simulation in CUDA using OpenGL interop to map the vorticity to a texture and using its inbuilt interpolation capacities to solve the Poisson equation with a CUDA FFT over the interop-ed texture. Particle advection is handled in a geometry shader pass. This is a case of using interop for computational rather than display purposes. Özek et al. [30] also use OpenGL and CUDA as a compute environment for accelerating a Level of Detail (LOD) algorithm where the occlusion query is done in OpenGL and the buffer storing the pixels is mapped to perform parallel summation in CUDA, which is done to avoid atomic add serialization in the OpenGL fragment shader (the alternative without interop).

Abdellah et al. [1] perform Fourier Volume Rendering (FVR) entirely in GPU, which is possible by executing the computational core in CUDA and mapping the results to OpenGL textures with interop. Unlike most of the other cases shown here, this process involves multiple kernel executions and OpenGL context switches.

Hadji-Kyriacou et al. [9] developed a 3D raymarching rendering engine in real time, using CUDA/OpenGL interop to share buffers, reducing redudant memory use and improving performance. Interop-ed textures are loaded and bound through CUDA rather than OpenGL to avoid binding and mapping textures at every frame from the graphics side.

Lift [27] is a ray-tracing framework for educational real-time comparison of ray-tracing algorithms, which incorporates a denoiser implemented in OptiX. Vulkan/CUDA-OptiX interoperability gives the denoiser access to the image buffer in opaque memory layout, with is then copied to a linear memory buffer in CUDA so that it can be used as input for the denoiser. The back and forth memory copies are regulated by a Vulkan timeline semaphore.

Lipinksi et al. [14] use OpenGL/CUDA interop for GPU image compression, by mapping the loaded GPU texture to CUDA so the image data can be encoded by Thrust algorithms without duplicating GPU memory or transferring to host.

Tang et al. [29],[28] developed and algorithm for efficient Computer-Generated Hologram (CGH) calculation that uses interop for obtaining camera input from OpenGL, passing it to the CUDA kernel performing the fast CGH calculation, and returning the CGH result back to OpenGL for display. Synchronization in this case is handled by the interop mapping; when memory is being shared between APIs CUDA is able to compute the result, and when unmapped it is time for OpenGL to render the calculated CGH.

Farokhmanesh et al. [7] present a neural network for reconstruction of large size 3D ensembles at a reduced time and memory cost, which enables to fit the model in GPU memory for presentation. This display is handled using CUDA/Vulkan interop of the buffer holding the outputted correlation estimates, which is then mapped to a 3D texture.

2.2.2 Visualization applications

ParaView[2] is a high-performance framework for data analysis and visualization based on VTK. It supports both interactive visualization with a graphical front-end and a programmable pipeline via Python scripts. It can display 3D data derived from regular and adaptive resolution meshes, (un)structured and polygonal grids, data tables and composite datasets from the mentioned types. It also supports visualizations derived from the original data such as isosurfaces, vector fields and streamlines. It supports loading datasets with formats common to most Python graphical libraries. It also has a CUDA plugin for performing GPU-accelerated transformations of the previously loaded data.

Blastsight is an interactive 3D visualization application for academic and industrial mining uses, developed in Python and OpenGL [25]. It can render meshes, points and lines or tubes. As mining datasets such as block models require rendering a great number of elements, it has been designed with performance measures such as a "turbo" mode for concatenating meshes in a single logical mesh, in order to increase rendering performance for arbitrarily-sized datasets. Its functionality can be considered as a high-performance subset of Mayavi.

Camarón is a visualization tool focused on mesh rendering and analysis, written in OpenGL/C++ [5]. Apart from reading meshes from various formats and displaying them with their statistics, it can also display isolines and isosurface representations codified in the source files. Further work[3] over the library increased performance by 75% while using up to 18% less memory.

Datoviz is a performance-oriented scientific data visualization library and intermediate level graphics API, writen in Vulkan/C++ with native Python bindings [21]. Part of its code and design ideas are derived from VisPy [4], a higher-level API that uses OpenGL instead of Vulkan for its graphics backend. It supports display of markers, lines and meshes in 2D and 3D space, while also providing plot generation capabilities similar to traditional Python plotting libraries such as Matplotlib. CUDA-Vulkan interoperability is not currently supported, but is a long-term planned feature.

Mayavi is a general-purpose 3D visualization library with interactive and scripted modes, written in Python and VTK [19]. It allows operation with raw Python data such as `numpy` arrays and maps it implicitly to VTK structures, unlike ParaView. This allows for easier integration in existing Python-based scientific workflows such as scripts or Python notebooks. Compared to the other alternatives listed here, it is not as well-optimized to display large datasets.

The Point Cloud Library [24] (PCL) contains a visualization module based on VTK and written in `C++` for rendering 3D point clouds. It also provides algorithms for feature estimation, surface reconstruction, model fitting and segmentation, that are useful for robotics and sensor applications. Point clouds can be set to N-dimensional datasets through `C++` templates and produce the corresponding visualization. Point cloud visualizations can be adjusted for drawing different 3D shapes, color and geometry handlers. Though the base visualizer class can provide interactive visualizations of static datasets, there is an advanced mode that allows manually calling refresh functions for updating the point cloud data.

2.3 Background

2.3.1 Computing APIs

The two main APIs considered are CUDA and OpenCL. At first glance, the former has larger and modern feature set (for example, tensor cores of Nvidia GPUs), while the latter is agnostic to GPU vendors. Both are competitive in terms of performance, but OpenCL is more sensitive to architectural details [6]. Besides that, each API has its own specification and data structures, so implementations of a same program will be different without a compatibility interface such as HIP. Both APIs use the SIMT model (Single Instruction, Multiple Threads), that allows divergence between threads (as part of a branching instruction, for example) at the cost of serialized execution between a same group of threads, also known as *warps*.

2.3.2 Graphics APIs

Discussion here is centered on comparing OpenGL with Vulkan, as they both are multi-platform APIs but the former is considerably older than the latter. Vulkan can be considered the spiritual successor to OpenGL, but is not a direct replacement; in fact, recent OpenGL versions have adopted features introduced first in Vulkan starting from version 4.6. Figure 2.2 shows the Vulkan pipeline as defined by the API specification, and is the full version of the pipeline shown in figure 2.1.

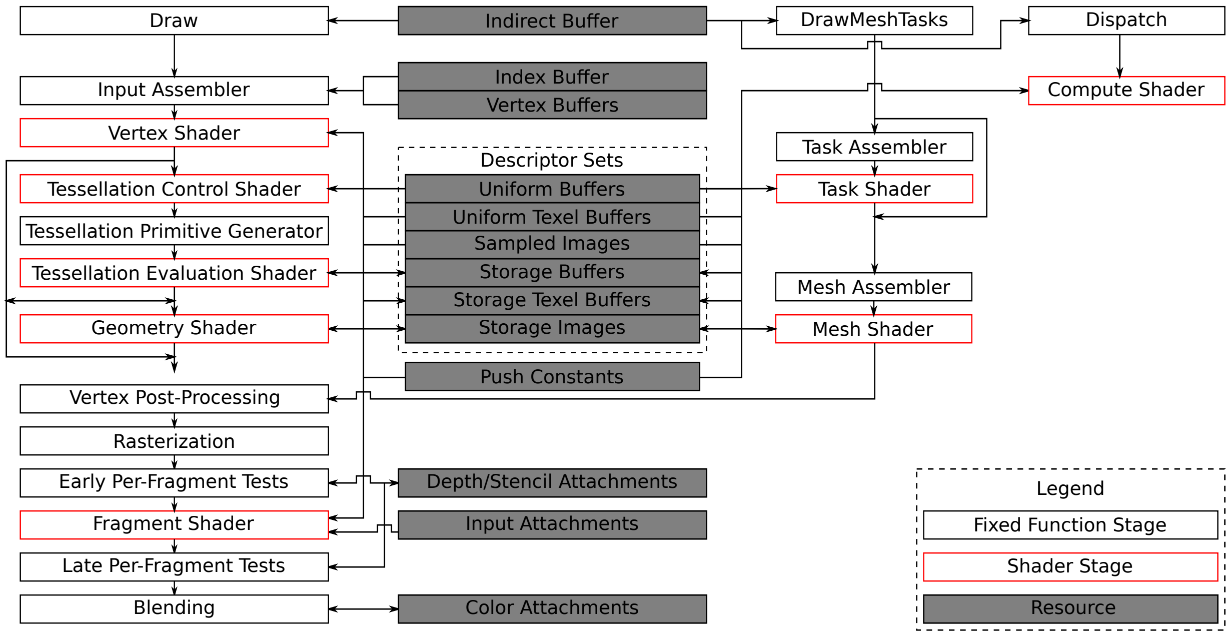


Figure 2.2: Vulkan pipeline diagram.

Vulkan has structures like the instance, swapchain, render pass, descriptor sets, pipeline objects and synchronization structures that need to be manually created but are automatically provided in OpenGL, like the default framebuffer. This leads to a direct increase in programming complexity and verbosity, but the increased explicitness helps in reducing code ambiguity and side effects. For example, manual synchronization means the developer is free to adjust its behavior to better suit the application at hand.

The Vulkan memory model has separate memory and resource structures, unlike OpenGL where each resource owns the memory region allocated when creating it. Resources in Vulkan are handles that specify how to access a memory region to interact between its contents and the graphics pipeline. Though this scheme is more complex, it offers greater flexibility in resource usage by making it possible to alias one or more resources to a single memory region, either partially or completely. For example, this can be used to reuse memory bound to a buffer to be used later with an image, or to interpret said buffer as an image with the same size. It is also possible to join multiple memory allocations in a single one and bind resources with offsets from start the larger memory region, increasing memory efficiency and performance. All memory allocations must satisfy alignment requirements, and the memory flags supplied when creating them determine the purpose of that resource and the memory heap that supports it. For example, device memory can be flagged for being mapped to host memory or device-only access, which will be respected by the API and may be used for optimized access for that memory. Figure 2.3 shows how the different resource parameters determine the heaps supporting them.

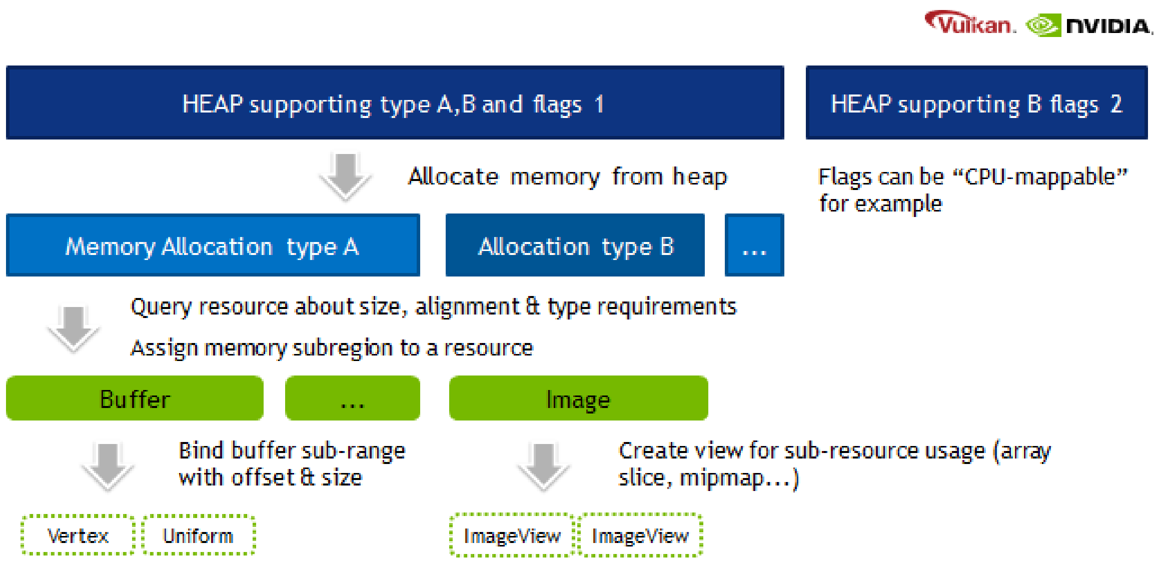

Vulkan.
NVIDIA.
HEAP supporting type A,B and flags 1
HEAP supporting B flags 2
Allocate memory from heap
Flags can be "CPU-mappable" for example
Memory Allocation type A
Allocation type B
...
Query resource about size, alignment & type requirements
Assign memory subregion to a resource
Buffer
...
Image
Bind buffer sub-range with offset & size
Create view for sub-resource usage (array slice, mipmap...)
Vertex
Uniform
ImageView
ImageView

Figure 2.3: Vulkan memory heap scheme.

Vulkan has also been noted to perform similarly to CUDA for compute work, with an slight performance edge for CUDA at the cost of vendor lock [11]. Hong et al. show this comparison holds true for equivalent implementations, but CUDA supports features like recursion that improve performance considerably and that cannot be implemented in Vulkan.

2.3.3 Shading languages

Shading languages were introduced with GLSL starting from OpenGL version 2.0, as a way to interact with the programmable rendering pipeline introduced there [13]. Since then, the language has evolved with new OpenGL versions and new languages have appeared such as HLSL (High Level Shading Language) for Direct3D and MSL (Metal Shading Language) for Metal. More recently, SPIR-V rose as a intermediate-level language [12] that can be generated as output from any of the previously mentioned high-level languages. This allows generating portable, lightweight and protected shader code [13] that can come from any source as long as the right toolchain is used to compile it to SPIR-V.

Even though SPIR-V is useful for language-independent shader code, generating shader variants to cover multiple use cases is still an issue. One option is to generate shader variants through the preprocessor, but is prone to errors as invalid generated code may go unnoticed. Shader metaprogramming languages [15] are an alternative for generating shader code from a source file instead of directly compiling the actual source file. Slang [10] is a modern shader metalanguage that can be used for these ends. It can generate output for different targets such as D3D12, Vulkan, D3D11, OpenGL, CUDA and CPU. Slang code has a similar syntax to HLSL but with additional syntax and language constructs, so it should not be compiled with the HLSL toolchain. It is compiled either with the `slangc` CLI compiler or through the `C/C++` API. The latter requires additional setup from needing to create structures for compile sessions, compiler requests and loading shader modules, but offers finer control over the whole compilation process. Another advantage for the Slang API a program can perform reflection from the shader code parameters, so it is possible for example to get binding indices at runtime instead of hard-coding the values, which is prone to errors and regression issues. This increases integration between shader and host code and reduces the possibility of shader runtime errors.

2.3.4 Interoperability

A system must have a CUDA-enabled device and a Vulkan-enabled device to qualify for interop. These requirements are not difficult in practice for workstations, as most recent consumer-grade Nvidia GPUs that support CUDA also support Vulkan, starting from the Maxwell GPU architecture and beyond. However, it is also necessary that the contexts for both APIs be instantiated on the same device. A system may have multiple devices that support any of the APIs separately, but not both at once. A common case example happens on laptops that have an iGPU and a GPU, where the first may support Vulkan but not CUDA, but only the second may support both. This implies CUDA and Vulkan device lists may not match or have the same entries, so it is necessary that at least one device exists on both.

When a candidate device is identified, it must fulfill a series of requirements to qualify for interop. Device suitability depends on having both a display and graphics queue, supporting all Vulkan extensions[1] required by the library, and the swapchain supporting at least one surface format and one present mode. The external memory Vulkan extension is needed for allocating device memory that can be imported by CUDA, which is requisite for interop, and the external semaphore extension is required for synchronizing compute and graphics work

[1] `VK_KHR_external_memory_capabilities` and `VK_KHR_external_semaphore_capabilities`.

(described in section 3.3.2), but it is not strictly necessary for interop. However, both are included as core functionality starting from Vulkan 1.2.

It is not possible to do interop using a CUDA-only device to transfer data to a Vulkan-only device, as it will result in runtime errors when trying to access the interop data. When using one or more compute-only devices, all visualization data must be transferred to the interop device first, and it must have Vulkan and CUDA initialized. In that case, computing devices do not need to instantiate or support Vulkan, but they must first copy data to a interop-enabled resource, whether through peer-to-peer copy or via a staging buffer in host memory.

Chapter 3

Analysis and design

As both CUDA and Vulkan have native `C++` APIs, that language is also a natural choice for implementing the proposed solution. In order to encapsulate all Vulkan code for visualization from the existing CUDA code (that is, the experiment code), this implementation should be packaged as a library, using `C++` object-oriented features to group Vulkan resources according to the visualization they belong to, which would allow creation of multiple visualizations that could be combined in a single scene. Shader resources should also be included in these structures, but should not be managed directly by the library user. Instead, all graphics resources should be indirectly managed through user parameters that manage resource creation details as a flowchart. For example, those parameters could be used to decide whether a buffer should be a vertex or index buffer kind, or if an image object represents a 2D or 3D image.

Accounting to the above, a `C++` library using Vulkan and CUDA named *Mìmir* is proposed. In this design, Vulkan handles visualization and CUDA binds the interop resources so that they can be used by the client program. Library interaction with user code is divided into initialization, interop resource creation (named as "views") and display functions.

3.1 Engine design

The library interacts with CUDA code through an engine class, which handles interop and visualization logic as shown on figure 3.1. An engine instance needs to be initialized before starting any interop work, and this process involves creating the display window, setting up the Vulkan context, choosing the GPU to use, creating framebuffers, render passes and the swapchain structure. During display, the engine handles the rendering loop and synchronization with the compute work. All user interaction with the library passes through the engine instance; this includes retrieving the interop resource handles and starting the visualization.

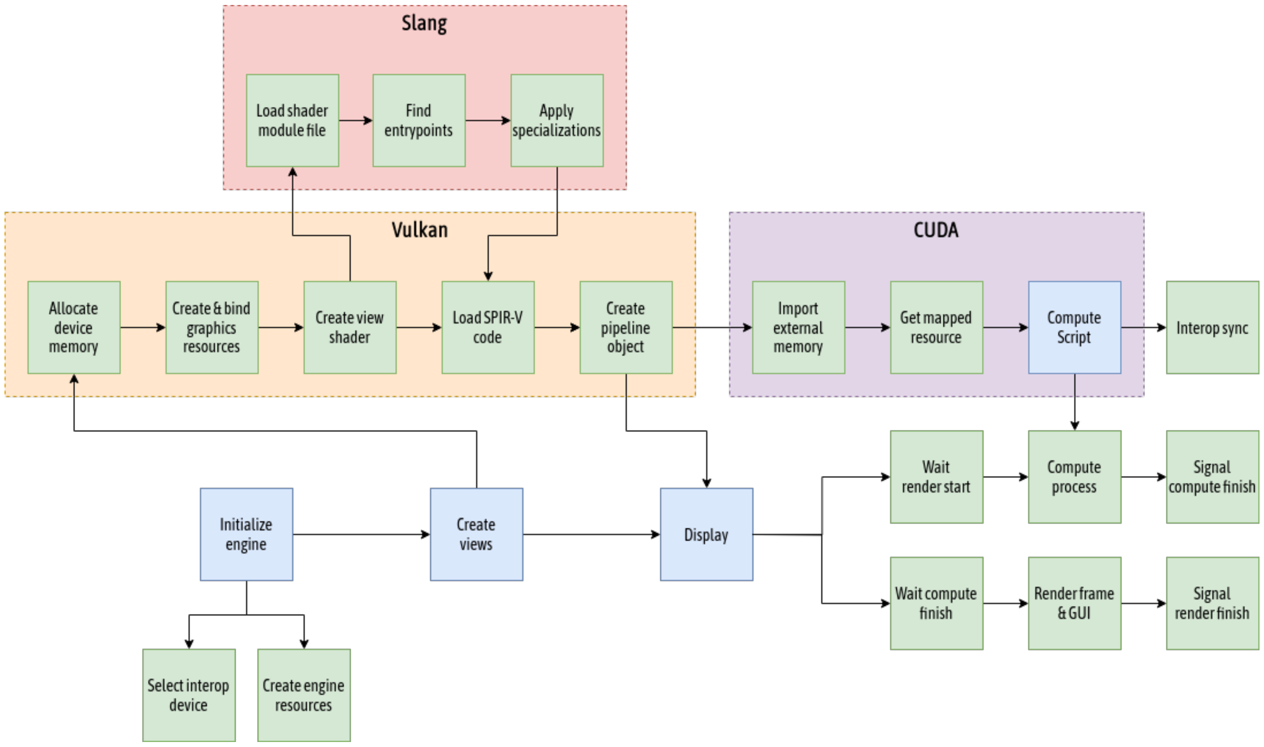


Figure 3.1: Overview of the engine flow. Blue stages are handled by the library user, while green stages are managed by the library itself.

3.1.1 Initialization

Before producing any graphical output, the Vulkan API needs explicit initialization of a variety of other resources (named engine resources in figure 3.1) apart from buffers and images. Only those resources not directly interacting with interoperability are listed for this stage.

- *Instance:* A handle for the current Vulkan context being initialized.
- *Surface:* Stores graphics output after all rendering steps are completed, which then are ready for being presented to screen.
- *Physical device:* A handle to the actual device(s) involved in Vulkan operation. A physical device must support Vulkan to be visible at this stage. For the library, this device will be the interop GPU.

- *Logical device:* A handle to the device instantiated with the requested Vulkan features. For the library, these features include having a graphics and present queue, for performing rendering operations and displaying output to screen respectively. This handle is used throughout most Vulkan functions to identify which device the API will work with.
- *Swapchain:* This structure manages how the images are presented for display. The present mode used to create the swapchain has direct effect over visualization frame rate and GPU energy usage. This value is set as parameter in the library. A swapchain is associated to a set of swapchain images, whose value depends on the capabilities of the current device.
- *Command pool:* This resource allows creation of command buffers, which are used to record and enqueue Vulkan commands for later execution. As the design considers a single rendering thread, only one command pool is needed, and one command buffer per swapchain image is used.
- *Render pass:* An object that encapsulates multiple rendering phases (or subpasses) over a set of output images [26]. Currently, all visualizations use the same render pass, created at this stage. This render pass is used as input for framebuffer and pipeline object creation.
- *Framebuffer(s):* Vulkan has no default framebuffer unlike OpenGL, so the engine must create one for each swapchain image. Each framebuffer is created with color and depth targets.
- *Descriptor sets:* Handle to a set of resources bound to the pipeline as a group [26]. This allows bind shader parameters to specific graphics resources. Unlike OpenGL, it is not possible to bind resources individually.
- *Fence(s):* A structure for Vulkan host/device synchronization, that allows host flow to wait on a barrier to be cleared by a device function. Once fence is created per swapchain image, which is used to wait for the command buffer associated to that image to finish.
- *Binary semaphore:* A structure for Vulkan queue synchronization that can wait/signal for zero/one values. It is used to ensure that rendered frames on the graphics queue are ready for presenting.

3.1.2 Device

Creation of the above resources is verbose and may obfuscate the proposed design. To separate the above from interop structures, a *Device* class is created. This structure represents the interop GPU, and as such, it holds the handles of the physical and logical devices so it can create and destroy the various graphics resources as needed. The CUDA interop API (the CUDA box in figure 3.2) is further encapsulated by a *InteropDevice* class to separate graphics and compute workflows.

3.2 Views design

The view structure encapsulates the many interop resources and auxiliary structures needed to translate user-defined parameters into visual information effectively. A view initializes and stores interop resources of a dataset in such a way that altering its contents also changes the associated visualization accordingly. This occurs by mapping the Vulkan graphics resource with a CUDA resource handle through the interop functions, as shown on figure 3.2. For common linear buffers, this handle is a raw pointer to the beginning of the interop memory region, but it is also possible to interop with CUDA opaque memory layouts, in which case the interop handle is a CUDA texture object[1]. As a Vulkan image can be *mip-mapped* (have multiple levels of detail or mip-levels), it is also possible to get multiple CUDA texture handles from a view creation call.

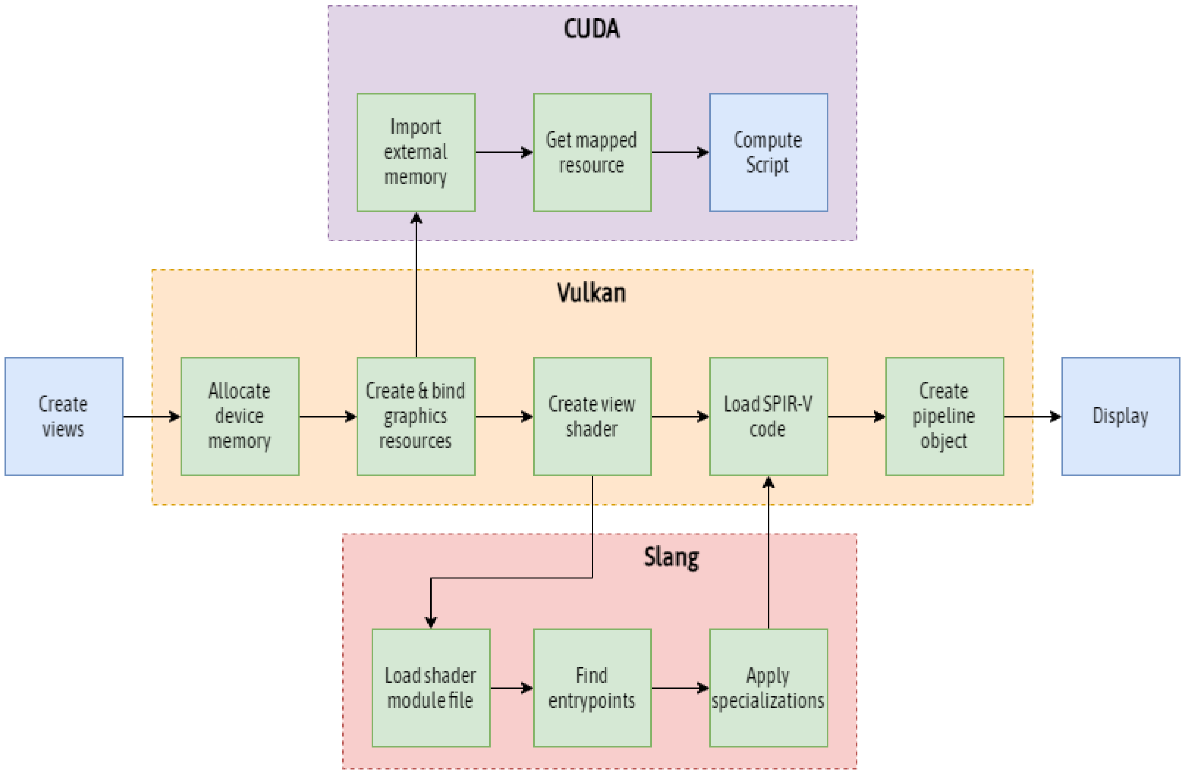


Figure 3.2: Overview of the view creation flow. Blue stages are handled by the library user, while green stages are managed by the library itself.

View parameters are used to describe the various classifications a data visualization can have. The *data domain* parameter indicates if a domain is two or three-dimensional; even though all visualizations are designed to be displayed in 3D space, the actual data contents can be displayed in a plane or throughout all space. *Domain types* indicate whether the domain space is structured or not. Currently, structured space domains can only be arranged

[1]CUDA texture objects are read-only. Surface objects offer read-write capability.

in square (for 2D spaces) or cube (for 3D space) layouts, which means positions for these domains can be inferred by the engine instead of being supplied explicitly as an input. *Data types* can be any one of marker, edge, voxel or image sets, and the setting of this parameter determines how the interop resource will be passed to the shader stage before rendering. Finally, the resource type is used to know which Vulkan resources are to be matched with a CUDA resource for interop, using the following guidelines:

Buffers These structures have linear layout data access, and are mapped to CUDA array pointers. The *structured* or *unstructured* prefixes refer to the spatial structure of the data to be stored in the buffer. Unstructured spatial domains have the positions of their elements throughout space, so the position of an element must be provided explicitly. On the contrary, structured domains have the positions of their elements implicitly defined by some regular ordering, meaning those positions can be generated and used internally by the engine without the user needing to provide them.

Textures Similar to buffers, textures are further divided by layout, which can be opaque or linear. Linear textures are mapped to linear memory (so they are also structured buffers), while regular textures represent opaque layout access and are mapped to cuda mip-map objects, which can hold one or more `cudaArray` objects per mip level.

Source Code 3.1: View parameters structure.

```
// Specifies the number of spatial dimensions of the view
enum class DataDomain { Domain2D, Domain3D };
// Specifies whether the data space follows a regular structure or not
enum class DomainType { Structured, Unstructured };
// Specifies the CUDA resource that will be mapped to the view
enum class ResourceType { Buffer, Texture };
// Specifies the view type that will be visualized
enum class ViewType { Markers, Edges, Voxels, Image };
// Specifies the data type stored as element of a view
enum class DataType { Int, Char, Float, Double };

struct ViewParams {
    size_t element_count; // Number of elements in the view
    size_t element_size; // Bytesize of each element
    uint3 extent; // Data space bounds
    DataType data_type;
    DataDomain data_domain;
    DomainType domain_type;
    ResourceType resource_type;
    ViewType view_type;
};
```

View parameters are defined in 3.1. Combinations of these provide flexibility in input data type, number of elements and padding size per datum, while the logical meaning of the data to display is contained in the data type field. That is, when displaying voxels, the `element_count` field would refer to the amount of voxels to evaluate in the shader pipeline.

3.3 Interoperability design

Interoperability is based on the NVIDIA samples for CUDA-Vulkan interoperability [2]. In the proposed design, interop work can be divided into three stages: ensuring it can be done and identifying the target GPU, creating the interop resources and coordinating the APIs working over those resources.

3.3.1 Setup

The interop-enabled resource (or resources, depending on visualization type) mentioned above is allocated at this step, and is interacted with through the view structure. The engine handles view creation with the `createView` method, designed to have a similar signature to `cudaMalloc` since it should be used the same way: the double indirect pointer is modified to hold the address to the beggining of the newly allocated array in device memory. Instead of passing allocation size like in `cudaMalloc`, view creation requires a `ViewParameters` structure that contains all the values specifying the view structure and interpretation for rendering. In this case, the allocation size is derived from `element_size` $\times$ `element_count`, with the fields passed as parameters. Using parameter structures as function inputs in this way is more common in Vulkan than in CUDA, but it is still present in the latter for functions like `cudaMemcpy3D` and its corresponding `cudaMemcpy3DParms`.

The reason of `createView` having a similar signature to `cudaMalloc` is that interop memory allocation must be made from Vulkan and exported to CUDA, but it cannot be done the other way around. For creating interop-able memory, Vulkan handles the memory allocation like normal, but the resource bound to it must be created using the appropriate

[2]`https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/vulkanImageCUDA`. Retrieved 2023-06-23.

flag for buffers[3] or images [4]. Memory created with this flag can be later imported by CUDA and mapped to the corresponding resource (buffer or image).

3.3.2 Synchronization

Once an interop resource is initialized, both CUDA and Vulkan can write and read from it as if it were created natively from the respective API. This means data-races will occur unless some synchronization policy handles concurrent access order. In order to have compute and rendering work run separately

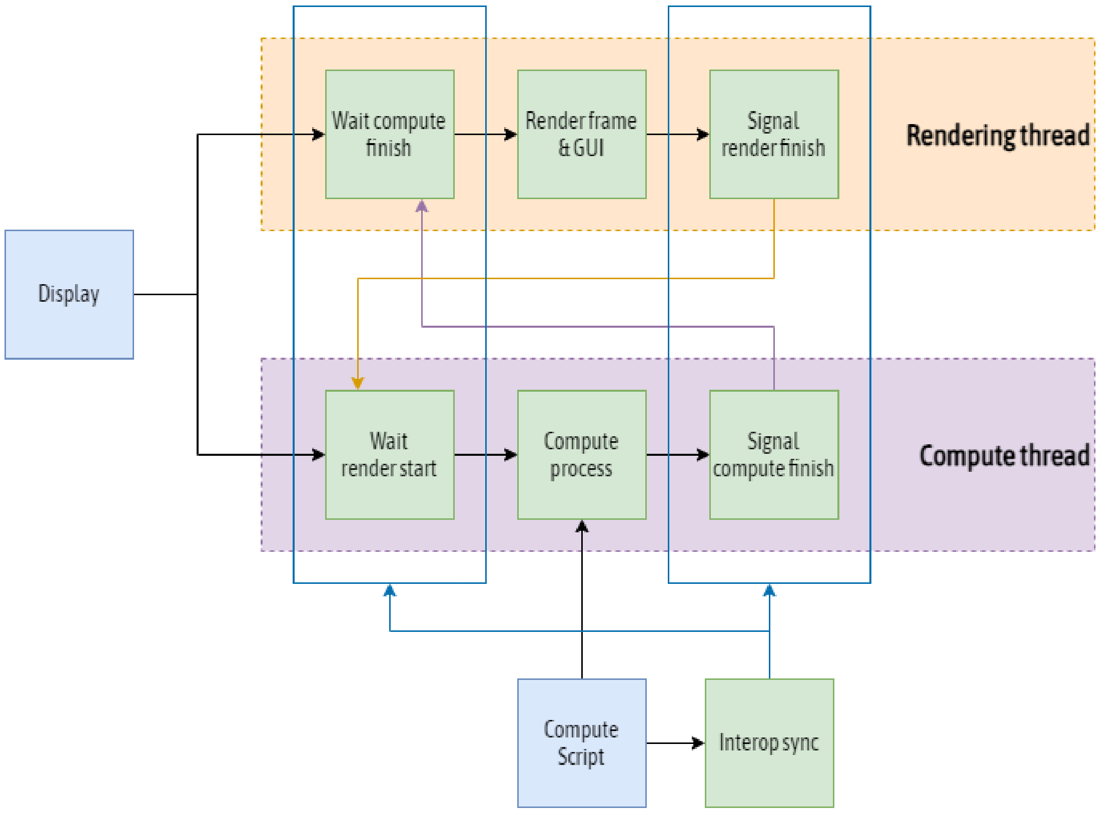


Figure 3.3: Overview of the interop synchronization scheme. Blue stages are initiated by the user, while green stages are managed by the library itself.

The synchronization structure and since it must be an interop resource to communicate between APIs. One such resource is the external timeline semaphore, which has various advantages over older Vulkan binary semaphores: they can wait and signal for values other

[3]`VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO`.
[4]`VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO`.

than 0 or 1 (the *timeline value* must be a 64-bit monotonically increasing integer), signaling for a value advances the timeline state to it, and unlocks any functions waiting for values equal or lower than the signaled one. Also, they allow host-device communication, unlike binary semaphores that can only coordinate device flow. The timeline value of a semaphore can be queried from host without being waited on, if necessary. The engine uses a single timeline semaphore, created at engine initialization and used throughout its lifetime. It is created in the same way as interop buffers and images; that is, allocated in Vulkan with the appropriate flag and imported by CUDA. After that, each API has their own semaphore handle with their own wait/signal functions, but is the import function that performs the linking between the two.

Algorithm 1 CUDA-Vulkan interop with synchronization

```
Require: $V = \{v_1, ..., v_N\}$ list of view structures
Require: $T$ interop timeline semaphore
Ensure: $F(V)$ framebuffer with color target applied to it
 1: procedure displayInterop($V, T$)
 2:     $t_w \leftarrow 0$
 3:     $t_s \leftarrow 1$
 4:     while running do
 5:         $vulkanWait(T, t_w)$
 6:         renderFrame()
 7:         $vulkanSignal(T, t_s)$
 8:         $t_w \mathrel{+}= 1; t_s \mathrel{+}= 1$
 9:         $cudaWait(T, t_w)$
10:         $t_w + = 1$
11:         gpuProcess($v_i$)
12:         $cudaSignal(T, t_s)$
13:         $t_s + = 1$
14:     end while
15: end procedure
```

Listing 1 shows how the timeline semaphore is used to synchronize rendering and compute work. Vulkan first waits for any pending CUDA process to end before starting to render the next frame, and signals to CUDA that rendering finished before starting compute work. This process is needed even in a single-threaded flow because CUDA and Vulkan device functions are non-blocking, so device may be working some frames behind host flow. **gpuProcess** is any CUDA process working over interop-mapped memory for one or more views in V. This would typically be a kernel(s) launch over the array or texture associated to any v_i, but it could also be a `cudaMemcpy` with the interop resource handle as destination.

This design allows for different execution modes depending on how the synchronization should be handled. The first mode ("sync" mode) uses as arguments the GPU process function along the number of times it should be called. This has the drawback of requiring the user to encapsulate **gpuProcess** into a function, which may require significant refactoring depending on how the code is organized, but also fits to a limited number of use cases, mainly those that run iterative algorithms without flow control between steps. The second mode ("async" mode) removes the previous issues by requiring explicit CUDA to Vulkan synchronization in user code, but at the same time gives users more control over the synchronization points. This mode works by launching a rendering thread that runs algorithm 1 without lines 9 to 13, while the main thread executes the compute workflow (that is, the flow of the CUDA program calling the library API). The *cudaWait* and *cudaSignal* must then be called from the compute workflow before and after the GPU modifying a view respectively. The semaphore and its timeline values still remain as part of the engine and do not have to be managed explicitly by the user in this mode.

It is also possible to completely disable interop synchronization, called "de-sync" mode, where no semaphore functions are called and timeline values do not increase. A lack of synchronization means compute and rendering threads run independently, so compute processes may finish faster than the frames are being rendered, or the other way around, where the engine may potentially render many frames without any compute step completing yet. The first case has an effect on iterative experiment visualizations, where some visual states may be skipped if the compute work is finishing faster than the graphics work is able to present frames for display. The second case is not an issue for the visualizations themselves, but it may impact interop performance if the GPU is being overused. This could happen because both APIs are sharing device utilization, so extra resources being used in rendering equal to less compute power when the device is running at full capacity.

3.3.3 Frame limit

Under this design, visualization with interop sync enabled and VSync present mode would be the "safe" choice, but it restricts the frame rate to the refresh rate of the monitor. Using immediate mode removes the rate limit, but can produce visual artifacts and leads to the performance issue mentioned above. The engine also provides an FPS target value to set an upper limit over the display frame rate. Whenever the engine would be presenting more frames per second than the target value (when set), the rendering routine will sleep to match the limit rate. Because render work runs on its separate thread, this throttling will not have direct effect on compute performance.

3.4 Visualization design

This section encompasses all processes involved in transforming interop structures into visual information through shaders. Once the engine creates the view resources according to its parameters, there is still need to generate the programs that can process those resources into fragment output. The view specification supports adding views with different number of spatial dimensions, data type and size, so shader variants for each of those parameters must be available at runtime for each of the supported usage patterns (these variants could exist as source files or be generated when requested). Also adding lightning modes and other ambient effects into the mix would result in combinatorial explosion of shader variants to maintain. A possible solution consists in using pre-processor macros with a compile-time script, which is prone to runtime errors for the generated shaders. Another option would be using branching (preferrably at compile time) to select shader variant based on the parameters, creating a number of "uber-shaders" of big size and difficult to read. For these reasons, neither alternative scales well.

The previous problems can be avoided using the Slang shader metalanguage, so that the engine can generate shader code at real time when creating a view. Slang code can be linted and organized in modules and duplicated code can be reduced through the use of generic methods that perform the same logic over different types that support it. Shader source code is then organized into the different view types supported (markers, edges, images and voxels). Values of the view parameters translate into strings that match into Slang specialization names, and all specializations must be defined when compiling a shader or the process will fail. Each view type also defines specializations the describe the underlying data layout of a view and also affects its appearance. Specializations for the different input data types (`char`, `int`, `float` and `double`), allowing the shader code to correctly interpret them, so for example a `Float2` specialization compiles a shader that reads a vector of 2 32-bit floating point values[5] as elements for the input buffer or texture. There also are unique specializations to each use case; marker views allow setting the marker shape (circle, square or diamond), while image view specializations control how to sample the associated texture (read all channels or combine them into a grayscale value). Currently there are no implemented specializations for edge and voxel view types.

[5]`vec2` in GLSL.

Chapter 4

Implementation

This section contains implementation details of the proposed library design, including engine and view creation, shader composition, source code structure and how to use the Mìmir library. Sample use cases are provided for all the proposed visualization output types: markers, edges, textures and voxels. These synthetic cases show the code needed to create a visualization for each output type and how the result changes with the different view parameters and options.

4.1 Initialization

The initialization stage includes all code that is needed to produce visualizations, but before displaying them. This includes initializing the Vulkan context and resources, setting up the display window and allocating device memory and importing it for use as interop buffers and/or textures, which then can be read from shader code to produce visual output.

4.1.1 Engine class

The Mìmir engine is implemented as a class that calls code from other modules, as described in section 4.3. Apart from Vulkan and interop resources, it also handles all structures needed to output to screen, such as windows. GLFW is used to interact with windows, but other backends could be used as long as they provide functions to create a Vulkan surface object to interact with the window handles.

The engine class has its own set of options (shown in listing 4.1) that can be modified before and after initialization. Sensible defaults are used in case any of these option values is left unspecified. Window size is self-explanatory but also affects framebuffer size, and consequently, performance and memory usage, so greater window sizes will consume more GPU resources. This value can either be set to specific values through the library API or with classical mouse drag motions. Present mode can be Immediate or VSync, meaning finished frames will be presented either immediately or after each monitor vertical blank respectively. This means immediate mode allows unlimited frame rate while VSync will cap engine FPS to the monitor refresh rate.

Source Code 4.1: Engine options structure.

```
struct EngineOptions
{
    std::string window_title = "Mìmir";
    int2 window_size         = { 800, 600 };
    PresentOptions present   = PresentOptions::VSync;
    bool enable_sync         = true;
    bool fullscreen          = true;
    uint target_fps          = 0;
};
```

The available synchronization modes described in section 3.3.2 are also set in this section. The boolean `enable_sync` field determines whether to use interop synchronization or not. In the latter case, any synchronization calls in user code will become no-op.

4.1.2 Device selection

As part of its initialization and before being able to perform any interop work, the engine must first identify and select a suitable device as stated in section 3.3. CUDA and Vulkan provide their respective functions for listing and selecting devices, but their data structures are different and the order of the lists may not match, since their indexing depends on the API. To ensure both APIs are selecting the same device, entries from both lists can be matched by comparing their GPU-UUID [1] metadata.

[1]Universally Unique IDentifier.

Automatic GPU selection for interop is done by iterating through available devices from the CUDA API, finding the Vulkan device whose UUID matches the CUDA device, and checking whether it is suitable for visualization, using Vulkan functions to query and check the device properties stated in 3.3.1. The first device that fulfills the above criteria will be deemed as suitable and selected as the one used by the library, though the automatic selection can be further improved by implementing additional criteria for choosing the best candidate, such as the amount of VRAM or CUDA compute capability. It is also possible to set the interop GPU manually by passing the CUDA device ID [2] and skipping the loop over all CUDA devices, but the selection will be rejected if the device also does not support Vulkan, and so the graphics engine will not be able to start.

4.1.3 View creation

A view structure is composed of several Vulkan resource handles and a container for all user-facing values, which in turn determine how the resources are initialized. Each view maps to a single interop resource, and multiple views can be created in the engine, where they are added to an array of current views. The view creation function uses a structure of view parameters and a handle to a CUDA resource as inputs, creates the Vulkan interop resources based on the parameters, then modifies the handle to point to the mapped CUDA resource. This way, view creation works in a similar way to the `cudaMalloc` family of functions, so adding interop resources through the library consists in replacing the original CUDA memory allocation code with the view creation function. Views depend on *parameters*, which determine resource creation and cannot be modified after the view is created, and *options*, which change how the view is displayed and can be adjusted at real time. Listing 4.2 shows all available options.

Source Code 4.2: View options structure.

```cpp
struct ShaderInfo {
    std::string filepath;
    VkShaderStageFlagBits stage = VK_SHADER_STAGE_ALL_GRAPHICS;
};

struct ViewOptions {
    // Customizable name for the view
    std::string name;
    // Flag indicating if this view should be displayed or not
    bool visible = true;
    // Default element color if no per-instance color is set
    float4 color{0.f,0.f,0.f,1.f};
    // Default element size if no per-instance size is set
    float size = 10.f;
```

[2] As per the `cudaSetDevice()` function.

```
    // External alternate shaders for use in this view
    std::vector<ShaderInfo> external_shaders;
    // For specializing slang shaders associated to this view
    std::vector<std::string> specializations;
};
```

The visibility flag can be toggled to prevent a view from being displayed, which can increase performance when a view is not needed. It is possible to set element color and sizes for each view, so even if there are two views with the same data type, they will not share the same configuration. A view can also use external precompiled SPIR-V shaders, provided they match the shader code structure used by the library.

Uniform buffers are managed directly in the engine class for all views instead of being allocated and managed per-view like the buffer or image handles. This allows the rendering function to load the uniform buffers for the different frame indices as offsets from the singular memory region. The engine manages uniform buffer memory similar to a `C++ STL` vector. If a view is removed, the region used by the corresponding uniform buffer will be marked as cleared and refilled at the next rendering call, and the offsets are recalculated accordingly. When creating a new view that exceeds the current memory capacity, the current memory is freed and a new one with the appropriate capacity is created.

4.2 Visualization

This section includes details for all processes that directly impact on the visual output once all interop requisites have been made at the previous stage.

4.2.1 Shader selection

Slang shaders are compiled before creating the pipeline associated to a view, because the compiled program must be included in the pipeline creation info in the form of SPIR-V code. A shader instance is determined by the parameters of the view it belongs to; the view type type determines shader source code file to use, while the view type and data domain set the names of the stage entry points and generate specializations that adapt the shader code to said parameters. Additional user-defined specializations are also added at this stage, whose meaning varies according to the view type: Marker specializations define different shapes, while texture specializations change the outputted colors, such as sampling only one texture channel or converting the result to grayscale.

4.2.2 Shader code

The following is a description of the various shaders implemented for the library, divided into the various supported data types.

Markers A geometry shader pass converts a single point coordinate into four vertices of a quad centered at that point. For 2D views, antialiased markers [23] were used for fragment coloring. The various marker shapes are implemented as specializations, so their name can be added to the specializations option field for the correspondent view. For 3D markers, front-facing billboards are generated and shaded through basic raycasting, creating the illusion of a sphere with volume.

Edges Edges are currently drawn through inbuilt line primitives, resulting in aliased lines. Edge endpoints are stored in an index buffer whose contents point to a vertex buffer containing the coordinates. Due to this, edge views also require a marker view previously created, even if that view is not actually used (it can be hidden through the `visible` option in 4.2).

Textures Can be distinguished between linear (mapped to a raw pointer) and opaque textures (mapped to a CUDA mipmap object). As both resources are mapped to a `VkImage` from the Vulkan side, shader source code is the same.

Voxels Voxel shading uses the same technique used in [25]. Similar to markers, a geometry shader converts the voxel center into a 3D cube, generating a maximum of three camera-facing faces instead of the full six.

In addition to the above modes, it is possible to load and use externally generated shaders for a view, as long as the external shader code adjusts to the data structures set by the library. These shaders can be used to implement different behavior than the alternatives offered by the library, and have a reduced load time as they do not pass through the Slang compilation flow. External shaders are enabled through view creation parameters, by passing the SPIR-V source path and the shader stage contained in that file.

4.2.3 Display

Display must be explicitly started from user code through the `display` and `displayAsync` functions, which show the display window and the contents of any *visible* views (those with the `display` option active). The first function performs synchronization automatically as per algorithm 1 for a number of iterations passed as input, but requires the compute function to be encapsulated in a callable function and is fit only for simple interop cases that wrap the kernel between synchronization calls and do not need additional steps in between. The "async" call does not have these limitations, but does require explicit synchronization calls in CUDA code. The `prepareWindow` and `updateWindow` functions perform the compute wait and signal operations described in figure 3.3 respectively. The wait function has a default timeout of one second in case it is called in wrong order or more than once; otherwise, it would cause freezes at runtime.

4.2.4 User interface

Parameters like colors and camera values may be scripted and modified from the API, but it may be desirable to get visual feedback in real time. The library provides standard mouse motions to handle camera zooming, panning and rotating. A minimal graphical interface is also supplied using the ImGui library, showing the list of added views, their parameters and modifiable values, including color selectors. As a generic interface cannot adjust to specific usage patterns of each client applications, the library also defines a `setGuiCallback` function to embed GUI elements. The callback is called before the frame render step and should consist of only ImGui API calls to avoid slowdown. Through this function, users can set custom interface elements to interact with application code without needing any GUI setup.

4.3 Code organization

Library code is organized through the following source files:

1. `cudaview.cpp`: Contains the main engine code, which handles initialization, shutdown, rendering and synchronization work. Contains calls from most of the other modules.

2. `vulkan_device.cpp`: Defines the `VulkanDevice` class as abstraction for the physical (the actual GPU being used) and logical (device instance with requested features) device Vulkan objects. This class handles creation and cleanup of Vulkan objects such as buffers, images, semaphores and fences.

3. `interop_device.cpp`: Inherits from `VulkanDevice`, and is the class actually being used by the engine. Manages view creation and interop resource creation by using functions from the base device class.

4. `vk_pipeline.cpp`: Contains the slang shader compile behavior and posterior pipeline creation, which is called each time a new view is created. All shader related code is contained here.

5. `vk_initializers.cpp`: Implements initializer functions for most Vulkan structures used by the engine, initialized with sensible defaults and matching common use cases by the library code.

6. `framelimit.cpp`: Contains routines to handle FPS target values.

7. `validation.cpp`: Contains validation, error checking and debug functions for CUDA and Vulkan calls. Vulkan validation layer support is managed here. The corresponding header can also be included in user code for decorated CUDA error checking.

4.4 Usage

The engine class is the main library entrypoint and the most relevant structure in the design, as it is the only one that allows interacting with the interop resources and the corresponding visualization. The `cudaview.hpp` header contains all the user-facing code necessary to produce interop visualizations, so it is the only one that must be included, but there are additional functionalities such as CUDA and Vulkan error checking in the `validation.hpp` header that can be used in user code if needed.

Library integration into existing CUDA code (the client code) can be divided into three tasks: engine initialization, engine creation and interop synchronization. The engine instance can be created with minimal configuration, providing only window sizes, or by manually specifying them and passing the settings structure to the creation function, as shown on listing 4.3. The initialization call can be put anywhere in client code as long as it is placed before view creation. However, if the client code sets the CUDA device manually through `cudaSetDevice()` with a given device ID, then the engine options must contain the same value before initializing the engine, or it may override the chosen device with the automatic selection procedure as shown in section 4.1.2. In the same way, calling `cudaSetDevice` after engine initialization will likely result in runtime failure.

Source Code 4.3: Engine initialization flow.

```cpp
#include "cudaview/cudaview.hpp" // main library header

// Define kernels or other compute work here

int main() {
    // COMPUTE: Perform experiment setup; define environment variables, allocate
    ↪ non-interop resources (cudaMalloc, etc.)
    int N = ELEMENT_COUNT;

    // Pointers to interop resources should remain unallocated
    int *d_data = nullptr;

    // Optional: setup engine options at startup
    ViewerOptions options;
    options.window_size   = {width,height}; // Starting window size
    options.show_metrics  = false; // Show metrics window in GUI
    options.report_period = 0; // Print usage stats every N seconds
    options.enable_sync   = true; // Use interop synchronization
    options.present       = PresentOptions::Immediate; // Display frames ASAP
    options.target_fps    = 120; // Requires immediate mode

    // Initialize the engine
    CudaviewEngine engine;
    engine.init(options); // Alternatively, engine.init(width, height) initializes
    ↪ the engine with defaults
    ...
}
```

The view creation function `createView` replaces the allocation call for the CUDA resource that will be displayed. Similar to `CudaviewEngine::init`, it uses a parameter struct `ViewParams` and a CUDA resource address as inputs. Fields like the data domain, data type and resource type depend on the desired visualization, but the element count and size can be derived from the existing code. For example, for linear buffers, using the replaced `cudaMalloc` function as base, `element_count` is the number of elements in the buffer and `element_size` is the byte size of each element typically computed via `sizeof`. Likewise for opaque textures, these values can be obtained from the CUDA texture description structure. When successful, the function returns a view handle and modifies the input resource address to point to the imported interop resource. Receiving the view handle is necessary for interacting with it through the library API after creation, but can be omitted if it will not be modified that way.

Listing 4.4 shows a code sample creating a 3D structured space Voxel view with integer elements arranged in a cubic grid. The buffer for the implicit Voxel positions in structured space is automatically allocated by the function. The replaced allocation call is shown at the end of the listing.

Source Code 4.4: View creation flow.

```
// Create a voxels view
ViewParams params;
params.element_count = N*N*N;
params.element_size = sizeof(int);
params.data_domain = DataDomain::Domain3D;
params.resource_type = ResourceType::StructuredBuffer;
params.view_type = ViewType::Voxels;
params.options.color = {0,0,1,1}; // Draw blue voxels by default
// The following call initializes
InteropView *v1 = engine.createView((void**)&d_data, params);
// The above is equivalent to the following regular CUDA call:
// cudaMalloc((void**)&d_data, sizeof(int) * N*N*N);
```

Once the views are created the mapped arrays behave just as if they were created from CUDA, so they can be operated with functions like `cudaMemset`, being a destination for `cudaMemcpy` or initialization through a kernel call. Listing 4.5 shows display with explicit synchronization using a created view. Once opened, the display window will not close even if compute work ends, so it must be closed through the `exit` function or by the window interface. This does not release any engine resources, so compute work using the interop resources will still work after calling this function. The engine will clean up all its resources once its handle goes out of scope.

Source Code 4.5: Interop synchronization flow.

```
engine.displayAsync(); // Starts window display with visualization from all view
↪  resources
for(int i=0; i < NUMBER_STEPS; ++i){
    engine.prepareWindow(); // Notify the engine that compute started

    // Actual compute work with the interop array
    compute_kernel<<<grid, block>>>(d_data, N);
    cudaDeviceSynchronize();

    v1->toggleVisibility(); // Can change view parameters in real time through the
    ↪  API
    engine.updateWindow(); // Notify the engine that compute finished
}
engine.exit(); // Closes the window

// COMPUTE: Experiment cleanup; deallocate non-interop resources (cudaFree, etc.)
```

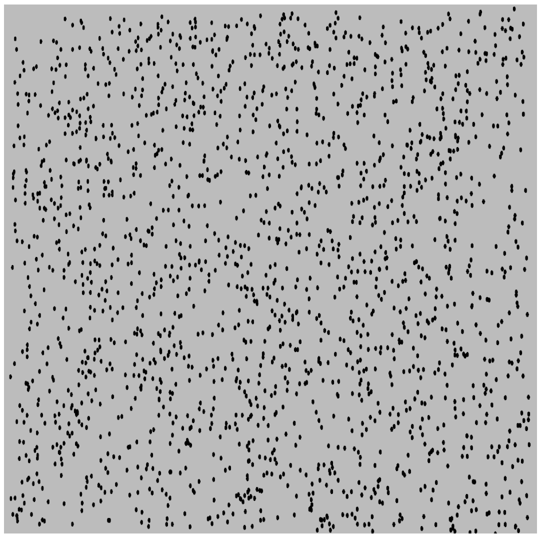
4.5 Example uses

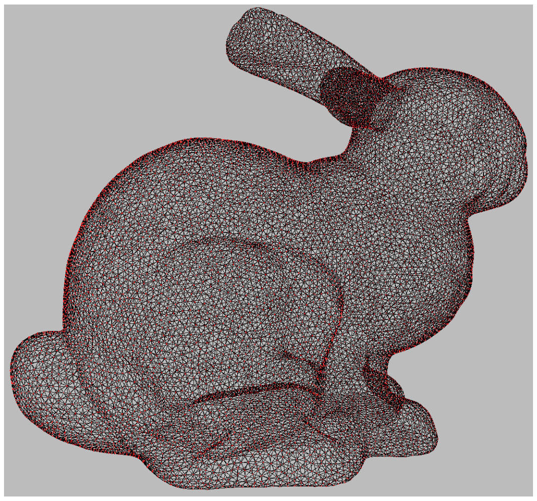
A number of sample programs were developed to validate and check library behaviour for the different supported data types, shown on figure 4.1. For markers, the sample consists of the same set of randomly moving particles, but this time through 2D space. Unlike the experiment, this sample uses the simplified sync mode, so it uses a GPU work function along the number of iterations it runs instead of having to manage synchronization in compute code.

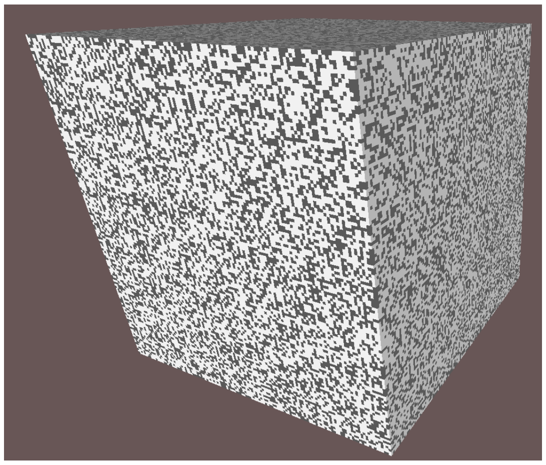
Edge visualization is demonstrated with a simple `.off` mesh model viewer that does not perform any compute work, so there is no synchronization either. The edge view reads endpoint coordinates from a point cloud view previously added, where the point coordinates are the mesh vertices.

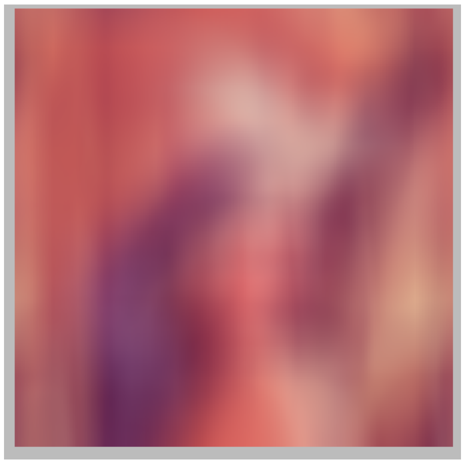
It is also possible to mix and match other view types as long as the underlying device resources do not alias (partial or full overlap of their memory regions). The sample for linear textures computes a distance field with the parallel Jump Flood Algorithm [20] (JFA) over a set of points moving randomly throughout 2D space. The distance field is recalculated after each point position update kernel and mapped to a linear texture view, as the implemented JFA kernel works over linear memory.

The voxel visualization consists of a cellular automata over a regular (structured) square grid. The compute workflow uses a ping-pong buffer scheme where the input of a step is used as the output of the next one in an interchangeable fashion. Allocations for the two buffers are replaced by voxel views, but only the buffer containing the result is shown (this is set through the view options with the `display` field). This way, at *ping* (the original state) the view holding the kernel input is shown and the auxiliary buffer is hidden, but after *pong* (after launching the kernel) the display flag is toggled for both views so that the outdated state is hidden and the kernel output is shown.


(a) Markers


(b) Edges


(c) Voxels


(d) Opaque texture

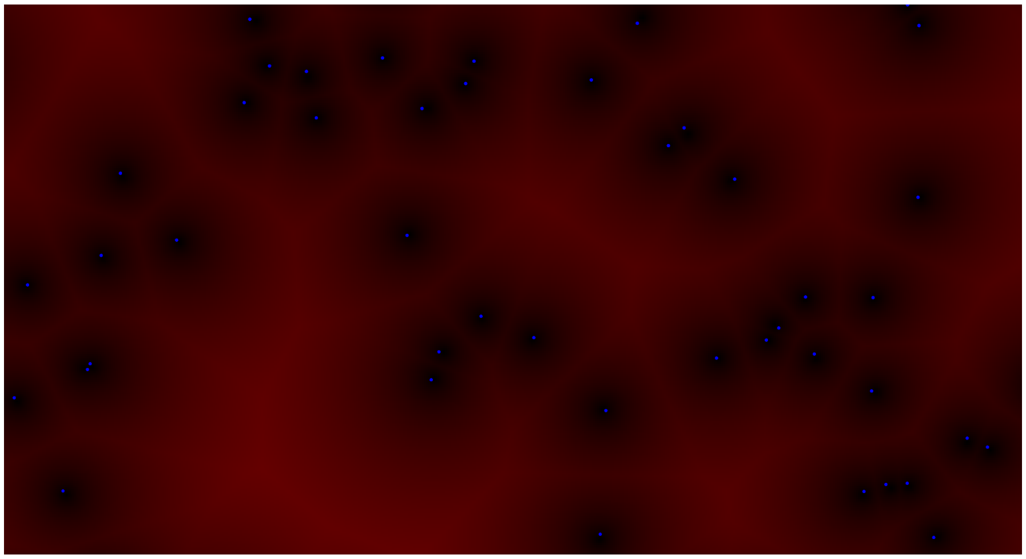

(e) Linear texture

Figure 4.1: Visualizations for all data types provided by the library.

For opaque textures, the Vulkan interop code from the CUDA samples was ported over to the library. This code computes a box filter with a periodically varying radius over a mipmapped image. Only the kernels were copied over, so all rendering and interop work is done with the library and the original visualization can be reproduced.

Chapter 5

Results

Experimental results are divided into three types. The first family measures library (interop and rendering) performance in terms of time, FPS, GPU power and memory efficiency. These results use interop synchronization enabled, which produces idle time for both compute and rendering as the timeline semaphore tells each workflow to wait and continue when appropriate. It is possible to turn off synchronization to eliminate this performance hit at the cost of missed iteration displays or graphical artifacts, which could be beneficial for use cases where those drawbacks are not significant or can be ignored. The second result type compares visualization performance with the proposed synchronization settings against turning off all interop synchronization. The final result compares Mìmir against alternative visualization backends, memory layouts and existing libraries, such as PCL [24]. An additional result is presented, consisting of a library use case implemented over real code, which adds mesh visualization for a GPU Delaunay mesh edge-flip algorithm.

5.1 Experimental setup

The experiment consists in measuring the performance of a CUDA program that simulates a set of points moving randomly through 3D space inside a bounded square box. No interactions of any kind between particles (such as collisions or forces) are considered. Both the starting positions and the random movement are generated with the cuRAND library, and the positions are updated for a number of integration steps, which are ran as iterations. Starting positions are uniformly distributed through the simulation extent and their displacements are obtained from a normal distribution, so they can be positive or negative in any of the three axes. Updated positions are then clamped to the extent before writing them to global memory. The routines for initializing the simulation (RNG[1] state and initial point positions) and the integration steps are implemented as separate kernels, and only the main work loop doing the position updates is considered for measurements.

[1]Random Number Generator.

Source Code 5.1: Experiment interop setup.

```
ViewerOptions options;
options.window_size = {width,height}; // Starting window size
options.enable_sync = enable_sync;
options.present     = present_mode;
options.target_fps  = target_fps;
CudaviewEngine engine;
engine.init(options);

ViewParams params;
params.element_count = point_count;
params.element_size  = sizeof(float3);
params.extent        = {200, 200, 200};
params.data_domain   = DataDomain::Domain3D;
params.domain_type   = DomainType::Unstructured;
params.resource_type = ResourceType::Buffer;
params.view_type     = ViewType::Markers;
engine.createView((void**)&d_coords, params);
```

A view for this program is added following the design described in chapter 3 using the code in listing 5.1, producing a setup such as the one shown in figure 5.1. The view represents an unstructured buffer of sphere markers, associated to the positions array of type `float3` represented over a $3D$ domain with an extent of $200\times200\times200$. Camera settings remains fixed at a zoom level enough to fit all markers inside the view frustum. Display uses immediate mode with a FPS target T for each run, using values $T \in \{30, 60, 144, 240, 0\}$, where $T = 0$ means unlimited frame rate (no target FPS value). The framebuffer sizes used were chosen to match commonly used display resolutions, such as FHD (1920×1080), QHD (2560×1440) and UHD (3840×2160). Each run explores a combination of the above values and the input sizes, except for the number of integration steps which was left constant to 1000 iterations. All other visualization parameters (marker color, marker size, lightning) remain constant throughout the experiment run and across all runs. Additionally, the base execution running only the CUDA kernels (without any display) is also evaluated and annotated as the "none" mode in the plot legends.

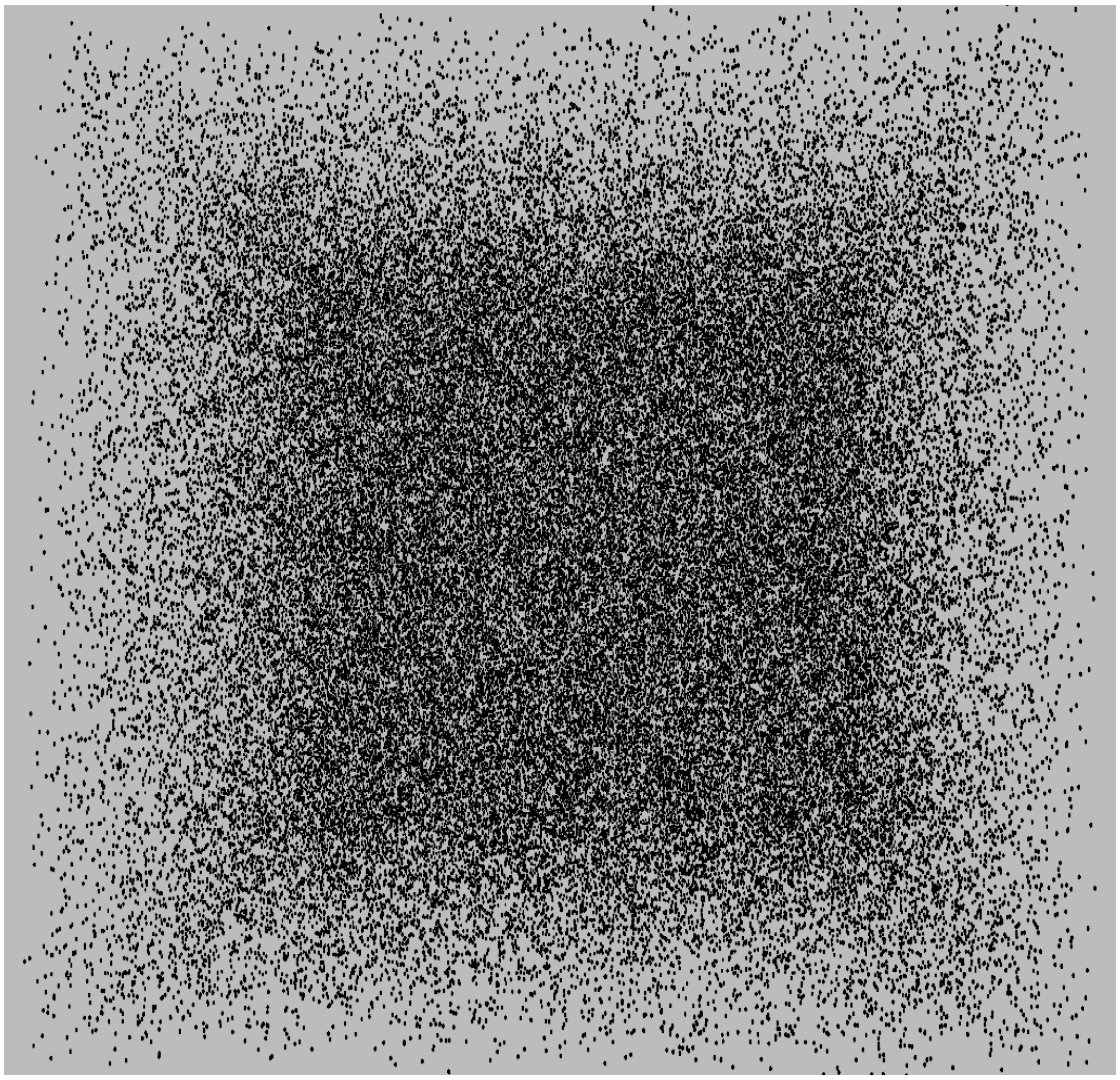


Figure 5.1: Example visualization of an experiment run for $N = 100K$.

The experiment considered the following metrics, measured for every run:

- Rolling average frame rate for the last 240 frames.
- GPU time taken to run all work iterations, as measured by CUDA.
- Average GPU power and total energy, as measured by CUDA.
- Vulkan device memory usage and budget, as measured by Vulkan extension.
- Used and free GPU memory, as measured by CUDA.

The NVML library performs the measurements from the CUDA side, while the Vulkan side uses the `VK_EXT_memory_budget` extension. This extension allows querying device memory state at the moment the corresponding function is called, returning statistics of all heaps managed by Vulkan. Each heap has a property bit field that makes possible to determine if a heap is allocated on host or device memory, based on its flags. For this experiment, GPU memory usage was computed as the sum of allocations across all device heaps, that is, those with the device memory bit set. The same process is repeated for the device budget, which is the available memory before further Vulkan allocations fail or degrade performance. In contrast, NVML reports used and free memory for all VRAM, including processes not belonging to the experiment.

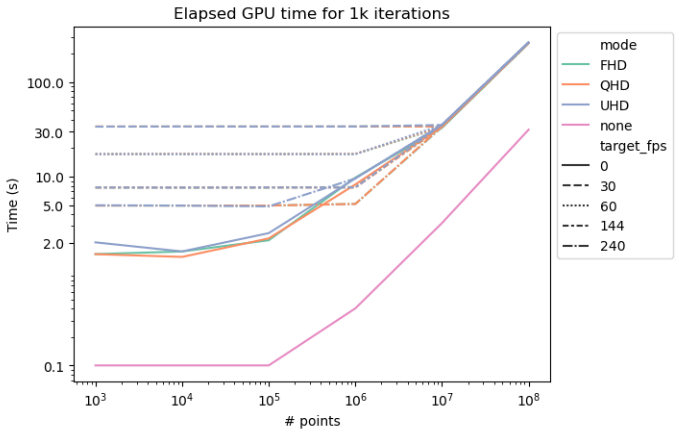
5.2 Testing environment

The experiment was performed in a workstation platform with the following specs and compile settings:

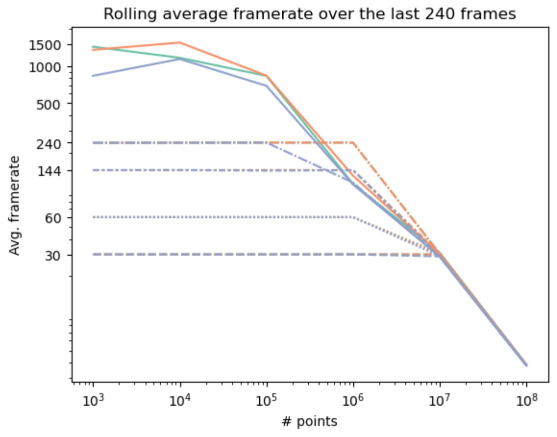
- *Host:* AMD Ryzen 7 3700X CPU / 16GB RAM
- *Device:* GeForce RTX 2070 SUPER (8GB VRAM)
- *C++:* `gcc 11.3`, `C++17` standard
- *CUDA:* Version `12.2`, driver version `535.54.03`
- *Vulkan:* Version `1.3.236`
- *Display:* size 24"; resolution 1920×1080; refresh rate 144 FPS

The simulation parameters are the amount of simulated points, the number of iterations and the RNG seed value, while the visualization parameters are the resolution of the display framebuffer in $W \times H$ format, the target FPS and whether to use interop synchronization or not (named as "de-sync" mode). All runs were executed at fullscreen for exclusive display mode. For runs that require QHD and UHD resolutions, the `nvidia-settings` utility was used to change the viewport to the desired size.

5.3 Experiment results



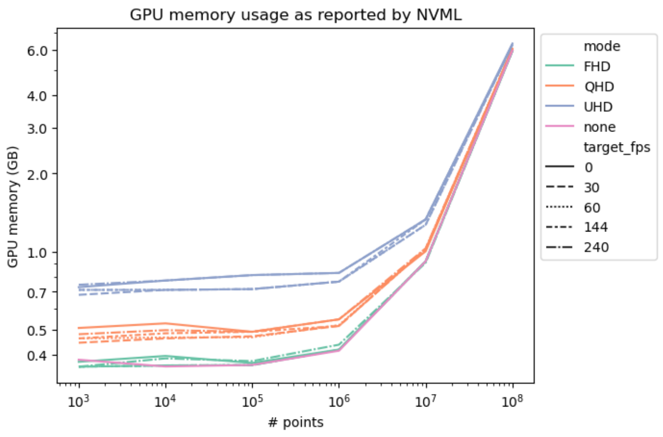
(a) Elapsed GPU time.



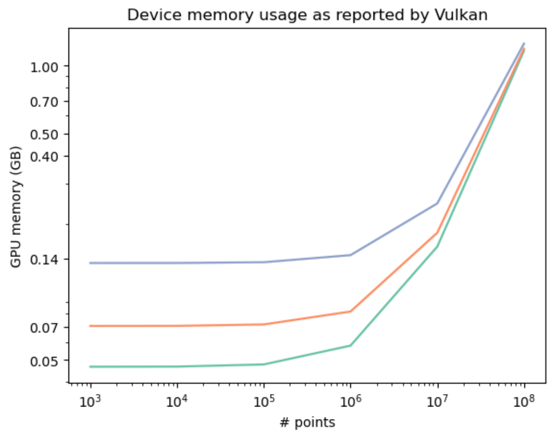
(b) Average frame rate.

Figure 5.2: Performance results for various combinations of framebuffer size and target FPS.

Figure 5.2 shows performance results for different combinations of configurations; the FPS for the non-rendering base case is obviously zero. Frame rate limits are shown to be correctly applied when set, and FPS degradation starts occurring starting at $N = 10^6$ for the largest FPS target. This is consistent with the total GPU time (rendering and compute), which also grows sharply starting from that value. For $N = 10^8$ rendering performance is no longer responsive. Elapsed time increases with lower FPS targets because compute is being stalled in order to match the rendering pace being forcibly delayed.



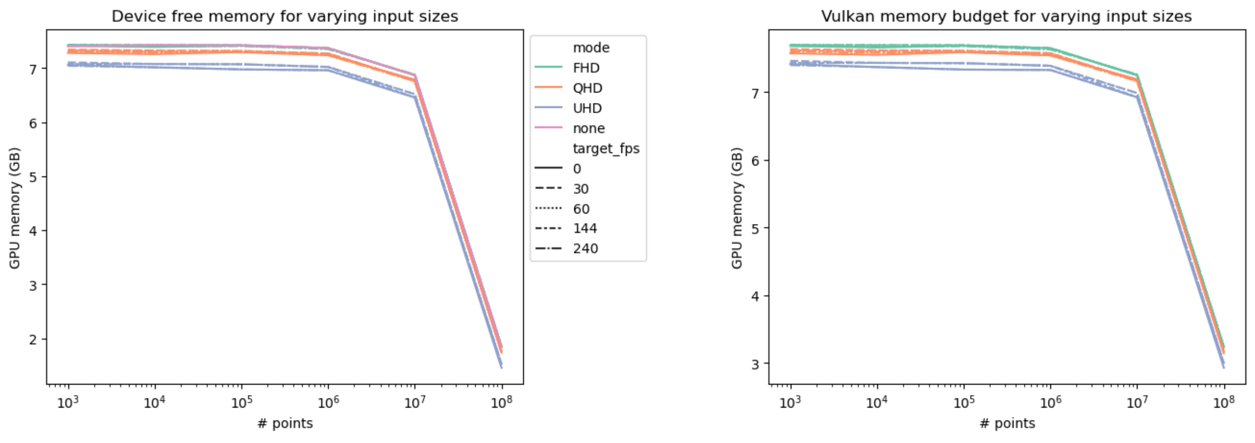
(a) Using NVML.



(b) Using `VK_EXT_memory_budget`.

Figure 5.3: Device memory usage across different experiment runs, measured from CUDA and Vulkan APIs.

Figure 5.3 shows device memory usage from Vulkan context and from the whole GPU. As expected, Vulkan memory use does not depend on target FPS, but it does depend on framebuffer size and input size, because interop memory allocation is handled by the Vulkan API. Base memory usage in Vulkan is zero because allocation in that case is via `cudaMalloc`, but it gets correctly measured through NVML. It is worth noting that NVML does measure differences in memory usage between FPS targets, and the ordering is as expected. That is, base uses the least memory, then the resolutions starting from the lowest, and the resolutions themselves are ordered by the target FPS from least to most (target 0 is unlimited FPS). Memory usage is still different across FPS targets even at $N = 10^7$, where figure 5.2 shows all cases are displaying at approximately 24 FPS. Differences in Vulkan memory usage remain constant across the three resolutions; UHD memory is roughly double that of QHD, which in turn is about double that of FHD. This is consistent with memory usage depending majorly on framebuffer size.

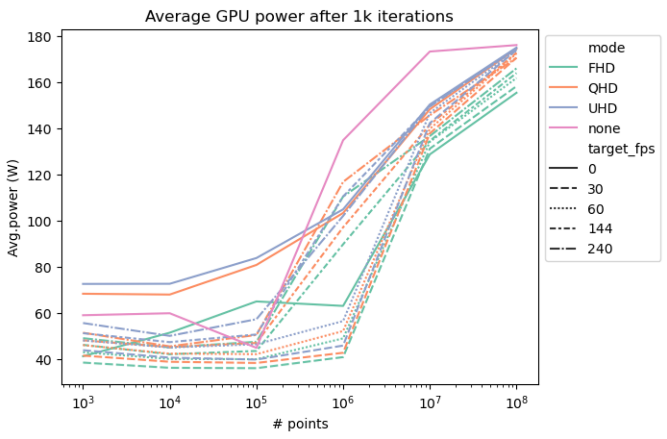


(a) Using NVML.

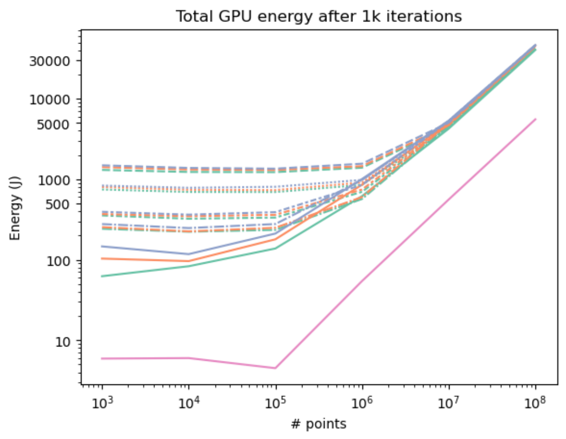
(b) Using `VK_EXT_memory_budget`.

Figure 5.4: Available device memory across different experiment runs, measured from CUDA and Vulkan APIs.

Similar to above, figure 5.4 shows Vulkan budget and total GPU free memory, measured through the Vulkan device budget extension and NVML respectively. Memory usage measured from Vulkan follows the same curve as the measurements from CUDA, with a difference of $250 - 350$ MB between budget (Vulkan) and free memory. This is because they do not necessarily measure the same thing; Vulkan budget represents the total free memory across all heaps, while NVML measures the total memory from VRAM.



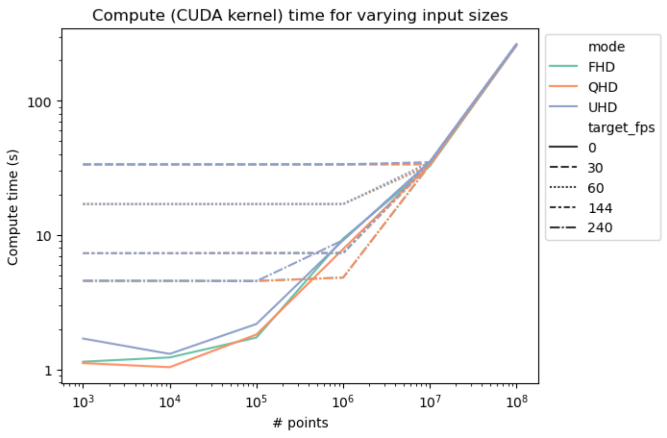
(a) Average power.



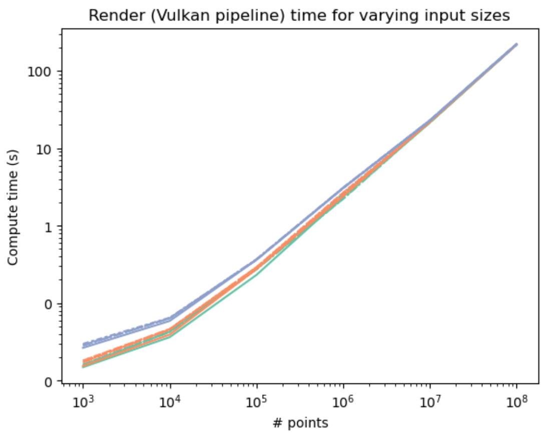
(b) Total energy.

Figure 5.5: Device energy metrics for concurrent compute and graphics work.

Figure 5.5 shows GPU average power after all iterations. As expected, total energy is the lowest without display as there are no Vulkan work being done. Average power in this mode is the highest for larger input sizes, which is consistent with the lower time needed for completing the execution. Higher resolution and higher FPS targets contribute to greater power usage (here FPS target zero means the highest possible FPS value). Energy consumption is dominated by input size when large enough, however.



(a) Compute time.



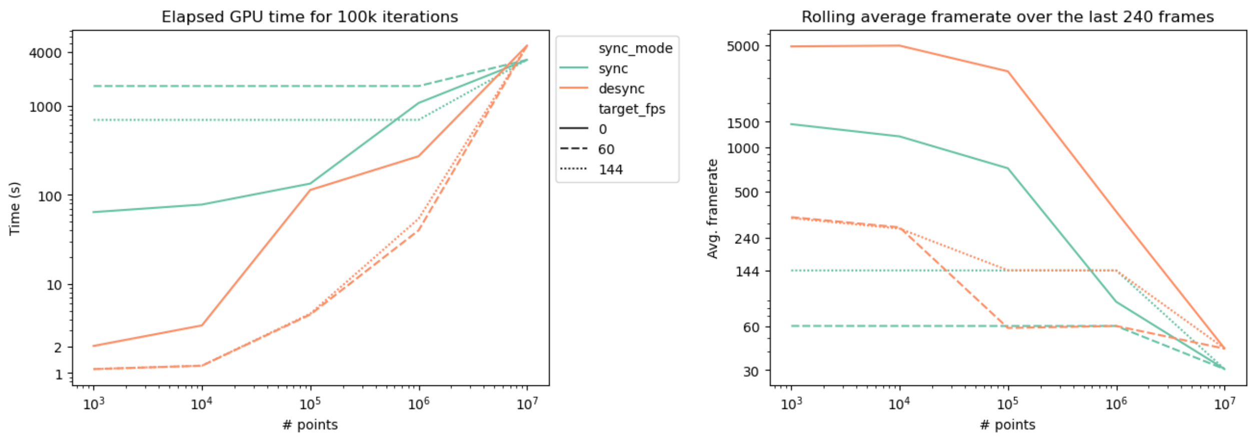
(b) Pipeline time.

Figure 5.6: De-aggregated device time between total compute and graphics pipeline processes.

Figure 5.6 shows the total time separated between CUDA kernel and Vulkan pipeline times respectively. The compute time curve follows the same pattern as the total time results shown on figure 5.2, while total pipeline time grows linear to input size.

5.3.1 Synchronization modes

This experiment compares between the synchronous interop mode (the default) and de-synchronized execution, where compute and rendering run independently. Resolution was fixed at 1920×1080 and the FPS target values $60, 144, \infty$ were considered. The amount of iterations is raised to 10000 in order to give the visualizer enough time to process and present frames before compute work ends in de-sync mode.

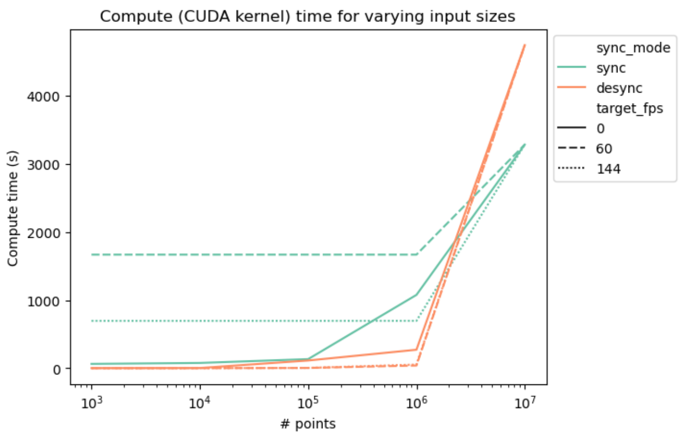


(a) Elapsed GPU time. (b) Average frame rate.

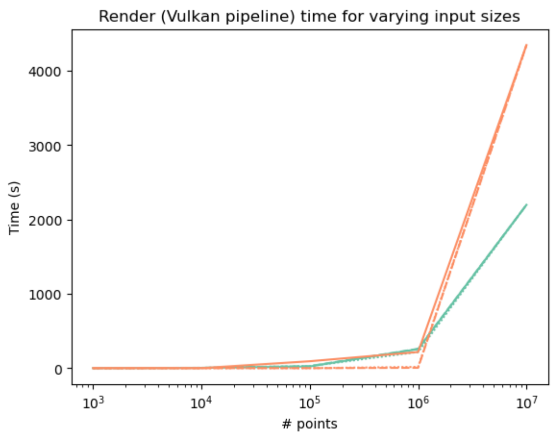
Figure 5.7: Performance results for different combinations of synchronization mode and target FPS.

Following the same structure as with the previous experiment, figure 5.7 shows performance results for both modes. Unlimited de-sync mode (FPS target set to zero) starts an order of magnitude faster than for the smallest input size, but the gap decreases with greater N and the difference is reversed at $N = 10^7$. The de-sync execution for $N = 10^8$ clocked at more than 10 hours in wall time before being cancelled. FPS target values are honored in sync mode as expected, but are broken for smaller N as there are not enough displayed frames before the compute workflow finishes executing all kernel iterations. Setting a limit value still produces an impact in this mode because only the rendering thread is affected by it, but the compute thread runs uninterrupted.

Even though compute and rendering are running fully independent from each other, they will compete for processing power when the GPU is at close to full utilization. For lower input sizes the device has resources to spare, but frame rate lowers as the compute kernel becomes more demanding. Near $N = 10^6$ the frame rates for the unlimited FPS settings become close to the 144 target, meaning the sleep time for the FPS-bounded case is near zero. The GPU is close to reach full utilization at that point, so additional rendering work is likely to throttle compute work or vice-versa.



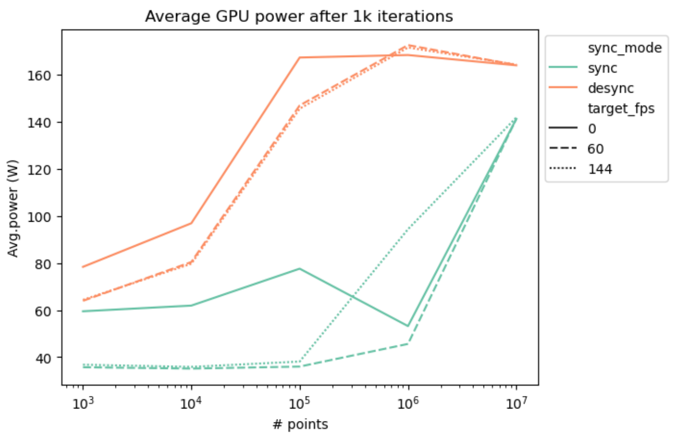
(a) Total kernel time.



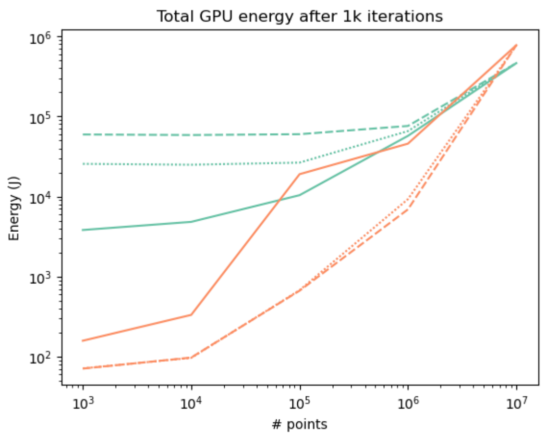
(b) Total pipeline time.

Figure 5.8: De-aggregated times for synchronous and de-synchronized modes.

Figure 5.8 shows separate kernel and pipeline total times for the execution modes. Like figure 5.7 for aggregated total time, both times are much lower for de-sync mode at smaller input but the trend is reversed before reaching 10^7. It is worth noting that in both modes the experiment state (input size, random seed, point size, visual effects) are exactly the same, with the only difference being the presence/absence of the interop synchronization mechanism.



(a) Average power.



(b) Total energy.

Figure 5.9: Device energy metrics different synchronization modes.

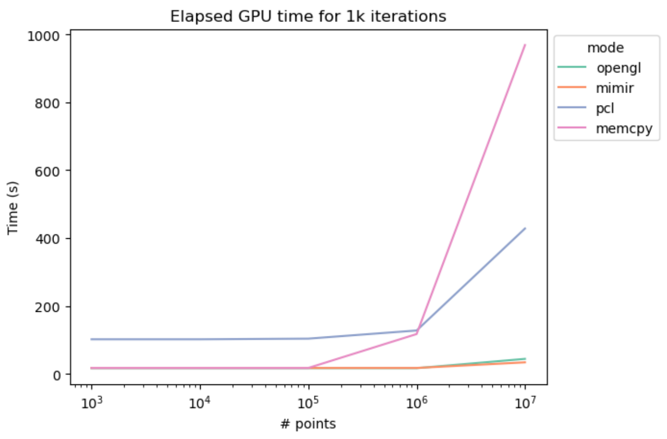
Figure 5.9 shows energy usage between sync and de-sync modes. Average power in de-sync mode is consistently higher than sync, and unlimited FPS settings use more power on average than fixed values within their respective synchronization modes. Meanwhile, total energy for uncapped de-sync execution is higher than FPS limited de-sync runs, but the opposite happens for unlimited sync runs against their limited alternatives. Disabling synchronization produces a noticeable amount of average power usage that becomes greater for lower FPS targets, where the largest difference is at 60 FPS and the lowest is for unlimited FPS. This is due to compute work running at full speed with no synchronization wait, unlike rendering work where low FPS targets lead to relatively longer thread sleep times.

5.3.2 Alternative execution modes

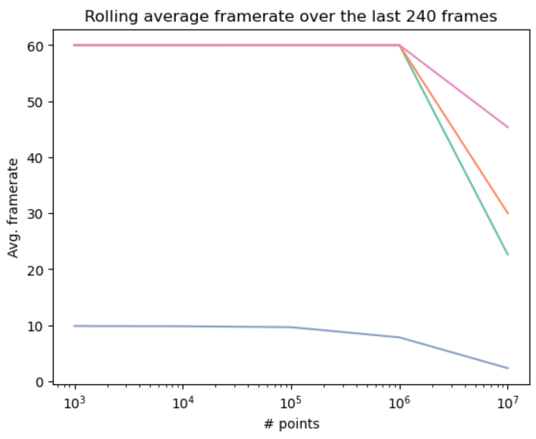
The final experiment consists in comparing alternative methods of generating a dynamic point cloud visualization. The following alternatives were considered:

1. The Mìmir library used in the previous experiments, running in sync mode and VSync enabled.
2. Same as the above, but performing all compute work in host and using `cudaMemcpy` to move the result to an interop buffer (noted as memcpy in the figure legends).
3. An OpenGL/CUDA interop implementation of the Mìmir flow for Markers.
4. Copying the kernel results to host and display them with PCL.

The OpenGL interop was implemented to work as close as possible as the configuration used by its Vulkan counterpart for this experiment, and GLSL shaders equivalent to the Slang shaders from Mìmir were used. The point cloud using PCL was rendered as squares with the default color and size configuration. The host compute mode uses a parallel OpenMP loop to iterate through the pointers and a PCG Random Number Generator [17] to generate the random displacements at a faster pace than the default `C++` Mersenne Twister RNG. The remaining alternatives use the same CUDA point cloud kernel and launch parameters. However, as the PCL 3D point cloud structure internally uses 4 floats per point, the kernel for that experiment was modified to read and write to a `float4` coordinates array, unlike all other kernels that work with `float3` elements.



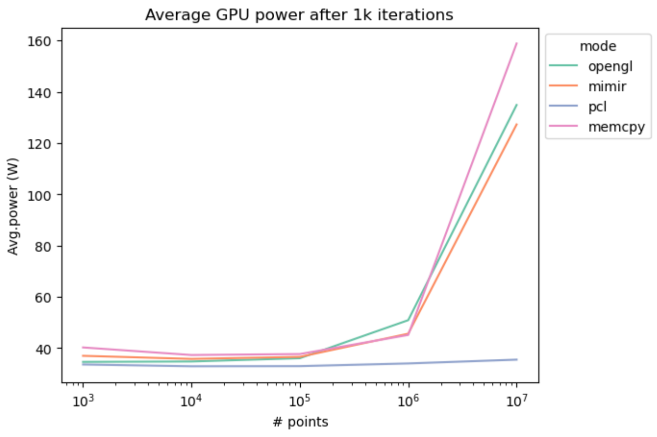
(a) Elapsed time.



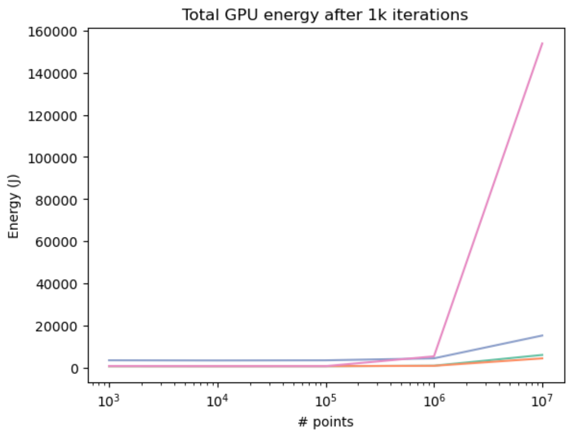
(b) Average frame rate.

Figure 5.10: Performance results for various input sizes and execution modes.

Figure 5.10 shows time and frame rate comparing the four alternatives. Vulkan and OpenGL show no differences until the highest input size, where Mìmir (Vulkan) has slightly better performance. Both average frame rates are lower than the memcpy variant for $N = 10^7$, as the device is not being used for compute work in this case. The PCL visualization has the lowest frame rate because of the multiple memory transfers involved; at each iteration, the GPU result data is copied to the PCL cloud structure and refreshed so that the visualization can display the changes. This involves an additional host-to-device memory transfer performed implicitly by the VTK backend to display the updated point cloud state.



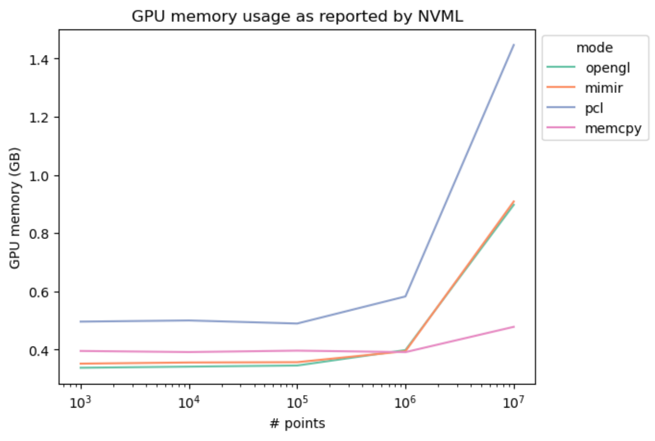
(a) Average power.



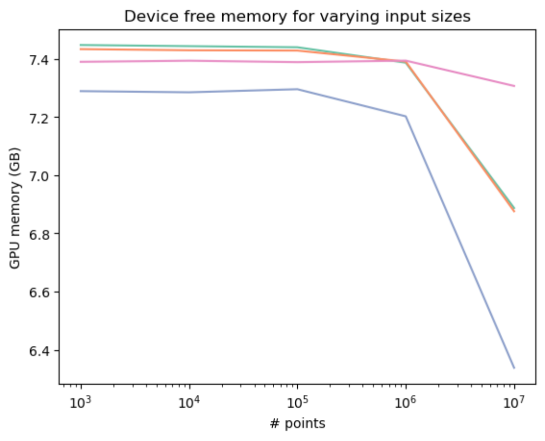
(b) Total energy.

Figure 5.11: Device energy metrics for various input sizes and execution modes.

Figure 5.11 shows power usage, where once again the Vulkan backend is slightly more efficient than OpenGL at higher input sizes. The memcpy variant has the most GPU energy utilization, as the Mìmir visualization has to wait for the slower host compute work to finish and receive the updated data from a likewise expensive `cudaMemcpy` call before processing the next frame, so each frame is being shown for more time compared to other alternatives.



(a) Used memory.



(b) Free memory.

Figure 5.12: Device memory usage for various input sizes and execution modes.

Figure 5.12 shows memory usage, where this the OpenGL backend is slightly more efficient than Vulkan. At the largest input size both of them use more memory than the memcpy variant due to the increasing size of the `curandState` array that is allocated along with the point coordinates. The PCL approach uses consistently more memory than the others, due to the different VTK backend and the need to use `float4` elements instead of `float3` for storing the point coordinates.

5.4 Study case: Tiuque

The codes presented in the previous section were made to fit the designed visualization modes and settings, so they may not represent real use cases. An additional experiment involves adding the Mìmir library to an existing codebase and use it to generate an application that generates at least one of the provided visualization types. One such application is the cleap library, which implements GPU edge-flipping [16] for triangle meshes that can be read from and written in `.off` format. The tiuque app provides a cleap frontend, adding support for loading, processing and saving meshes, with interface elements for selecting whether to perform all edge flip iterations at once or by one step at a time (educational mode).

Source Code 5.2: Tiuque interop setup.

```
ViewParams vert; // Vertices view (invisible)
vert.element_count   = cleap_get_vertex_count(m);
vert.element_size    = sizeof(float4);
vert.data_domain     = DataDomain::Domain3D;
vert.domain_type     = DomainType::Unstructured;
vert.resource_type   = ResourceType::Buffer;
vert.view_type       = ViewType::Points;
vert.options.visible = false;
engine.createView((void**)&m->dm->d_vbo_v, vert);

ViewParams tri; // Triangles (edges) view
tri.element_count  = cleap_get_face_count(m);
tri.element_size   = sizeof(uint3);
tri.data_domain    = DataDomain::Domain3D;
tri.domain_type    = DomainType::Unstructured;
tri.resource_type  = ResourceType::Buffer;
tri.view_type      = ViewType::Edges;
tri.options.color  = {0.f, 1.f, 0.f, 1.f};
engine.createView((void**)&m->dm->d_eab, tri);
```

The original cleap source code used OpenGL and GLEW for display, and the first (deprecated) interop API for mapping graphics buffers to CUDA arrays. All those functions were removed, and the memory allocations for both the mesh triangles' Element Array Buffer (EAB) and the Vertex Buffer Object (VBO) were replaced by the view creation functions for edges and markers as shown on listing 5.2. The markers view is left hidden because the original library does not display edge vertices.

Source Code 5.3: Tiuque app initialization.

```
void Application::init() {
    engine.setGuiCallback([=,this] {
        if (ImGui::BeginMainMenuBar()) {
            if (ImGui::BeginMenu("File")) {
                if (ImGui::MenuItem("Open Mesh")) {
                    ImGuiFileDialog::Instance()->OpenDialog("LoadFileDialog",
↪  "Load mesh file", ".off", ".");
                }
                if (ImGui::MenuItem("Save")) {
                    on_menu_file_save();
                }
                if (ImGui::MenuItem("Save As..")) {
                    ImGuiFileDialog::Instance()->OpenDialog("SaveFileDialog",
↪  "Save mesh file", ".off", ".");
```

```cpp
                }
                ImGui::Separator();
                if (ImGui::MenuItem("Quit")) {
                    engine.exit();
                }
                ImGui::EndMenu();
            }
            if (ImGui::BeginMenu("Run")) {
                if (ImGui::MenuItem("MDT-2D")) {
                    on_button_delaunay_2d_clicked();
                }
                if (ImGui::MenuItem("MDT-3D")) {
                    on_button_delaunay_3d_clicked();
                }
                ImGui::Separator();
                if (ImGui::BeginMenu("Options")) {
                    ImGui::MenuItem("Toggle Wireframe", "", &toggle_wireframe);
                    ImGui::MenuItem("Educational Mode", "", &educational_mode);
                    ImGui::EndMenu();
                }
                ImGui::EndMenu();
            }
            ImGui::EndMainMenuBar();
        }
        if (ImGuiFileDialog::Instance()->Display("LoadFileDialog")) {
            if (ImGuiFileDialog::Instance()->IsOk()) {
                std::string filename = ImGuiFileDialog::Instance()->GetFilePathName();
                load_mesh(filename.c_str());
            }
            ImGuiFileDialog::Instance()->Close();
        }
        if (ImGuiFileDialog::Instance()->Display("SaveFileDialog")) {
            if (ImGuiFileDialog::Instance()->IsOk()) {
                std::string filename = ImGuiFileDialog::Instance()->GetFilePathName();
                save_mesh(filename.c_str());
            }
            ImGuiFileDialog::Instance()->Close();
        }
    });
    engine.displayAsync();
}
```

The application GUI was replaced with the `setGuiCallback` library function, adding the menu items and options using ImGui API functions. Listing 5.3 shows how GUI elements can be associated to user-defined callbacks such as loading/saving meshes, selecting edge-flip algorithm and algorithm options, where all of them were previously defined in application code. The ImGuiFileDialog[2] add-on library for ImGui is used to handle load/save dialogs and is included with the library. The GUI callback must be set when initializing the application and after engine initialization. It also should be called before starting display, but it can be called after with no problem. The resulting GUI is shown on figure 5.13 along with a 3D mesh visualization.

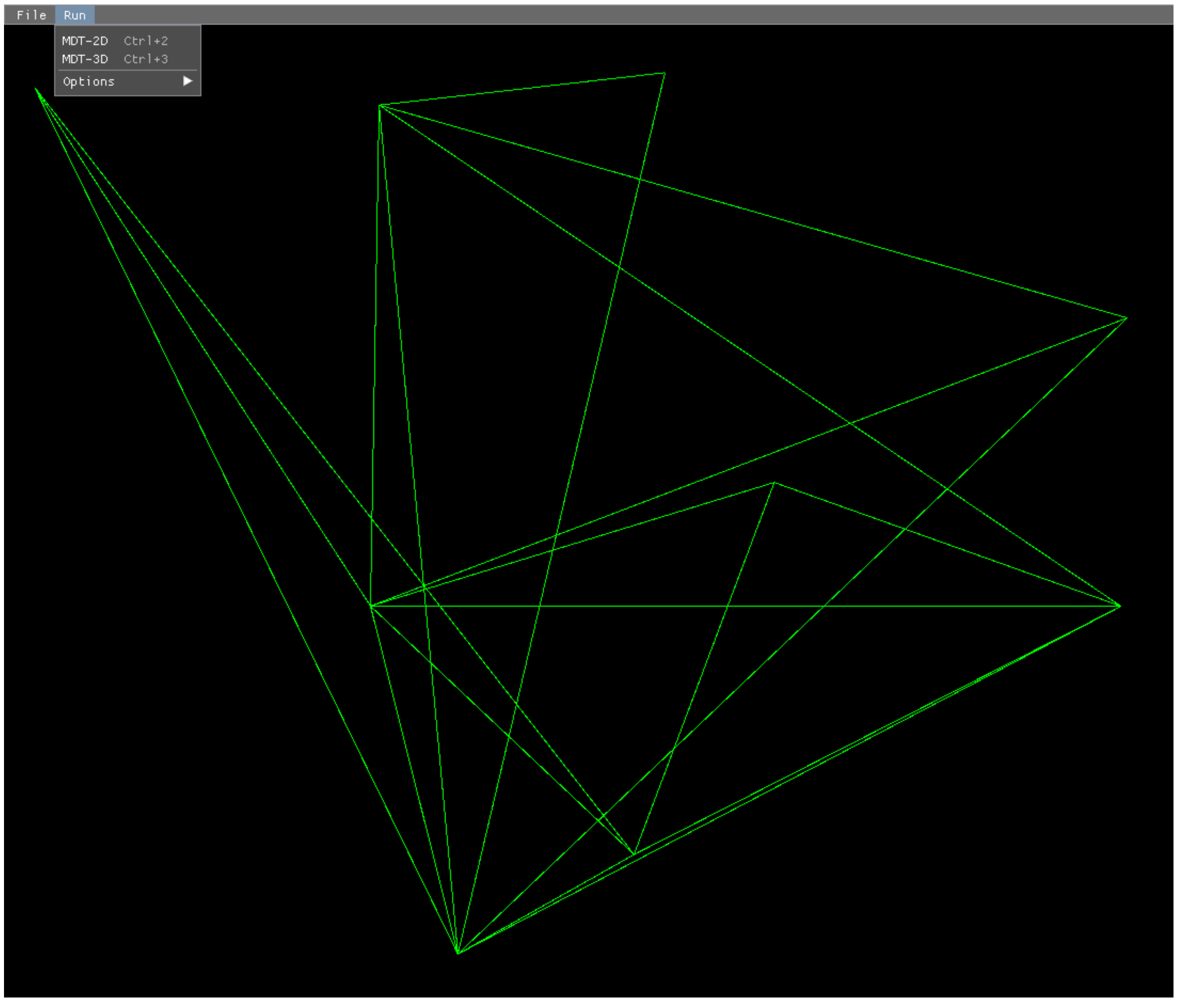


Figure 5.13: Tiuque app implemented with the Mìmir API.

[2]`https://github.com/aiekick/ImGuiFileDialog`. Retrieved 2023-06-23.

5.5 Discussion

The Mìmir library is shown to perform on a typical GPU-only environment as intended, but it also can be used to visualize data obtained from experiments generated in host memory. In this case, performance is determined by the time of host computing and the corresponding memory transfer, rather than the interop setup or library API. While it offers less performance than the original approach, it allows integrating visualizations to host-based experiments with no additional programming effort in comparison. However, experiment programs still needs to include and link with CUDA to perform the memory transfer and interact correctly with the library. Compared to host-based libraries like PCL, Mìmir showed an up to $\times 10$ improvement in total time and average frame rate, favored by its specialized design oriented towards interoperability and no memory duplication or transfer. Likewise, Vulkan used $\times 1.5$ less memory than VTK, which is in part caused by the larger size of the underlying data arrays. However, Mìmir allowed better memory utilization by only allocating the strictly needed amount and alignment, instead of needing to change the kernel fit the visualizer requirements.

When comparing the base (compute only) and visual (compute and graphics) cases it appears to be a noticeable increase in the total GPU time for the same number of iterations. However, this does not necessarily imply a performance penalty, as both the FPS target values and interop synchronization steps introduce wait times on the compute workflow that are not critical for producing visual results. Both features can be turned off when not needed, and results with the desync mode show that display does not produce a considerable overhead over computing (they fast enough so that compute finishes without presenting any frame, in fact). However, there is a noticeable overhead for desync mode against the synchronized variant, which for the experiments is reached after $N = 10^7$. After this point, desync execution becomes slower regardless of the FPS target value, due to compute and rendering flows competing for device utilization. This means visualization performance will degrade whenever full GPU utilization is reached, which likely depends on each algorithm and may be reached at different input sizes. This shows the value of sync mode as a "default" solution with good enough performance, and that does not need additional tuning from the user.

The experiments also evaluate the differences between OpenGL and Vulkan. Both rendering backends perform similar, with a slight advantage for Vulkan that does not merit an API switch by itself. However, OpenGL presents a series restrictions for alternate execution modes supported by Mìmir through Vulkan. The most notorious one is the lack of explicit synchronization calls, as OpenGL performs them automatically at each API call. While this design reduces programming complexity, it also reduces flexibility in deciding when and how to apply synchronization, as it cannot be turned off. This leads to not being able to implement de-sync mode in OpenGL. Another problematic aspect is multithreading, since only one OpenGL context must be active at only one thread at any time, and failure to do so produces runtime errors. Context switches must be called explicitly, and even so it limits OpenGL execution to the thread holding the context. This causes problems for the proposed compute/graphics thread scheme used for interop, since the compute thread has to retrieve the interop mapped pointers at each iteration while the rendering thread is running. As such, the OpenGL implementation used the fully synchronous, single threaded display function described in listing 1 instead of the asynchronous variant supported in Mìmir and used throughout the experiments.

In addition to the above experiments, measurements were made to compare performance between allocating regular device memory with `cudaMalloc` or using interop-mapped memory through `createView`, but without showing display for either execution. Total device time and memory usage were identical for both cases, meaning there is no overhead for using interop memory.

Chapter 6

Conclusions

This work presents a working implementation of a library that allows displaying CUDA programs in real time through CUDA-Vulkan interop, with minimal alteration of existing code. The library is designed to handle common CUDA use patterns and data structures in a concise way by creating views, whose structure parameters are mapped to the Vulkan resources required to produce the requested visualizations. Visualizations for multiple datasets can be created and displayed simultaneously, and the visual aspects of each one can be modified in real time from API functions or through the provided user interface. The GUI can be further extended from user code to generate interface elements specific to the application, such as load/save behavior or more graphical options.

The comparison between visualization experiments using the Mìmir library and the standard host-based alternative through libraries such as PCL allows to validate the stated hypothesis, since the presented implementation achieves a maximum framerate difference of 12× at a 0.6× GPU memory usage for the largest input size considered. The current Mìmir implementation fulfills the objective of enabling visualization for all four proposed use cases and for 2D and 3D modes, through the design of an architecture to allocate and import interop memory and generate the Vulkan resources needed to each view type or use case, as specified by the input parameters for a visualization. Other alternative execution modes were also considered for comparison, such as implementing an equivalent workflow using OpenGL or generating visualization data in host memory and copying to device mapped memory. These Vulkan-based Mìmir library has similar performance to equivalent visualizations in OpenGL, but the differences between APIs produce an impact on design details, such as multi-threaded context and memory ownership between the graphics and compute APIs. Vulkan was found to be a more appropriate choice for the proposed execution scheme, as better multi-thread support allows to further separate rendering and compute workflows.

Vulkan offers and utility for scientific GPU applications, but the learning curve and code complexity may discourage efforts in using it. In this area it also was possible to almost fully encapsulate the Vulkan-side code and the complexity it carries, enabling potential users to generate visualizations that run entirely on device memory without major Vulkan knowledge. Even for cases that do not adjust well to the proposed design, the interop performance analysis performed here can provide a reference for adjusting future interop applications.

The performance results show the library performs well for moderately sized input sets, however, it degrades before reaching 100 million points and beyond, meaning there is still room for improvement before reaching full memory utilization. It is also important to note that both rendering performance and memory usage are not trivial compared to compute use in a GPU, so both tasks will inevitably compete for input sizes large enough. Increasing library performance is key for this, as lower GPU utilization in rendering means more power available for computing. On the same note, having two workloads on the same hardware means the GPU doing the work will play a critical role in performance results. Parameters like VRAM amount allow visualization of larger datasets, while clock speeds, pixel rate and floating point performance may offer better frame rate for the same input size. Differences in architecture also play similar roles, as a newer-generation card may perform better than an older card with better specs for certain operations, if newer driver and API versions allow for it. For this reason, it would be desirable to measure performance for different GPUs in order to have a sample set for estimating performance for similar hardware.

From the experiments it is also shown that the interop synchronization and target FPS options are useful at tuning GPU utilization for a variety of use cases. For example, a low target frame rate can be used to stall the rendering loop and free GPU compute capability to iterative compute-heavy kernels, at the cost of reduced visual smoothness and responsiveness when not strictly needed. Likewise, enabling interop synchronization with fixed frame rates (between 30 and 60, for example) allows controlling the speed of iterative algorithms like simulations without missing simulation steps. The de-synced variant presented the problem of either running too fast or too slow depending on input size, which shows the value in implementing an using interop synchronization for GPU load balance. It is not mandatory to use any of these features to produce visualizations though, and the VSync mode with interop sync enabled can be used as a sensible default before tuning up the parameters.

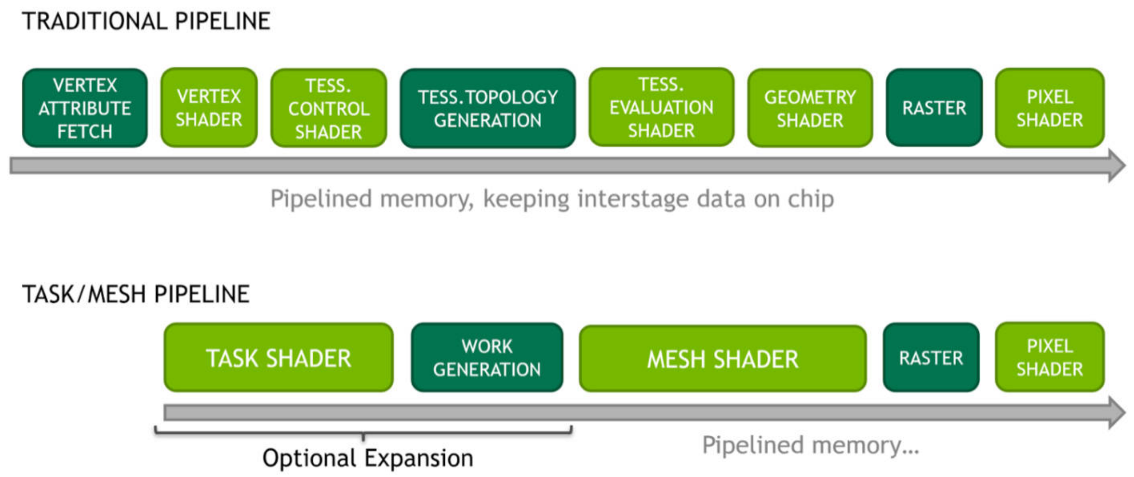


Figure 6.1: Comparison between regular graphics pipeline and mesh shader pipeline.

In terms of future work, there are less common use cases for the library that are currently unexplored and could warrant further optimization. For example, running kernels for multiple views, each running in separate CUDA streams, may lead to unnecessary stalling with the current scheme because all views are sharing one timeline semaphore. It would be worth considering adding *interop streams* with their own CUDA stream and semaphore, but it would also increase complexity by needing to specify which interop stream is being synchronized. In the topic of synchronization, it is also worth exploring additional modes like only using CUDA to Vulkan or Vulkan to CUDA wait/signal operations, which may prove beneficial for more specific use cases.

Another point of interest consists in implementing new or different aspects of the rendering workflow and evaluate their performance improvements. Examples include shader, render loop and memory usage optimizations, additional rendering algorithms and newer API features. An interesting case on last category is replacing pipeline objects with *shader objects*, which offer a more simplified flow in a similar way that timeline semaphores improve over the old scheme of binary semaphores and fences. Optimizations of this kind would likely benefit all use cases of the proposed library, offering an advantage for having a centralized design over having separate applications that may need to be redeveloped to fit a different type of visualization.

A possible new feature that may be of interest consists of adding mesh shaders to the view creation process. As shown on figure 6.1 and unlike the vertex, geometry and fragment shaders currently used here, mesh shaders execute on a different pipeline which goes directly to the rasterizer. These shaders output meshlets that could be used to generate marker and voxel geometry that can be instanced as shown on figure 6.2. It is still important to measure performance between shader implementations because the classic pipeline is highly optimized in comparison to this new method. Another feature of interest is adding OptiX support [18] for enabling ray-tracing in visualizations, and might be a good fit for this library because

OptiX is available as a Slang target for code generation.

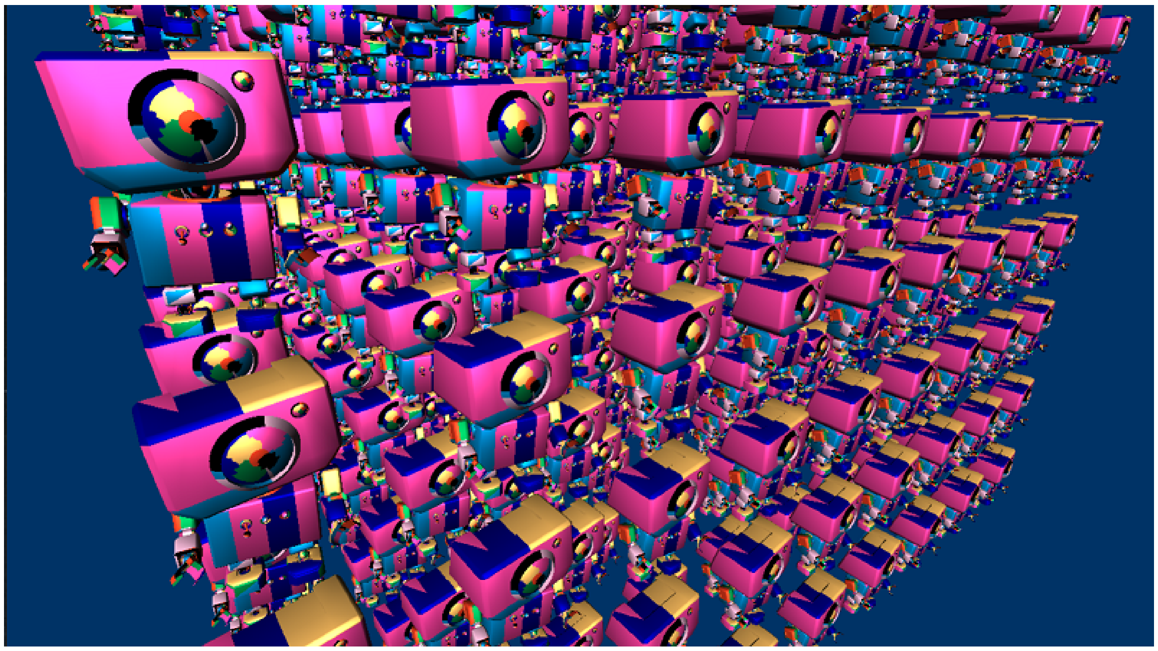


Figure 6.2: Mesh shader used to render multiple meshlet instances, obtained from the D3D12 Meshlet Instancing Sample.

The proposed solution is designed for workstation use and is not suitable for GPU servers, which have a different execution flow and resource usage. Such servers can have devices without display output, or even if they have, workarounds like X-forwarding will not work for a Vulkan-based backend such as the one presented. An alternative is to perform screencasting, obtaining the framebuffers from device memory and transferring them to the client, that would then view the generated content. Extending this library to work remotely would involve changing the design to allow a server doing interop which renders in headless mode and sends the framebuffer data to a client application which renders the result.

Bibliography

[1] Marwan Abdellah, Ayman Eldeib, and Mohamed I Owis. Gpu acceleration for digitally reconstructed radiographs using bindless texture objects and cuda/opengl interoperability. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 4242–4245. IEEE, 2015.

[2] J. Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *Visualization Handbook*, 01 2005.

[3] Julio Esteban Albornoz Valencia. Reingeniería de camaron: un visualizador de mallas de polígonos y poliedros. Master's thesis, Universidad de Chile, 2022. Available at `https://repositorio.uchile.cl/handle/2250/185497`.

[4] Luke Campagnola, Almar Klein, Eric Larson, Cyrille Rossant, and Nicolas P. Rougier. VisPy: Harnessing The GPU For Fast, High-Level Visualization. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, Austin, Texas, United States, July 2015.

[5] Aldo Canepa, Gonzalo Infante, Nancy Hitschfeld, and Claudio Lobos. Camarón: An open-source visualization tool for the quality inspection of polygonal and polyhedral meshes. pages 128–135, 01 2016.

[6] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multiplatform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[7] Fatemeh Farokhmanesh, Kevin Höhlein, Christoph Neuhauser, and Rüdiger Westermann. Neural fields for interactive visualization of statistical dependencies in 3d simulation ensembles. *arXiv preprint arXiv:2307.02203*, 2023.

[8] Nolan Goodnight. Cuda/opengl fluid simulation. *NVIDIA Corporation*, 548, 2007.

[9] Avelina Hadji-Kyriacou and Ognjen Arandjelović. Raymarching distance fields with cuda. *Electronics*, 10(22):2730, 2021.

[10] Yong He, Kayvon Fatahalian, and Theresa Foley. Slang: Language mechanisms for extensible real-time shading systems. *ACM Transactions on Graphics*, 37:1–13, 07 2018.

[11] SeongHu Hong, DoHyeong Kim, and Chang-Sung Jeong. Parallel application performance comparison with vulkan, cuda and openmp. *IEICE Proceedings Series*, 61(M2-6-5), 2016.

[12] John Kessenich, Boaz Ouriel, and Raun Krisch. Spir-v specification. *Khronos Group*, 3:17, 2018.

[13] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[14] Riley Lipinksi, Kenneth Moreland, Michael E Papka, and Thomas Marrinan. Gpu-based image compression for efficient compositing in distributed rendering applications. In *2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 43–52. IEEE, 2021.

[15] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, page 57–68, Goslar, DEU, 2002. Eurographics Association.

[16] Cristobal Navarro, Nancy Hitschfeld, and Eliana Scheihing. A parallel gpu-based algorithm for delaunay edge-flips. 03 2011.

[17] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[18] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)*, 29(4):1–13, 2010.

[19] Prabhu Ramachandran and Gael Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[20] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, 2006.

[21] C. Rossant and N. Rougier. High-performance interactive scientific visualization with datoviz via the vulkan low-level gpu api. *Computing in Science & Engineering*, 23(04):85–90, jul 2021.

[22] Diego Rossinelli, Michael Bergdorf, Georges-Henri Cottet, and Petros Koumoutsakos. Gpu accelerated simulations of bluff body flows using vortex particle methods. *Journal of Computational Physics*, 229(9):3316–3333, 2010.

[23] Nicolas P. Rougier. Antialiased 2d grid, marker, and arrow shaders. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):1–52, November 2014.

[24] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[25] Gabriel Alfonso Sanhueza Sanhueza. Blastsight: A 3d visualization library oriented to mining applications. Master's thesis, Universidad de Chile, 2020. Available at `https://repositorio.uchile.cl/handle/2250/176738`.

[26] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[27] Gonçalo Soares and João Madeiras Pereira. Lift: An educational interactive stochastic ray tracing framework with ai-accelerated denoiser. 2021.

[28] Chin-I Tang, Xianyue Deng, and Yuzuru Takashima. Cuda-opengl gpu-based real time beam tracking by mems phase slm. In *Emerging Digital Micromirror Device Based Systems and Applications XIV*, volume 12014, pages 40–44. SPIE, 2022.

[29] Chin-I Tang, Xianyue Deng, and Yuzuru Takashima. Real-time cgh generation by cuda-opengl interoperability for adaptive beam steering with a mems phase slm. *Micromachines*, 13(9):1527, 2022.

[30] Jiaji Wu, Long Deng, and Anand Paul. 3d terrain real-time rendering method based on cuda-opengl interoperability. *IETE Technical Review*, 32(6):471–478, 2015.