Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

REAL-TIME INTERACTIVE VISUALIZATION LIBRARY FOR CUDA PROGRAMS

Propuesta de tema de tesis para optar al grado de Magíster en Ciencias, mención Computación

Francisco Carter Araya

Profesores guía:
Nancy Hitschfeld K.
Cristóbal Navarro G.

Santiago de Chile
Junio, 2022

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 scientific visualization libraries, 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.

2. Problem statement

Currently there is no straightforward way to display GPU-allocated data in CUDA with existing softwares or graphics libraries, which read data in RAM from a loaded file or generated from code in the same pipeline. The typical workaround consists in transferring data from device to host, export it to a file, load it and display it with said libraries. For dynamic visualizations such as time-based simulations, where the data changes constantly from one timestep to another, achieving optimal (or even adequate) performance becomes unfeasible, as the mentioned process would need to be repeated at every iteration. This defeats one of the main purposes of GPU-accelerated visualizations, which is the interactive rendering of complex large-scale data [1]

Another possibility is to use GPGPU capabilities provided by Vulkan and OpenGL with Compute Shaders, which allow computation in a similar scheme to CUDA kernels. However,

as they are graphics-oriented APIs foremost, their setup, syntax and memory handling are much less concise. Moreover, they don't provide ways to use the latest architectural advancements in CUDA-enabled GPUs such as Tensor Cores and/or Raytracing Cores. Also, from an user standpoint, switching the experiment codebase from CUDA to Vulkan/OpenGL just for enabling visualization capabilities requires a developing effort that is bigger than the intended benefit. Instead, it would be expected to plug in said capabilities in the existing code in the form of a library or extension, just as it would be in traditional host-code solutions.

A promising alternative consists in connecting the graphics API and the GPU computing platform directly via interoperability between the two. Currently most 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 Vulkan code written specifically to a particular CUDA code.

3. Related work

3.1. Previous work

CUDA supports interoperability with Vulkan, starting from version 10 [1]. This process allows CUDA to get a memory handle to GPU memory allocated by Vulkan, in the form of raw pointers or texture objects depending on the memory usage. The mapped handles can then be used by both platforms independently, though this raises obvious concurrency issues between rendering and processing. As with all synchronization operations in Vulkan, synchronization with CUDA must be managed explicitly, using the semaphore structures provided by the interoperability API.

Arrayfire [2] Forge [2] represents a similar effort in connecting the computing and visualization pipelines, using the native interoperability between OpenGL and CUDA, over the which the Arrayfire library is built. It is focusing on large dataset plotting over visualization of raw datasets, and since it requires the Arrayfire wrappers over CUDA memory, it cannot be easily plugged in to any existing CUDA code that does not already use that library. However, it represents a successful interoperability use case.

3.2. Visualization libraries

This section presents a list of existing scientific visualization libraries, with varying degrees of specialization over specific problem domains, whose rendering and display techniques are relevant to this problem.

[1] https://docs.nvidia.com/cuda/archive/10.0/cuda-toolkit-release-notes/index.html. Retrieved 2022-05-13.
[2] https://github.com/arrayfire/forge. Retrieved 2022-06-17.

ParaView [3] is a high-performance framework for data analysis and visualization based on VTK [4]. 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.

Camarón [5] is a visualization tool focused on mesh rendering and analysis, written in OpenGL/C++. 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. The library was updated in 2022 to perform 75 % faster while using 18 % less memory.

MayaVi [6] is a general-purpose 3D visualization library with interactive and scripted modes, written in Python and VTK. 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.

BlastSight [3] is an interactive 3D visualization application for academic and industrial mining uses, developed in Python and OpenGL. 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.

Datoviz [7] is a performance-oriented scientific data visualization library and intermediate-level graphics API, writen in Vulkan/C++ with native Python bindings. Part of its code and design ideas are derived from VisPy [8], 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.

4. Research questions

- Are dynamic Vulkan visualizations of CUDA-mapped memory more efficient in memory use and frame rate than the traditional approach of generating the data in host memory and transferring it for rendering? Is Vulkan performance at inter-operating with CUDA better than OpenGL?
- Is it possible to add a visualization pipeline to CUDA code without altering the kernel call workflow? What changes or replacements would need to be made to CUDA code in

[3] https://repositorio.uchile.cl/handle/2250/176738. Retrieved 2022-05-13.

order to integrate Vulkan visualizations to it?

- To which degree can Vulkan code be abstracted from CUDA code while interoperating between both languages?

5. Hypothesis

A new high-performance Vulkan visualization library designed specifically for integration with CUDA code will perform better in memory use and frame rate than existing alternatives for the interactive display of GPU-generated data.

6. Main goal

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

7. Specific goals

- 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 by the library, such as setting colors, illumination and camera position.
- Research the scope to which it is more advantageous to use the proposed library over using OpenGL or Vulkan as a unified compute-rendering environment.
- Measure the improvement of generating mapped memory visualizations over doing explicit memory copies, in terms of memory use and frame rate.

8. Methodology

8.1. Research

The research process will start by reviewing the state of the art for HPC visualization, including general visualization techniques and libraries. As Vulkan is a relatively new API and there is not much research for using interoperability with CUDA, most of the reviewed work will be focused on scientific visualization libraries and their respective software architectures. As such, the approaches listed in section 3.2 are part of the state of the art in that area. The review scope will also be narrowed to visualization methods and libraries for single workstations, rather than distributed systems or dedicated HPC servers.

It is also important to review the current state of the Vulkan and CUDA APIs, putting special attention at the interoperability capabilities between the two. As Vulkan is a very recent and actively developed library, new releases and extensions can change or improve the interoperability behavior, which is at the core of the proposed solution. The official CUDA Toolkit code samples include Vulkan interoperability examples [4] [5], which can used as a base for developing the library architecture and features.

Even though most of the reviewed libraries are not developed in C++/Vulkan, their algorithms, data access patterns and shaders are a useful reference during the design of the library and later comparisons between them and the developed result. By knowing implementation details of the reviewed libraries, it becomes easier to understand and explain the performance differences that may arise.

8.2. Experimentation

All of the planned experiments will be performed with the proposed set of primitives: meshes, point clouds, textures and voxels. Likewise, the measured variables for all experiments will be frame rate and GPU memory use.

- For each primitive, gather datasets of varying input sizes and configurations, which may include element size, density, color (global or per-element) and other parameters of interest. Most of the listed primitives have publicly available datasets that should be used instead of generating new data.
- Generate visualizations of the generated datasets with the proposed approach and the state of the art libraries of softwares. Measure and compare the variables for static and dynamic (when applying rotation, translation, zoom, etc.) visualizations.
- Perform scalability and stress tests for the proposed approach and the state of the art libraries. Display large/multiple datasets up to hardware limits and measure the variables for varying input sizes, generating plots for performance and memory use.
- Implement kernels for iterative processing of the datasets. Perform a large number of iterations, monitoring the variables for possible performance degradation or memory leaks. Visually inspect snapshots of the experiment at regular intervals in search of rendering artifacts that may occur.

8.3. Current progress

Most of the work done consists on designing the interoperability architecture, generating visualizations and implementing both features into the proposed library. The library currently supports CUDA-Vulkan interoperability for the full set of primitives proposed on the previous sections. For each primitive, an expected CUDA data structure (that is, the structure of the input data to visualize) was selected for library implementation. The input data structures match common CUDA programming use cases; for example, mappings for

[4] https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/simpleVulkan. Retrieved 2022-05-13.

[5] https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/vulkanImageCUDA. Retrieved 2022-05-13.

linear memory buffers were implemented over opaque `cudaArray` layouts, which are typically less used. However, the architecture design accounts for ease of integration of new memory mappings which may be implemented in the future.

Sample CUDA programs containing use cases for each primitive were implemented and tested with the library, for validation of the generated visualization and preliminary performance analysis. With the current library design, a CUDA programmer would only need to replace the GPU memory allocation calls, as mapped memory allocation must be done in Vulkan. Moreover, interactive visualization operations such as adjusting the camera position and changing colors are possible in real time even when the corresponding CUDA kernel is running, with no detected visual artifacts or reduced performance. This is possible since the library implements synchronization between the visual pipeline, the CUDA kernel and the CPU control flow.

Ongoing work is focused on extending the library architecture to support visual parameters such as primitive colors and illumination, which could be scripted on the experiment code or changed at runtime with a graphical interface (GUI).

9. Expected results

The research is expected to produce the following results:

- An open source C++/Vulkan library for generating real-time visualizations of CUDA/-Vulkan mapped GPU memory.
- A sample code demonstrating the library capabilities, based on a real-world use case of an existing CUDA program that produces a visually meaningful result (that is, matching at least one of the proposed primitives)
- A review of the state of the art methods and libraries for HPC scientific visualization on workstations with consumer-grade GPUs.
- A report detailing the differences in performance and memory usage between the proposed GPU memory-mapping scheme and performing explicit CPU-GPU data transfers, defining the scope on the which memory mappings are more adequate over transfers.

Bibliografía

[1] Cao, Y., Wang, H., y Ai, Z., "Linking visualization and scientific understanding through interactive rendering of large-scale data in parallel environment," en 2015 International Conference on Virtual Reality and Visualization (ICVRV), (Los Alamitos, CA, USA), pp. 260–263, IEEE Computer Society, 2015, doi:10.1109/ICVRV.2015.59.

[2] Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K., y Melonakos, J., "ArrayFire: a GPU acceleration platform," en Modeling and Simulation for Defense Systems and Applications VII (Kelmelis, E. J., ed.), vol. 8403, pp. 49–56, International Society for Optics and Photonics, SPIE, 2012, doi:10.1117/12.921122.

[3] Ahrens, J., Geveci, B., y Law, C., "Paraview: An end-user tool for large data visualization," Visualization Handbook, 2005.

[4] Schroeder, W., Martin, K., Lorensen, B., y Kitware, I., The Visualization Toolkit: An Object-oriented Approach to 3D Graphics. Kitware, 2006, https://books.google.cl/books?id=rx4vPwAACAAJ.

[5] Canepa., A., Infante., G., Hitschfeld., N., y Lobos., C., "Camarón: An open-source visualization tool for the quality inspection of polygonal and polyhedral meshes," en Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - GRAPP, (VISIGRAPP 2016), pp. 130–137, INSTICC, SciTePress, 2016, doi:10.5220/0005830501280135.

[6] Ramachandran, P. y Varoquaux, G., "Mayavi: 3D Visualization of Scientific Data," Computing in Science & Engineering, vol. 13, no. 2, pp. 40–51, 2011.

[7] Rossant, C. y Rougier, N. P., "High-performance interactive scientific visualization with datoviz via the vulkan low-level gpu api," Computing in Science Engineering, vol. 23, no. 4, pp. 85–90, 2021, doi:10.1109/MCSE.2021.3078345.

[8] Campagnola, L., Klein, A., Larson, E., Rossant, C., y Rougier, N. P., "VisPy: Harnessing The GPU For Fast, High-Level Visualization," en Proceedings of the 14th Python in Science Conference (Huff, K. y Bergstra, J., eds.), (Austin, Texas, United States), 2015, https://hal.inria.fr/hal-01208191.