InteropUnityCUDA: A Tool for Interoperability Between Unity and CUDA

David Algis, Bérenger Bramas, Emmanuelle Darles, Lilian Aveneau

InteropUnityCUDA: A Tool for Interoperability between Unity and CUDA

David Algis[1,2,*], Berenger Bramas[3], Emmanuelle Darles[1], and Lilian Aveneau[1]

[1]Université de Poitiers, Univ. Limoges, CNRS, XLIM, France
[2]Studio Nyx
[3]Inria Nancy Grand Est, ICube, France
[*]Corresponding Author : david.algis@univ-poitiers.fr

2025-02-13

Abstract

Unity is a powerful and versatile tool for creating real time experiments. It comes with a homemade c-like language to program GPGPU massively parallel algorithms: Unity's compute shader language. As Unity has been primarily made for multi-platform games creation, this language comes with several limitations: for example, it does not support multi-GPU computation, and it lacks of complete math libraries. To overcome these flaws, the GPU manufacturers have developed programming models that allow developers to leverage the power of modern GPUs for general-purpose computing, like CUDA or HIP. Therefore, to bring these features and flexibility to Unity, this article proposes an open source tool to perform interoperability between Unity and CUDA.

Keywords: Unity, CUDA, Interoperability, Software Tools, Real-time Systems, Parallel Programming, Programming Techniques

1 Introduction

Unity is one of the most widely used game engines in the industry[1]. As a result of this popularity, Unity has expanded its usages, offering a tool more suited to cinema, or immersive teaching. Therefore, Unity offers a wide range of tools, from rendering engine to networking services by way of virtual reality support. What's more, one of the keys to Unity's success is that it has been designed to support as many operating systems and devices as possible.

As an engine for real time application it provides two high performance tools for parallel computing: Job system for Burst compiler for multithreading on central processing unit (CPU), and compute shader for multithreading on the graphics processing unit (GPU)[2], which is the focus of this paper.

A compute shader provides high-speed general purpose computing and takes advantage of the large numbers of parallel processors on the GPU. Compute shader in Unity closely match to DirectX 11 technology. Moreover, they have been made to follow the philosophy of Unity, in a way that any compute shader program should run on a wide variety of devices. Therefore,

[1]Pérez *et al.* [15] and an article of Slash Team [18] indicates that it has the largest share of the game engine market: 38%, just ahead Unreal Engine which has 15%.

[2]Note that it does not concern computation parts of the rendering pipeline that are covered by more traditional rendering shaders (vertex shader, fragment shader, *etc.*).

Unity's compute shader is using the lowest common denominator technology, ensuring it runs smoothly on both the latest GPU and older, less powerful devices.

This leads to some lacks of functionality. More specifically, compute shader are missing: debugging tools, object-oriented programming (OOP) possibilities, standardized libraries with implementations of classical parallel algorithms, or ways of doing meta-programming development. It is, in fact, complex to carry out large-scale developments with compute shader only.

For these reasons, when developers want to build specific high-performance applications, they use more dedicated programming language and application programming interfaces (API) like OpenCL, SYCL, HIP and CUDA.

CUDA is historically the most widely used. Indeed, in addition to a *network effect* presented by Cusumano [4], and to being very convenient for the developer, its performance is comparable to or even better than that of its competitors as demonstrated by Su *et al.* [17], Fang *et al.* [8] and Costanzo *et al.* [3].

In order to profit of Unity rendering functionality and CUDA polyvalence, the authors of this article have made a tool to perform interoperability between Unity and CUDA: `InteropUnityCUDA`.

It makes possible to edit with CUDA API any previously allocated memory on GPU from Unity. For example, imagine a texture that is created and rendered in Unity while in the same time the content of the texture is written by CUDA. `InteropUnityCUDA` is completely available on GitHub with a MIT licence, at this address: https://github.com/davidAlgis/InteropUnityCUDA.

The contributions of this article are as follows:

- **A novel methodology for interoperability between game engines and GPU backends**: We present a method for enabling direct interaction between game engine graphics objects and GPU backends. While demonstrated using Unity and CUDA, this approach can be generalized to other game engines and GPU APIs, offering a flexible solution for high-performance computing applications in interactive environments.
- **Allowing CUDA plugins cooperating with Unity rendering pipeline**: `InteropUnityCUDA`'s main purposes is to allow Unity extension by natives plugins that can read and write Unity's graphics object through CUDA. These plugins are not provided by `InteropUnityCUDA`, which give tools for this interoperability.
- **An in-depth analysis of the benefits and limitations of `InteropUnityCUDA`**: We detail how `InteropUnityCUDA` leverages CUDA's capabilities, such as multi-GPU support and advanced memory management, to overcome the constraints of Unity's compute shaders. This discussion also highlights potential limitations of the approach, providing insights for further development and adaptation.
- **Comprehensive performance comparisons with Unity's Compute Shader**: We evaluate the performance of `InteropUnityCUDA` in multiples experiments, comparing its performance to Unity's native compute shaders to illustrate the potential improvements in computational efficiency.
- **Open-source access to `InteropUnityCUDA` and examples**: To support reproducibility and encourage further research, we provide the tool and accompanying applications in two GitHub repositories, including usage examples and benchmarks.

The rest of this article is organized as follows: Section 2 positions this paper in the literature and state of the art. Section 3 focuses on a complete description of the principle of `InteropUnityCUDA` and answers the questions of how it works, how to use it, and what its limitations are. Section 4 compares `InteropUnityCUDA` and Unity's compute shaders,

through four different use cases including CPU to GPU memory copy, vector addition, array reduction, and 2D waves simulation. These four examples are publicly accessible on a GitHub repository BenchmarkCSvsInteropUnityCUDA under a MIT licence, at this address: https://github.com/davidAlgis/BenchmarkCSvsInteropUnityCUDA. Finally, Section 5 concludes this paper and outlines some directions for future work.

2 Related Works

2.1 Software extension

Software extensibility enables applications to adapt and expand by incorporating additional modules or plugins without altering the core system. This modular approach allows developers to introduce new features or modify existing ones, enhancing flexibility and maintainability. Hao *et al.* [10] propose a framework that employs design patterns to support third-party extensions, enabling seamless integration of new functionalities into existing applications. Similarly, Kouskouras *et al.* [11] demonstrate how combining design patterns with aspect-oriented programming facilitates software extension by promoting modularity and reducing code tangling. Unity has multiple supports of extensibility, for instance through native plugins written in C, C++, or Objective-C, allowing interaction with the rendering pipeline via graphics API code as demonstrated in the Github repository NativeRenderingPlugin, available at this address `https://github.com/Unity-Technologies/NativeRenderingPlugin`. However, it lacks direct integration in Unity's rendering pipeline with CUDA. `InteropUnityCUDA` addresses this gap.

Different works propose to extend software using a well known design technique: the Command pattern. For instance, Damyanov *et al.* [5] use this pattern in the context of the web, allowing to encapsulate queries and objects as function parameters. In a different context, Zhao *et al.* [19] use it in microservice architecture applications, allowing easy extensions of softwares and introduction of new functionalities. Kröher *et al.* [12] use the Command pattern for integrating distributed and central control in self-adaptive system. In this article, the extension of Unity for Cuda is implemented using a combination of Unity native plugins and the Command pattern.

2.2 Cross-Platform GPU Abstractions and Wrappers

Several libraries enable cross-platform GPU programming through unified abstractions. Rockenbach *et al.* [16] provide GSParLib, a C++ interface for GPU processing with CUDA and OpenCL focusing on portability between different GPU platforms. Carter *et al.* [1] developed Kokkos to address performance portability across many-core architectures, including both GPU and multi-core CPU, by unifying data parallelism and memory access patterns in a scalable C++ API. Ernstsson *et al.* [7] have made SkePU 2 a tool to enhance the flexibility of skeleton programming for heterogeneous systems. Each of these frameworks enables streamlined high-performance computing but lacks direct integration with game engines like Unity, which this work specifically addresses. Hence, their purpose is quite different and they deal with GPU usage in programming. However, they can be used through `InteropUnityCUDA`.

2.3 Integrating CUDA with Unity

There are few known attempts to use the potential of CUDA in Unity. We can, however, cite two Github repositories: On the one hand, Unity3D-CUDA illustrates a method for executing CUDA kernels in Unity by pre-compiling CUDA code into PTX, which can then be integrated into Unity projects using C# bindings. This repository is accessible at this address `https://github.com/przemyslawzaworski/Unity3D-CUDA`. This repository only provides a

proof of concept of this method, but not a full pipeline allowing large scale production. On the other hand, in a more roundabout way uNvPipe provides a wrapper for NVIDIA's NvPipe, a zero-latency video compression library designed for interactive remoting applications, enabling GPU-based video encoding directly within Unity. This repository can be accessed at `https://github.com/hecomi/uNvPipe`. Both solutions are valuable but focus on specific and limited functionalities rather than the full range of CUDA's capabilities within Unity.

2.4 Other Game Engines

Other game engines offer varying degrees of CUDA integration. Unreal Engine, built in C++, allows seamless CUDA use, as demonstrated by Kuang *et al.* [13], enabling direct GPU computation integration without wrappers. Godot Engine, while lacking native CUDA support, permits custom C++ modules with external library bindings; a demonstration of CUDA usage is available at this address `https://github.com/davidAlgis/godot_cuda`. Unity's C# foundation, by contrast, requires additional tools for full CUDA access, a gap that this work aims to address. Moreover, no demonstrations were found for either engine that apply CUDA calculation directly to rendering objects.

3 Presentation of the Tool and its Architecture

3.1 Underlying Principle

`InteropUnityCUDA` allows interoperability between Unity and CUDA, to bypass the flaws of compute shader and enjoy all the benefits of CUDA. It allows Unity extension by natives plugins that can interact with Unity's "graphic object" through CUDA. The principle of `InteropUnityCUDA` can be summarized in one sentence: *Send and "cast" the pointer of the memory of Unity "graphic object" to CUDA to modify it directly through CUDA API.* It is important to note that this approach is not specific to Unity; it could be applied to any game engine that allows C++ interaction and access to native memory pointers of "graphic objects."[3]

More precisely, `InteropUnityCUDA` works with three types of such Unity's "graphic object":

- Texture2D[4]: A 2D texture is commonly used to store image data. It represents a two-dimensional grid of pixel colors.
- Texture2D Array[5]: An array of 2D textures with the same width, height, and format, allowing multiple textures to be stored in a single object and accessed in parallel.
- Compute Buffer[6]: A buffer designed to be used with compute shaders, enabling the transfer of structured data between the CPU and GPU.

Unity supports multiple graphics APIs (such as OpenGL, Vulkan, and DirectX11), and these three Unity types serve as wrappers for the native types of the chosen graphics API. The process of using `InteropUnityCUDA` in a Unity project involves the following steps (see Figure 1):

1. Create one of the above Unity types for rendering in Unity.
2. Retrieve a pointer to the corresponding native type of the graphics API.

[3]For example, `InteropUnityCUDA` in Unity uses the method `GetNativeTexturePtr` to retrieve the memory pointer of a 2D texture, whereas in Unreal Engine, the equivalent method would be `GetReferencedTexture`.
[4]see `https://docs.unity3d.com/ScriptReference/Texture2D.html`
[5]see `https://docs.unity3d.com/ScriptReference/Texture2DArray.html`
[6]see `https://docs.unity3d.com/ScriptReference/ComputeBuffer.html`

3. Send this pointer to `InteropUnityCUDA`.
4. Use CUDA's interoperability functions to register the pointer, enabling future access within CUDA kernels.
5. Read/Write from a CUDA kernel some data to this pointer.
6. When the rendering experience is done, use CUDA's interoperability functions to unregister the pointer (undo step 4).
7. Release the graphics object in Unity (undo step 1).

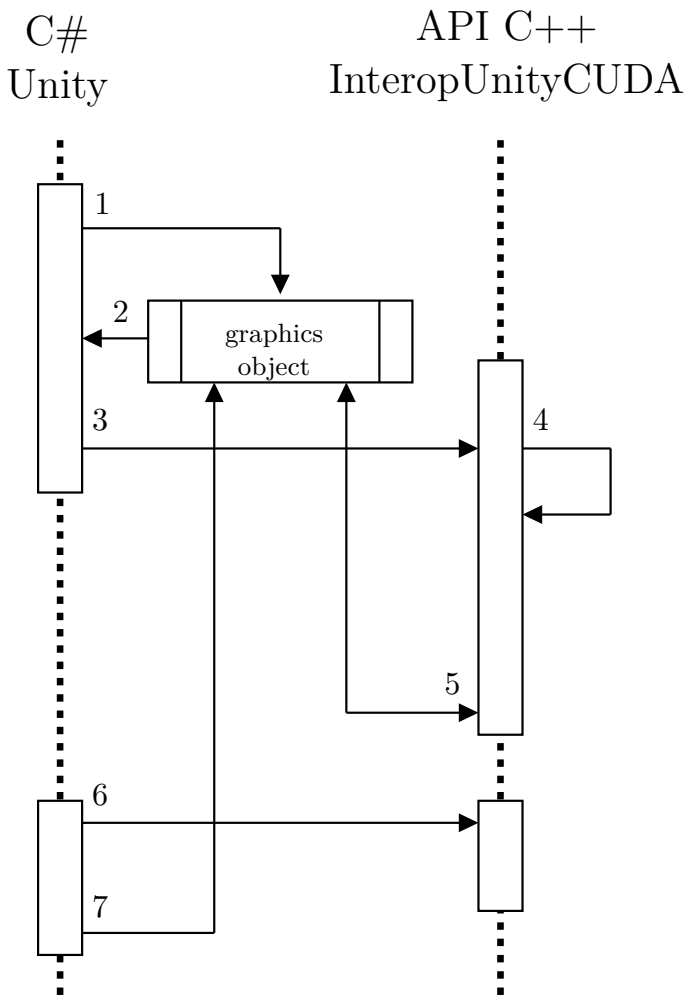

Figure 1: Sequence diagram for the process of using `InteropUnityCUDA` in a Unity project. Each number corresponds to the step described in Section 3.1.

These steps represent the main challenge of implementing `InteropUnityCUDA`, that is: a way to juggle between graphics object memory in two different environments. As mentioned before, these steps are not specific to Unity, they could be applied to any game engine. This principle, allows `InteropUnityCUDA` to profit from Unity rendering and its easy creation for graphics memory in addition to CUDA polyvalence and performances for editing this memory.

It should be noted that these operations must be executed from the render thread in Unity, which can be achieved using the Unity's `GL.IssuePluginEvent` method. The documentation on this method is available at this address `https://docs.unity3d.com/ScriptReference/GL.IssuePluginEvent.html`. This Unity method is quite specific, as it sends *a user-defined*

command to a native code plugin. The event sent is an integer. As discussed in the next section, this induces some specific mechanisms to allows using different events.

3.2 Using Action Pattern

While `InteropUnityCUDA` has been designed to be used intensively, manually calling many functions via `GL.IssuePluginEvent` can be tedious when dealing with a wide range of functions. To address this, `InteropUnityCUDA` is build upon two different mechanism: a dictionary of actions and the **action pattern** (aka **command pattern**). Let's remember that this pattern is defined as a behavioral design pattern that encapsulates a request as an object, allowing for parameterization of clients with different requests (for more details about this, see Figure 2 and the dedicated chapter in the book of Gamma *et al.* [9]).

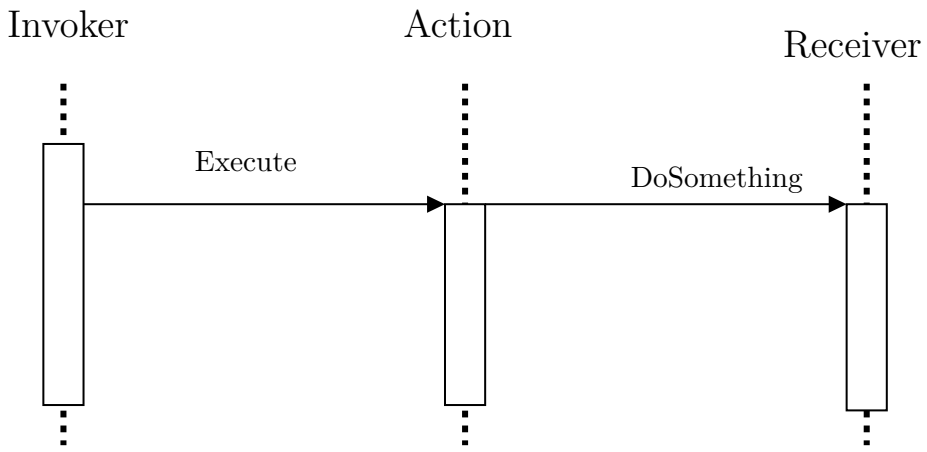

Figure 2: Sequence diagram for the command pattern.

The command pattern offers two particular advantages. First, it simplifies the process of interoperability calls, as the user (from Unity) only needs to create an "*Action*" from a tier's library, to register it to `InteropUnityCUDA` to simplify its calling through `GL.IssuePluginEvent` Unity function, and then to execute it at each rendering frame. Second, it allows for a better encapsulation, as a user mostly interacts with `InteropUnityCUDA` library only, and not directly with the library that implements the action and performs some CUDA computing.

An abstract class named `Action` is proposed in `InteropUnityCUDA` to build extension libraries in C++/CUDA. Unlike traditional pattern implementations, which have a single execution function, this abstract class contains three virtual functions named `Start`, `Update`, and `OnDestroy`. They correspond to a "minimum" life cycle for game engine logic: `Start` performs allocation and prepare the action, `Update` calls the main computation for each frame update, and `OnDestroy` free the allocated memory in `Start`.

Any concrete class that performs computations should inherit from this abstract class. Its implementation is then proposed in tier's CUDA library. A generic wrapper for these Actions is also proposed for the Unity programmer. Once the user has built such an action, he can register it in `InteropUnityCUDA`, and then call its `Update` method at each rendering frame. Notice that this call (the same for `Start` and `OnDestroy`) is made through `InteropUnityCUDA` only, again to simplify the work of game developers in Unity.

3.3 Wrappers for Graphics API

CUDA's interoperability functions are specific to each graphics API. This introduces certain limitations as discussed in Section 3.5. Hence, to simplify the development of CUDA based

plugin and to respect the Unity's philosophy, three wrappers are proposed in `InteropUnityCUDA`, one for each of the three kinds of Unity Graphical Object (Texture 2D, Texture 2D Array, and Compute Buffer). These wrappers enable the object's data to be used in CUDA without knowledge of the originating graphics API.

These wrappers are generally set by Actions inside their `Start` methods and released in their `OnDestroy` ones. Thus, they are not exposed to the Unity programmer, but only for the CUDA library one. In practice, game programmers send a pointer of some Unity wrapper graphic object to the action constructor, and this pointer is used for later registration inside the CUDA library that implements the action.

3.4 A Complete Example

To clarify the whole process of Unity and CUDA interoperability, let's look at an example where it is assumed that the user want to create an Action that writes in a Unity 2D texture from CUDA. An implementation of this example can be found in Appendix B. As mention in Section 1, `InteropUnityCUDA`'s main purposes is to allow Unity extension by natives plugins that can read and write Unity's graphics object through CUDA. These plugins are not provided by `InteropUnityCUDA`, which gives tools for this interoperability only. Notice that such an example of how to implement 2D texture writing via `InteropUnityCUDA` can be found in `SampleBasic` library on GitHub at `https://github.com/davidAlgis/InteropUnityCUDA`. Upon the creation of:

- A new library: `MyPlugin`.
- A class in `MyPlugin`: `MyActionTex++` that inherits from `Action`. It stores a pointer to a texture, that is initialized by the constructor that receives the graphical object from Unity call, using a simple function that is exported by `MyPlugin` library (this implements the **Factory pattern**). In the `Start` method, it registers the pointer of texture defined in constructor in its surface object, which is an equivalent of texture for CUDA API. In `OnDestroy` method, it does the reverse. Its `Update` method calls the CUDA kernel that fills the texture with come calculation.
- A class in Unity: `MyActionTex#` that inherits from `ActionUnity` that makes the link between `MyPlugin` and `MyActionTex++`.

When the simulation begins (see Figure 3), the process is initiated by sending the native pointer of the new Unity texture from `MyActionTex#` to `MyActionTex++` (1). `MyActionTex++` then forwards this pointer to `InteropUnityCUDA` (2), which returns a Texture object initialized according to the graphics API (3). Following this, `MyActionTex++` provides `MyActionTex#` with a pointer to the newly created Action (4). `MyActionTex#` registers this Action with the `InteropUnityCUDA` library, assigning it a unique identifier (5), and `InteropUnityCUDA` stores the Action in a dictionary indexed by this ID (6). Finally, `MyActionTex#` calls the `Start` method of the Action through `InteropUnityCUDA` using the unique ID (7), allowing `InteropUnityCUDA` to retrieve the corresponding Action from the dictionary (8) and invoke the `Start` method of `MyActionTex++` (9). At this stage, `MyActionTex++` registers the texture for later CUDA access (10).

During the simulation update phase, `MyActionTex#` periodically calls the `Update` method of the Action via `InteropUnityCUDA` using the unique ID (11). `InteropUnityCUDA` retrieves the relevant Action using the ID from the dictionary (12) and invokes the `Update` method of `MyActionTex++` (13). In turn, `MyActionTex++` executes a CUDA kernel that performs read/write operations on the texture (14), and Unity updates the rendering using the texture modified by CUDA (15).

At the conclusion of the simulation, `MyActionTex#` initiates the cleanup process by calling the `OnDestroy` method of the Action via `InteropUnityCUDA` using the unique ID (16).

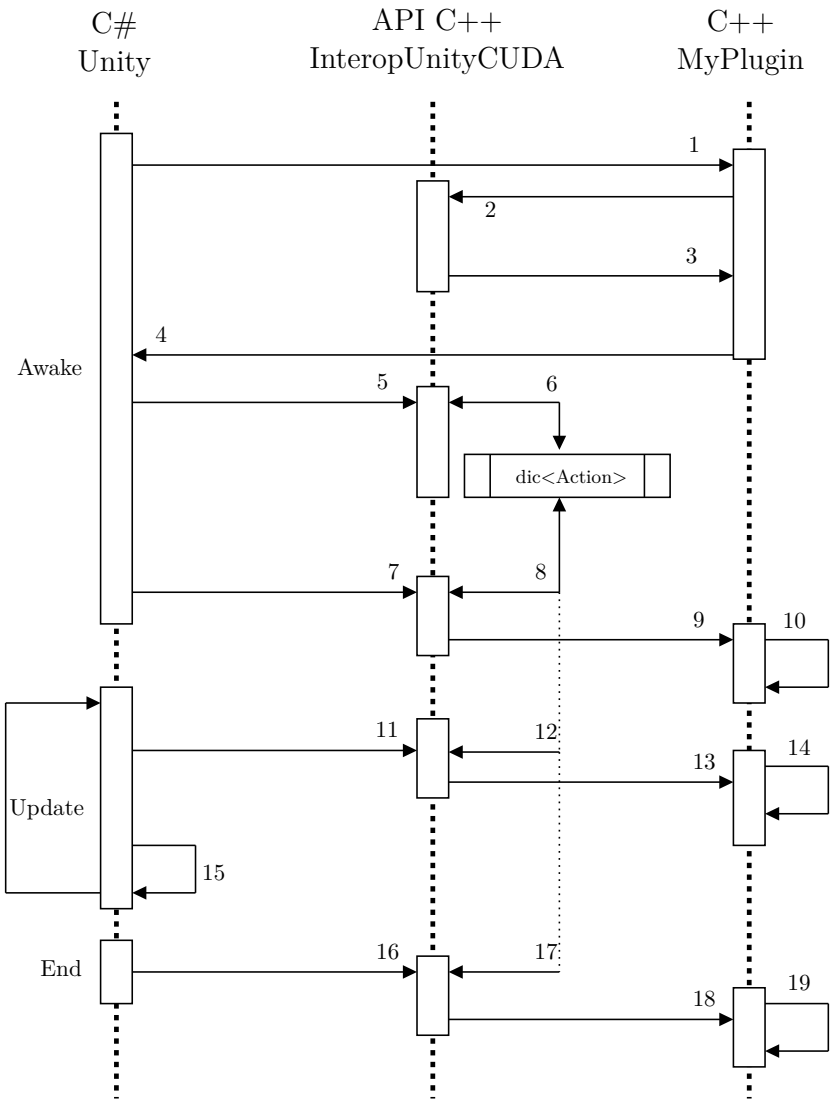

Figure 3: Sequence diagram representing an example of usage of `InteropUnityCUDA` with the command pattern. Each number corresponds to the step described in Section 3.2.

`InteropUnityCUDA` retrieves the associated Action from the dictionary (17) and invokes the `OnDestroy` method of `MyActionTex++` (18). Finally, `MyActionTex++` unregisters the texture, clearing it from CUDA usage (19).

Although `InteropUnityCUDA` introduces additional code complexity (as shown in Appendix B, it requires an Action in C#, an Action in C++, and dedicated CUDA code instead of a single compute shader with minimal calls) this is offset when managing large-scale projects with numerous graphic objects. Furthermore, compute shader syntax is generally more cumbersome than CUDA's, so the additional setup in `InteropUnityCUDA` becomes negligible in comparison to the advantages it offers when tackling large GPU computations. There are several examples of use of `InteropUnityCUDA`, directly in project, in the `Sample` folder.

3.5 Limitation of InteropUnityCUDA

`InteropUnityCUDA` comes with a few limitations. First, as it uses CUDA it works on Nvidia GPU only. These devices represent the vast majority of GPU on the market (see R. Dow [6]). However, there are some devices on which it doesn't work, like Macbook Pro, PlayStation 5,

etc. Additionally, as the interoperability is graphics API dependent, it needs an implementation for each graphics API and the registration of the graphics object pointer in CUDA is made with CUDA interoperability functions. That is why the support of graphics API depends on these functions and their future update by Nvidia. Today, `InteropUnityCUDA` supports the two following API:

- OpenGL ES.
- DirectX 11.

4 Comparison InteropUnityCUDA/Compute Shader

4.1 New Available Features with InteropUnityCUDA

`InteropUnityCUDA` introduces a range of new capabilities for high-performance computing in Unity by leveraging CUDA features not available in its compute shaders. For instance, it enables multi-GPU support for distributing computations across multiple GPUs for intensive tasks, and utilizes CUDA streams to facilitate asynchronous data transfers, reducing latency and enhancing performance.

Among the new possibilities introduced by `InteropUnityCUDA`, is the change of language paradigm. Indeed, compute shader is a C-like language; as for CUDA it works perfectly with C++ language and benefits of all the advantage of C++, such as OOP or meta-programming development. Additionally, CUDA gives access to a large panel of debugging tools, like cuda-gdb, the NSight suite, or the possibility to use `printf` directly within a kernel. In terms of memory management, `InteropUnityCUDA` supports advanced CUDA memory models, including unified memory, pinned memory, and warp-level primitives, which facilitate efficient data sharing and synchronization across threads. Additionally, it enables dynamic parallelism, allowing kernels to launch other kernels, thereby supporting more complex computations directly on the GPU. Besides, `InteropUnityCUDA` gives access to a wide range of functions available in the different CUDA libraries as cuFFT, Thrust, cuBLAS, cuSolve, CUB, and cuRAND. These libraries are widely used and have been extensively debugged and optimized. Developers also benefit from access to a massive online repository of CUDA code. These possibilities can be leveraged to speed up application development and execution in Unity.

Table 1 summarizes a comparison of `InteropUnityCUDA` features against compute shader ones. This list is not exhaustive; CUDA is an extensive tool with numerous possibilities that cannot be fully described here.

Table 1: Feature comparison between Unity's compute shaders and `InteropUnityCUDA`

Feature	**Unity's Compute Shader**	**`InteropUnityCUDA`**	**Description**
Multi-GPU Support	No	Yes	Distributes computations across multiple GPUs, useful for large tasks and high-performance applications.
CUDA Streams	No	Yes	Enables asynchronous operations to overlap data transfer and calculations, reducing latency and improving performance.
Debugging Tools	No	Extensive	Offers access to advanced CUDA debugging tools like NSight, cuda-gdb, and `printf` within kernels.
Dynamic Parallelism	No	Yes	Supports launching kernels from within other kernels, allowing more complex, hierarchical computation structures and adaptability within GPU programs.
Library Support	Basic Math	Extensive	Supports CUDA libraries providing a wide range of optimized functions.
C++ Advantages	No	Yes	Offers C++ features like OOP and templates, enabling more structured and maintainable code within CUDA.
Unified Memory Access	No	Yes	Simplifies memory management by enabling both CPU and GPU access to a shared memory pool, reducing the need for explicit data transfers.
Advanced Memory Management	No	Yes	Provides access to pinned, mapped, and managed memory, allowing faster and more efficient data transfer between CPU and GPU.
Warp-Level Primitives	No	Yes	Supports warp-level functions like `warpShuffle` and `warpVote`, enabling fine-grained data sharing within thread groups for efficiency.
Occupancy Control	No	Yes	Allows detailed control over GPU resource allocation (*e.g.* shared memory and thread block size) to optimize performance.
Extended Profiling and Monitoring	Limited	Extensive	Offers deep profiling tools (*e.g.* Nsight Systems) to analyze memory patterns, warp divergence, and cache utilization.

4.2 InteropUnityCUDA/Compute Shader Performance Comparison

Section 4.1 provides a list of functionalities accessible through `InteropUnityCUDA` library. The focus now shifts to what is generally the main motivation for high-performance computing: computational efficiency. To explore this, four tests are proposed that compare different types of computations on compute shaders, their equivalents using `InteropUnityCUDA`, and a baseline CPU implementation to observe the speedup achieved through GPU acceleration. The complete code and data are available in the github repository BenchmarkCSvsInteropUnityCUDA at `https://github.com/davidAlgis/BenchmarkCSvsInteropUnityCUDA`.

Firstly, it should be made clear that these tests are not, and cannot be, representative of all types of computations that can be made on GPU. However, the authors think that they provide a correct view of the benefits and drawbacks of `InteropUnityCUDA`. With these clarifications in mind, let's take a closer look at the tests. Four tests were conducted:

1. GetData.
2. VectorAdd.
3. Reduce.
4. WavesFDM.

Each of them are discussed in dedicated sections below, but first, some general points should be highlighted. As mentioned earlier, there is a compute shader code and its equivalent using `InteropUnityCUDA`. The API for controlling compute shader does not offer timer functions (like event management in CUDA) or simple synchronization function (like `cudaDeviceSynchronize`) to estimate the execution time of a given compute shader. For this reason, after each test some data were read on CPU to force the synchronization between GPU and CPU. This approach allows for the estimation of the execution time of the core computation in addition to the extra data transfer between GPU and CPU[7]. The tests include comparisons with a sequential CPU version, where no data is transferred to the GPU. This CPU version offers a baseline for understanding the potential speedup of GPU computations. The test procedures are detailed below.

4.2.1 Get Data

This test evaluates the data transfer performance between the GPU and CPU, which is a fundamental aspect of GPGPU applications. Efficient data movement is crucial because it can become a bottleneck in high-performance computing tasks, especially when dealing with large datasets.

More specifically it measures the time taken to copy a compute buffer of size N from the GPU memory to the CPU memory. For the Unity's compute shader, it uses the built-in `GetData` method. Some details about this method can be seen at this link `https://docs.unity3d.com/ScriptReference/ComputeBuffer.GetData.html`. In contrast, `InteropUnityCUDA` utilizes the `cudaMemcpy` function provided by CUDA for data transfer. The CUDA documentation about this method is available with this link `https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html`.

In this test, the CPU version was not evaluated, as it would not make sense to measure GPU-to-CPU data transfer performance on the CPU alone.

4.2.2 VectorAdd

This test evaluates the performance of a simple but fundamental parallel calculation: the addition of two vectors. It is an example of a more general design pattern in parallel programming: the binary map or transform, which involves applying a binary operator or function to a couple of inputs (see McCool *et al.* [14]). It sums two vectors of N floats and stores the result in a third vector, then copies the first float of the resulting vector back to the CPU. Vector addition, while simple, is often part of more complex computations, making it a

[7] An alternative is to use the homemade package Compute Shader Performance Estimation available at this address `https://github.com/davidAlgis/com.studio-nyx.compute-shader-performance-estimation`, which utilizes functions to retrieve the execution time as measured by Unity, written in the Unity profiler. However, since the focus is on comparing `InteropUnityCUDA` and compute shaders, a more flexible solution was chosen.

useful baseline for comparing performance between Unity's compute shader and `InteropUnityCUDA`. This test allows us to examine both compute and memory access efficiency, providing insights into how well each platform handles basic arithmetic operations on the GPU.

4.2.3 Reduce

This test applies a parallel reduction on an array of size N and copies the result from the GPU to the CPU. For this test, the CUB library is utilized to perform the reduction using its DeviceReduce function. More details about this function is available in CUB documentation at this address `https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceReduce.html#_CPPv4N3cub12DeviceReduceE`. The results demonstrate impressive performance; however, an equivalent implementation using compute shaders cannot be directly applied, as no similar library exists. Additionally, no open implementation of parallel reduction for Unity's compute shaders was found, so a custom implementation was developed. This consists of doing a partial reduction into each block of threads of the compute shader, and then to add their results to a global variable. To protect the access of this variable, a spinlock is made thanks to some atomic functions.

Before resorting to the spinlock to sum the results of each block, we initially attempted a more conventional approach: recursively calling the kernel to accumulate the sums. However, we were unable to achieve this due to the compute shaders inherent limitations. The implementation of the spinlock itself proved challenging, notably because of an opaque documentation, requiring multiple attempts to get it working correctly. This implementation is available in Appendix A.

It could be argued that this custom implementation is not directly comparable to CUB's, yet the comparison of their performance remains informative and highlights that the `InteropUnityCUDA` implementation requires significantly less effort for a far better result in terms of computation times.

4.2.4 WavesFDM

This test solves the 2D waves equation d times using the finite difference method. Following the work of Cords and Staadt [2], it calculates the evolution of the height of a 2D domain from an initial filled circle wave at the center of the domain. The aim is to compute the height of a 2D domain at time $t+\Delta t$ based on the heights at times t and $t-\Delta t$. The domain is discretized using uniform grids for times $t+\Delta t$, t, and $t-\Delta t$. A 2D texture array with a resolution of $N \times N$ and size d is used to encode these grids. Denoting the height of the domain at pixel coordinate (i, j) and at time $n \times \Delta t$ by $h^n_{i,j}$, the following explicit scheme is applied for each frame and each pixel of the texture:

$$h^{n+1}_{i,j} = a\left(h^n_{i+1,j} + h^n_{i-1,j} + h^n_{i,j+1} + h^n_{i,j-1}\right) + bh^n_{i,j} - h^{n-1}_{i,j} \tag{1}$$

where a and b are constants defined by the problem properties. Finally, after having determined the new heights, a second kernel is invoked to swap the different height as follows: $h^{n-1}_{i,j} = h^n_{i,j}$ and $h^n_{i,j} = h^{n+1}_{i,j}$, then to store in a buffer the value of $h^{n+1}_{0,0}$, and finally to copy this buffer from GPU to CPU for synchronization. The parameter d is defined to artificially increase the computational load. In such a context, $d = 10$ is used.

4.2.5 Results and Discussion

These tests were run on a Nvidia GeForce RTX 2070 Super with an Intel Core i7-10700. Moreover, in order to get a legitimate view of the performances of each test, they are executed $m_{sample} = 10000$ times across different problem sizes N. Figure 4 shows the execution times

for the CPU, `InteropUnityCUDA` (and so CUDA platform) and Unity's compute shaders using both OpenGL and DirectX 11[8].

Results on CPU show that for lighter workloads, such as in the VectorAdd and Reduce tests, the sequential CPU approach is faster than the GPU for smaller values of N (no later than 10^5) due to lower overhead. However, when the workload is more intensive, as in the WavesFDM test, the GPU already overtakes the CPU even at low N. Globally, for large N and whether using CUDA or compute shaders, the GPU versions consistently outperform the CPU in all tests.

In the GetData test (see Figure 4a), measuring the time to copy a compute buffer from GPU to CPU, OpenGL and CUDA perform roughly equivalently well and they both outperform DirectX 11 for $N > 10^6$, indicating a more efficient data transfer handling.

For the VectorAdd test (see Figure 4b), where two vectors are summed, apart from DirectX 11 which has a reasonable variance, CUDA and OpenGL have a huge variance. The result is a wide oscillation in performance measurement. These results are likely attributable

[8]These are the only two graphics APIs supported by `InteropUnityCUDA`, making them the only ones suitable for comparison.

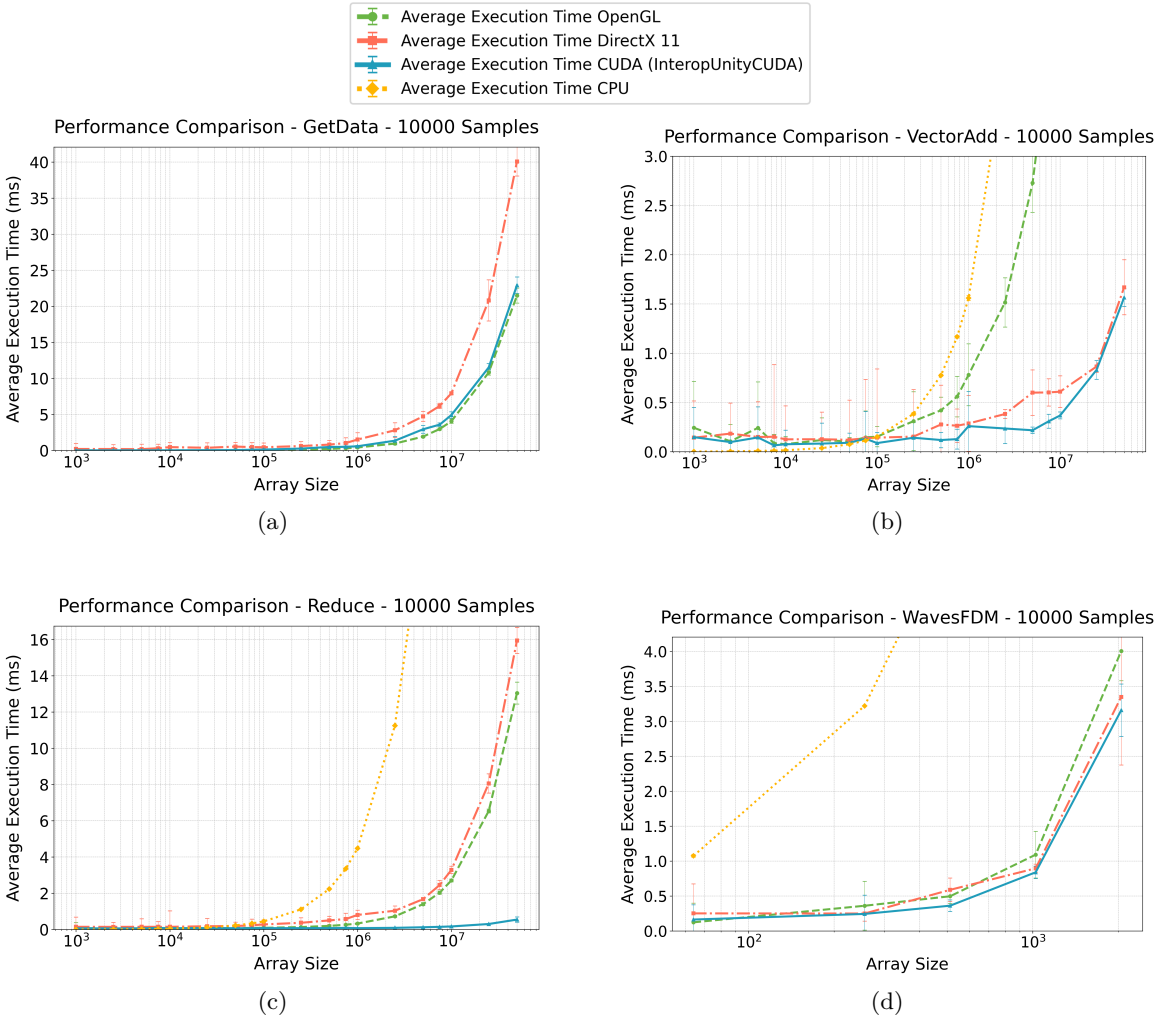

Figure 4: Performance comparison of the four tests using `InteropUnityCUDA` (and so CUDA platform), Unity's compute shaders (with 2 graphics API: DirectX 11 and OpenGL) and CPU with 10000 samples. For Figure 4b the graph has been cropped on the time axis, to provide a better view of DirectX 11 and CUDA performances. Indeed, for $N \in [2 \times 10^6, 5 \times 10^6]$, the execution time for OpenGL rises above 3 ms to reach 26.49 ms.

to the fact that vector addition is a too small operation for the GPU to handle efficiently when $N < 5 \times 10^5$. CUDA gives slightly better results than DirectX 11. What's more, they both perform much better than OpenGL, which for $N > 10^6$ has a duration that increases linearly to 26.49 ms for $N = 5 \times 10^7$.

The Reduce test (see Figure 4c), involving parallel reduction on an array, highlights equivalent performances between CUDA, DirectX 11 and OpenGL for small array. For $N > 10^5$ it highlights CUDA's substantial performance advantage over DirectX 11 and OpenGL. It is observed that the performance of CUDA improves as the array size increases, resulting in execution times that are 20 to 30 times faster than those of the compute shader implementation. This is due to the optimized CUB library in CUDA, which lacks a direct compute shader equivalent.

In the WavesFDM test (see Figure 4d) that solves the 2D wave equation using a finite difference method, CUDA performs marginally better than both graphics API for every N, but the differences are not significant. It can be assumed that this comes from the efficient texture reading in compute shader, and because this problem is memory bound (meaning that there is not enough calculations to see any improvement).

Overall, the results indicate that while OpenGL and DirectX 11 can handle certain levels of computational tasks efficiently, CUDA generally provides equivalent or superior performance, especially for large data sets ($N > 10^5$ seems a correct threshold for these tests and with the used device) and more complex operations. The significant performance benefits of CUDA can be seen in all tests, making it a better choice even on a pure performance criteria. The results underscore the benefits of using `InteropUnityCUDA` for leveraging CUDA's capabilities within Unity for performance-critical applications. Additionally, `InteropUnityCUDA` streamlines development by allowing user to utilize CUDA library functions, such as the reduce in test above, which significantly accelerates the implementation process.

5 Conclusion and Future Works

For future works, adding support for more modern graphics APIs like Vulkan and DirectX 12 would enhance the versatility of `InteropUnityCUDA`. The interoperability between these APIs and CUDA seems based on different techniques[9] that the one used for the two supported graphics API. In consequence, it will lead to completely different implementation. Furthermore, to increase the range of compatible devices, a support to HIP and SYCL could be added, which would further expand device compatibility and ensure broader accessibility and functionality for various user needs, but also imply to rename the plugin.

To conclude, in this paper, `InteropUnityCUDA` addresses several key limitations of Unity's compute shaders, including the lack of debugging tools, limited object-oriented support, absence of parallel libraries, and restricted meta-programming options. Moreover, Section 4 shows that even for basics tasks, when the size of the problem is large, `InteropUnityCUDA` overtakes the Unity alternatives. It also opens up Unity's existing GPGPU computing capabilities. Hence, the authors can only strongly advise you using our tool for addressing complex problems or handling large-scale computations.

Acknowledgements

The first author David Algis has been supported by the Association nationale de recherche et de technologie (ANRT)[10] and by the Pôle Image Magélis. We would like to thank the

[9] More concretely, it looks like that they use a more "modern" interop based function: cudaImportExternalMemory. More details about this function is available at this address `https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXTRES__INTEROP.html`.

[10] Contract number 2021/1157

colleague of David Algis at Studio Nyx, particularly Jeremy Bois. Moreover, we would like to thanks the reviewers for their valuable comments and proposals to this paper.

References

[1] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns". In: *Journal of Parallel and Distributed Computing*. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing 74.12 (Dec. 1, 2014), pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731514001257 (visited on 10/31/2024).

[2] H. Cords and Cords Staadt. "Real-Time Open Water Environments with Interacting Objects". In: *Proceedings of the Fifth Eurographics Conference on Natural Phenomena*. NPH'09 (2009), pp. 35–42. URL: https://dl.acm.org/doi/10.5555/2381692.2381697.

[3] Manuel Costanzo et al. "Comparing Performance and Portability between CUDA and SYCL for Protein Database Search on NVIDIA, AMD, and Intel GPUs". In: *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Oct. 17, 2023, pp. 141–148. DOI: 10.1109/SBAC-PAD59825.2023.00023. arXiv: 2309.09609 [cs]. URL: http://arxiv.org/abs/2309.09609 (visited on 07/04/2024).

[4] Michael Cusumano. "NVIDIA at the Center of the Generative AI Ecosystem—For Now". In: *Communications of the ACM* 67.1 (Jan. 2024), pp. 33–35. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3631537. URL: https://dl.acm.org/doi/10.1145/3631537 (visited on 07/04/2024).

[5] Daniel Damyanov and Zlatko Varbanov. "An Application about Server Communication: Using the Command Pattern on Web API Requests". In: *2024 16th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. 2024 16th International Conference on Electronics, Computers and Artificial Intelligence (ECAI). June 2024, pp. 1–5. DOI: 10.1109/ECAI61503.2024.10607395. URL: https://ieeexplore.ieee.org/abstract/document/10607395 (visited on 12/05/2024).

[6] Robert Dow. *Another Great Quarter—Curb Your Enthusiasm*. Jon Peddie Research. Feb. 27, 2024. URL: https://www.jonpeddie.com/news/another-great-quarter-curb-your-enthusiasm/ (visited on 07/17/2024).

[7] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems". In: *International Journal of Parallel Programming* 46 (Feb. 1, 2018). DOI: 10.1007/s10766-017-0490-5.

[8] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. 2011 International Conference on Parallel Processing. Sept. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45. URL: https://ieeexplore.ieee.org/abstract/document/6047190 (visited on 07/04/2024).

[9] Erich Gamma. *Design Patterns : Elements of Reusable Object-Oriented Software*. Reading, Mass. : Addison-Wesley, 1995. 428 pp. ISBN: 978-0-201-63361-0. URL: http://archive.org/details/designpatternsel00gamm (visited on 08/06/2024).

[10] Yiyang Hao, Hironori Washizaki, and Yoshiaki Fukazawa. “A Third-Party Extension Support Framework Using Patterns”. In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 2015 Asia-Pacific Software Engineering Conference (APSEC). Dec. 2015, pp. 198–205. DOI: 10.1109/APSEC.2015.36. URL: https://ieeexplore.ieee.org/abstract/document/7467301/references#references (visited on 11/20/2024).

[11] Konstantinos G. Kouskouras, Alexander Chatzigeorgiou, and George Stephanides. “Facilitating Software Extension with Design Patterns and Aspect-Oriented Programming”. In: *Journal of Systems and Software*. Selected Papers from the 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, September 7–21, 2006 81.10 (Oct. 1, 2008), pp. 1725–1737. ISSN: 0164-1212. DOI: 10.1016/j.jss.2007.12.807. URL: https://www.sciencedirect.com/science/article/pii/S0164121208000022 (visited on 11/20/2024).

[12] Christian Kröher, Lea Gerling, and Klaus Schmid. “Control Action Types -Patterns of Applied Control for Self-adaptive Systems”. In: *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2023, pp. 32–43. DOI: 10.1109/SEAMS59076.2023.00015. URL: https://ieeexplore.ieee.org/document/10173863 (visited on 12/05/2024).

[13] Ping Kuang et al. “An Improved Calculation System for Phase-Functioned Neural Network and Implementation in Unreal Engine”. In: *Cluster Computing* 22.6 (Nov. 1, 2019), pp. 15505–15516. ISSN: 1573-7543. DOI: 10.1007/s10586-018-2671-4. URL: https://doi.org/10.1007/s10586-018-2671-4 (visited on 10/31/2024).

[14] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., June 2012. 432 pp. ISBN: 978-0-12-391443-9.

[15] Carlos Pérez et al. *A Comparative Analysis of Energy Consumption Between The Widespread Unreal and Unity Video Game Engines*. May 23, 2024. arXiv: 2402.06346 [cs]. URL: http://arxiv.org/abs/2402.06346 (visited on 08/09/2024). Pre-published.

[16] Dinei A. Rockenbach et al. “GSParLib: A Multi-Level Programming Interface Unifying OpenCL and CUDA for Expressing Stream and Data Parallelism”. In: *Computer Standards & Interfaces* 92 (Mar. 1, 2025), p. 103922. ISSN: 0920-5489. DOI: 10.1016/j.csi.2024.103922. URL: https://www.sciencedirect.com/science/article/pii/S0920548924000916 (visited on 10/31/2024).

[17] Ching-Lung Su et al. “Overview and Comparison of OpenCL and CUDA Technology for GPGPU”. In: *2012 IEEE Asia Pacific Conference on Circuits and Systems*. 2012 IEEE Asia Pacific Conference on Circuits and Systems. Dec. 2012, pp. 448–451. DOI: 10.1109/APCCAS.2012.6419068. URL: https://ieeexplore.ieee.org/abstract/document/6419068 (visited on 07/04/2024).

[18] SlashData Team. *Did You Know That 60% of Game Developers Use Game Engines?* SlashData. Dec. 2, 2022. URL: https://www.slashdata.co/post/did-you-know-that-60-of-game-developers-use-game-engines (visited on 08/09/2024).

[19] Yongfeng Zhao et al. “Architecture Design of a Distributed Workflow Platform Based on Microservice Architecture”. In: 12704 (May 1, 2023), p. 127040L. DOI: 10.1117/12.2680217. URL: https://ui.adsabs.harvard.edu/abs/2023SPIE12704E..0LZ (visited on 12/05/2024).

A Compute Shader Code for Parallel Reduction

```
#pragma kernel Reduce

#define THREADS_PER_GROUP 1024

StructuredBuffer<float> arrayToSum;
globallycoherent RWStructuredBuffer<float> resultReduce;

RWStructuredBuffer<int> spinlock;

int sizeArrayToSum;
groupshared float partialSums[THREADS_PER_GROUP];

[numthreads(THREADS_PER_GROUP,1,1)]
void Reduce(uint tid : SV_GroupIndex, uint3 groupIdx : SV_GroupID)
{
    const uint i = groupIdx.x * THREADS_PER_GROUP + tid;
    if (i < sizeArrayToSum)
        partialSums[tid] = arrayToSum[i];
    else
        partialSums[tid] = 0;

    GroupMemoryBarrierWithGroupSync();

    for (uint s = THREADS_PER_GROUP / 2; s > 0; s >>= 1)
    {
        if (tid < s)
            partialSums[tid] += partialSums[tid + s];

        GroupMemoryBarrierWithGroupSync();
    }

    if (tid == 0)
    {
        int old = 1;
        [allow_uav_condition]
        while (old != 0)
            InterlockedCompareExchange(spinlock[0], 0, 1, old);

        resultReduce[0] += partialSums[0];
        InterlockedExchange(spinlock[0], 0, old);
    }
}
```

Listing 1: Compute shader code used for reduce in Section 4.

B A Full Example of Custom Action Creation

This example implements a custom and simple `Action` to use `InteropUnityCUDA`. This action retrieves a texture, created from Unity; then this action writes into this texture through a CUDA kernel. The steps are as follows:

1. Create a new library, named for instance `ActionLib`[11].
2. Include `PluginInteropUnityCuda` to `ActionLib`.
3. A file `my_action.h` with a class `MyAction` should be created by deriving from `Action` in `PluginInteropUnityCuda`. This class contains a constructor and the three methods

[11] All names used in the remainder of this section are examples for pedagogical purposes only.

that should be overridden. In addition, an external method to create this action from Unity is provided.

```
// my_action.h
#include "action.h"
#include "cuda_include.h"
#include "sample_kernels.cuh"
#include "texture.h"
#include "unityPlugin.h"

class MyAction : public Action
{
    public:
    // Initialization of MyAction with required parameters
    MyAction(void *textureUnityPtr, int width, int height) : Action()
    {
        // Create a Texture object that can be registered/mapped to CUDA
        _texture = CreateTextureInterop(textureUnityPtr, width, height, 1);
    }

    int Start() override
    {
        _texture->registerTextureInCUDA();
        _texture->mapTextureToSurfaceObject();
        return 0;
    }

    int Update() override
    {
        // Example: kernel invocation from a sample project
        kernelCallerWriteTexture(_texture->getDimGrid(),
                                 _texture->getDimBlock(),
                                 _texture->getSurfaceObject(), GetTimeInterop()
                                     ,
                                 _texture->getWidth(), _texture->getHeight());
        return 0;
    }

    int OnDestroy() override
    {
        _texture->unmapTextureToSurfaceObject();
        _texture->unregisterTextureInCUDA();
        return 0;
    }

    private:
    // Class attributes used in MyAction
    Texture *_texture;
};

extern "C"
{

    UNITY_INTERFACE_EXPORT MyAction *UNITY_INTERFACE_API
    createMyAction(void *textureUnityPtr, int resolutionTexture)
    {
        return (new MyAction(textureUnityPtr, resolutionTexture));
    }
}
```

Listing 2: Implementation of `my_action.h`

To integrate the custom action into Unity and register it with `PluginInteropUnityCUDA`, the following steps should be completed:

1. A first C# class `MyActionUnity` should be implemented by deriving from `ActionUnity` in `ActionUnity.cs`. This class serves as an intermediary, holding the necessary information for the native code implementation (*e.g.* `MyAction`).

2. The function defined in step 4 `createMyAction` should be imported.

3. The member `_actionPtr` must be assigned the return value of the imported function (*e.g.*, a pointer to `MyAction`).

```
// MyActionUnity.cs
using System;
using System.Runtime.InteropServices;
using UnityEngine;

public class MyActionUnity : ActionUnity
{
    // Define the name of the native library
    const string _myDllName = "ActionLib";

    [DllImport(_myDllName)]
    private static extern IntPtr createMyAction(IntPtr texturePtr, int width,
        int height);


    // Constructor to initialize MyActionUnity
    public ActionUnitySampleTexture(Texture texture) : base()
    {
        _actionPtr = createMyAction(texture.GetNativeTexturePtr(), texture.
            width, texture.height);
    }
}
```

Listing 3: Implementation of `MyActionUnity.cs`

4. A second C# class `MyInteropHandler` should be created by deriving from `InteropHandler` in `InteropHandler.cs`. This class is responsible for creating the action `MyActionUnity` and invoking the corresponding functions of `PluginInteropUnityCUDA` to manage it.

5. The method `InitializeActions()` must be overridden to initialize the actions in `PluginInteropUnityCUDA`. The action should be constructed, registered using the `RegisterActionUnity` function, and started by invoking the `CallActionStart` method. Analogously, the methods `CallUpdateActions` and `CallOnDestroy` should call the associated function for `MyAction`.

6. The event function of `InteropUnityCUDA` (`InitializeInteropHandler`, `UpdateInteropHandler` and `OnDestroyInteropHandler`) should be called manually. This let more controls for user on `InteropUnityCUDA` event calls.

```
// MyInteropHandler.cs
using System.Runtime.InteropServices;
using UnityEngine;

class MyInteropHandler:InteropHandler
{
```

```
7
8       // the id which will be use in registration of my action
9       private const string _MyActionName = "myAction";
10      // contains the texture we want to read/write through CUDA
11      Texture2D _texture;
12
13
14      public void Start()
15      {
16          _texture = new Texture2D(1024, 1024, TextureFormat.RGBAFloat, false,
                true);
17          _texture.Apply();
18
19          InitializeInteropHandler();
20      }
21
22      public void Update()
23      {
24          UpdateInteropHandler();
25      }
26
27      public void OnDestroy()
28      {
29          OnDestroyInteropHandler();
30      }
31
32      protected override void InitializeActions()
33      {
34          base.InitializeActions();
35          // Construct an action using the specified parameters
36          MyAction myAction = new MyAction(_texture);
37          // Register the action in PluginInteropUnityCUDA
38          RegisterActionUnity(myAction, MyActionName);
39          // Start the registered action
40          CallActionStart(MyActionName);
41      }
42
43      protected override void CallUpdateActions()
44      {
45          // Invoke the overridden Update method
46          CallActionUpdate(MyActionName);
47      }
48
49      protected override void CallOnDestroy()
50      {
51          // Invoke the overridden OnDestroy method
52          CallActionOnDestroy(MyActionName);
53      }
54  }
```

Listing 4: Implementation of `MyInteropHandler.cs`