

This is the peer reviewed version of the following article:

A Taxonomy of Modern GPGPU Programming Methods: On the Benefits of a Unified Specification / Capodieci, N.; Cavicchioli, R.; Marongiu, A.. - In: IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. - ISSN 0278-0070. - 41:6(2021), pp. 1679-N/A. [10.1109/TCAD.2021.3082863]

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

11/01/2026 19:14

This is the accepted manuscript of:

Capodieci, Nicola, Roberto Cavicchioli, and Andrea Marongiu. "A Taxonomy of Modern GPGPU Programming Methods: On the Benefits of a Unified Specification." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2021).

© IEEE 2021. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The definitive Version of Record was published in <https://doi.org/10.1109/TCAD.2021.3082863>

A Taxonomy of Modern GPGPU Programming Methods: On the Benefits of a Unified Specification

Nicola Capodiecì, *Member, IEEE*, Roberto Cavicchioli, and Andrea Marongiu, *Member, IEEE*

Abstract—Several Application Programming Interfaces (APIs) and frameworks have been proposed to simplify the development of General-Purpose GPU (GPGPU) applications. GPGPU application development typically involves specific customization for the target operating systems and hardware devices. The effort to port applications from one API to the other (or to develop multi-target applications) is complicated by the availability of a plethora of specifications, which in essence offers very similar underlying functionality. In this work we provide an in-depth study of six state-of-the-art GPGPU APIs. From these we derive a taxonomy of the common semantics and propose a unified specification. We describe a methodology to translate this unified specification into different target APIs. This simplifies cross-platform application development and provides a clean framework for benchmarking. Our proposed unified specification is called GUST (GPGPU Unified Specification and Translation) and it captures common functionality found in compute-only APIs (e.g., CUDA and OpenCL), in the compute pipeline of traditional graphics-oriented APIs (e.g., OpenGL and Direct3D11) and in last-generation bare-metal APIs (e.g., Vulkan and Direct3D12). The proposed translation methodology solves differences between specific APIs in a transparent manner, without hiding available tuning knobs for compute kernel optimizations and fostering best programming practices in a simple manner.

Index Terms—GPGPU, Parallel Programming Tools

I. INTRODUCTION

Architectural heterogeneity is becoming the reference hardware design paradigm in all computing domains, as it effectively addresses the energy and thermal walls implied by CMOS technology scaling. General Purpose Graphics Processing Units (GPGPU) represent probably the most widespread example of such design paradigm, and GPGPU programming is nowadays a very common paradigm to efficiently execute large data-parallel workloads. The massively parallel compute hardware of modern GPUs is no longer designed for graphics-related tasks only, as it was originally. By offloading the execution of compute-intensive, data-parallel code *kernels* from general-purpose Central Processing Units (CPU) on top of such hardware resources it is possible to hit unprecedented performance-per-watt targets. GPGPU computing has been successfully applied to a variety of performance-demanding computing tasks such as large scale simulations [1, 2], VLSI placement and gate sizing [3, 4], signal processing [5], Machine Learning (ML) [6] and everything else that requires high throughput performance over consistently large sets of input data [7]. Moreover, we are witnessing a trend towards the integration of increasingly powerful GPGPU compute

capabilities also at the System-on-Chip (SoC) scale, which enables the execution of machine learning (ML) and Artificial Intelligence (AI) workloads also on top of high-end embedded systems [8].

In order to reach a large variety of users, during the course of the years both the research community and the industry have proposed many high-level languages for simplifying access to such technologies: by providing explicit hooks to the compute pipeline of traditional graphics-oriented APIs (e.g. Khronos OpenGL and Microsoft Direct Compute over Direct3D), to the release of compute-specific GPU-Accelerated programming models such as NVIDIA CUDA and Khronos OpenCL. All these APIs aim at hiding the complexity of dealing with details specific to the hardware or OS-level drivers. However, while their library-based interface indeed exposes hardware functionality with some degree of abstraction, it is widely agreed that this style of programming is still very involved and, overall, low-level. This has motivated over the last decade a lot of research efforts aimed at further raising the level of abstraction of GPU programming methods.

An application developer chooses a target API based on a number of factors: (i) dependencies from legacy code in a project; (ii) time-to-market constraints and the expertise of the developers; (iii) the need to develop a product meant for multi-platform execution or to port an existing product from one platform to another. Once a program is written with a given API, the effort for porting it on a different API and/or to a different platform is typically not at all trivial. Thus, over the years the research problem of simplifying GPU programming has also tackled the specific issue that porting applications from one GPU system to another, or developing multi-target applications, is complicated by the availability of a large number of low-level APIs. Moving from the observation that these APIs ultimately offer very similar underlying semantics, a number of research and industrial efforts have addressed the need to derive some sort of unified specification/language capable of working across multiple APIs. These approaches and associated tools have proven effective at easing application porting to various GPU HW/SW platforms, but they have so far been very limited and only targeted a few specific APIs.

In this paper we provide an in-depth study of the anatomy of six state-of-the-art APIs for GPGPU programming: OpenGL Compute Shaders, Direct Compute over Direct3D11, OpenCL, CUDA and next-generation bare-metal APIs Vulkan and Direct3D12. Out of this study we derive a taxonomy of the semantics supported across the considered APIs, highlighting how the low-level services underlying virtually every available API for GPU development largely overlap in functionality.

From there, we discuss and explore the feasibility and the benefits of a unified specification that (i) captures common semantics to any GPGPU-related API consisting of host processor (CPU) and device (GPU) commands; (ii) defines a simple and comprehensive syntax for their deployment. To carry out an early quantitative assessment of the feasibility of such approach, we discuss a proof-of-concept implementation of a unified language front-end, and a methodology to transparently instantiate any of the targeted APIs as language back-ends. The proposed process is literally as simple as recompiling a program for a different target, as the methodology infers the required information from the unified functions and handles internally the differences among APIs. The code produced in this way can then be translated into an executable program using the (unmodified) native compilation flow for any target API. Finally, we provide an evaluation of the core functionality of the provided proof-of-concept implementation, discussing benefits, limitations and how to address them.

II. RELATED WORK

Graphic Processing Units were born as hardware components designed with the specific purpose to generate images for visual display. However, since the beginning of the 21st century programmable GPU pipelines – initially developed for better graphics processing – were found to fit scientific computing needs well, just like the matrix/array data used to formulate such problems. Since then, efforts to use GPUs as general-purpose processors have led to a radical transformation of both their hardware (fully programmable compute units, support for floating-point operations) and their programming methods, which evolved from cumbersome, complete reformulations of computational problems in terms of graphics primitives in the infancy of GPGPU computing, (OpenGL, DirectX) to the advent of truly general-purpose programming languages and APIs (OpenCL, CUDA).

Today, efficiently implementing applications in which a CPU *host* offloads data to a GPU capable of massively parallel computations is not a trivial task. Programming paradigms such as CUDA or OpenCL have a very involved coding style, which require “being fluent” in low-level *device*-side languages and related compilation procedures, for interacting with the *host*. Over the course of the years a plethora of approaches have been proposed to raise the level of abstraction in GPGPU programming, which typically build on top of low-level programming models and APIs.

Efforts towards simplifying heterogeneous systems programming can be roughly grouped in two macro-categories, namely (1) *libraries of building blocks* and (2) *language extensions*.

A **building block** is typically referred to as an easy-to-use abstraction of mathematical operations commonly adopted across many application domains in parallel calculus. Such operations are usually performed on large input data sets, typically organized in non-trivial data structures such as sparse matrices or graphs. Ease of programming is achieved by exposing an interface in which such function calls are able to hide one or more of the following aspects [9]: writing

and compiling compute kernels, organizing the relevant data structures in a GPU friendly manner (i.e., data strides and access pattern should trigger coalesced GPU memory accesses) and memory management in heterogeneous address spaces. Approaches that build on top of such libraries to further raise the level of abstraction have also been explored [10], including SkePU [11], Raja [12], HPX [13] and Kokkos [14].

Language extensions is another well-explored approach to raising the level of abstraction in GPGPU programming. The solutions available in research papers or in real-life development tools pertain to two categories: (i) *true language extensions*, where standard programming languages and associated compilers have been augmented with new datatypes and keywords to specify parallel execution on a GPGPU; (ii) *compiler directives*, where the information about where and how to modify the program for parallelization is more pragmatically provided via code annotations, without interfering with the original type system and semantics of the used language. In this category OpenMP is probably the most significant and well-known example. The OpenMP specifications have evolved in the latest releases to simplify the orchestration of computation between host and GPU-like accelerators. OpenACC (Open ACCelerator)¹ adopts the same philosophy and coding style, and was explicitly designed to simplify the porting of High Performance Compute applications to a wide-variety of heterogeneous hardware. SYCL², from the Khronos group, is another notable attempt at raising the level of abstraction for heterogeneous systems programming. Attempts to quantify the benefits of the more abstract coding style of such approaches have been made [15], reporting on average about 6.7x less programming effort when using OpenACC compared to OpenCL, or 3.6x less programming effort when using OpenMP compared to OpenCL, and about 3.1x less programming effort when using OpenMP compared to CUDA. However, the simplified coding style typically comes at the cost of some performance loss, which is very much implementation-dependent [15].

Besides the need for raising the level of abstraction for GPU programming, the research community has also addressed the problem that porting applications from one GPU system to another, or developing multi-target applications, is complicated by the availability of a large number of APIs. Indeed, besides CUDA and OpenCL several other APIs have evolved to become widely adopted standards for graphics rendering (OpenGL, OpenGL ES, Direct3D) and general-purpose computing (Vulkan, Direct3D12). Despite its cross-platform nature, OpenCL is not widely supported: a recent analysis targeting Android devices currently available in the market shows that OpenCL support is only available in 32% of these products [16]. This is, incidentally, the reason why a GPU backend for TensorFlow Lite has been only available as an OpenGL ES implementation until very recently³. A number of research and industrial efforts have addressed the need to derive some sort of unified specification/language, capable to work across

¹<https://www.openacc.org/>

²<https://www.khronos.org/sycl/>

³An OpenCL implementation has been announced from TensorFlow Lite developers: https://www.tensorflow.org/lite/performance/gpu_advanced

multiple GPU APIs. *RenderScript* [17], an Android-specific graphics and compute API, features implementations targeting OpenCL, OpenGL and standard CPU as a backend [18].

Intel's newly released *oneAPI* specification⁴ offers an unified programming model aimed at delivering a common developer experience across accelerator architectures.

Other authors have proposed the idea of unified semantics among different GPGPU APIs targeting OpenCL and CUDA as compilation backends [19], and proposing a framework offering a unified specification with easy-to-use abstractions for managing compute and data resources. Narrowing the application target to *tensor* operations, other approaches have investigated an abstraction layer for deep neural networks, capable of generating CUDA, OpenCL and Vulkan code [20].

Other approaches have proposed simple languages for specifying *device* code and a compiler generating highly-optimized CUDA and OpenCL kernels [21] [22]. Similarly, OpenMP and OpenACC implementations for GPGPUs exist [23], targeting CUDA and OpenCL as backends [24]. C++ AMP also features GPU implementations that rely on CUDA, DirectCompute or OpenCL [25] for low-level execution.

All these approaches and tools have proven effective at easing the task of porting applications to various GPU hardware/software platforms; however, they have so far only targeted a few specific APIs. In this paper, we aim at assessing the benefits of extending this type of methodology to all of the most widespread GPU APIs, analyzed in the following.

III. A TAXONOMY OF MODERN GPU API SEMANTICS

In the remainder of this paper we delve into studying the anatomy of six selected low-level APIs. Our aim is to understand which standard features are common to all these APIs, with an interest in exploring whether some sort of unified specification and semantics could be derived, and the benefits this could bring to GPGPU developers.

A. API selection

Figure 1 shows the main language abstraction of six widespread APIs for handling *graphics* and *compute pipelines* of modern GPGPUs. The figure highlights the fact that all the APIs provide features to express general-purpose computation for execution on the GPU (*compute pipeline*). The focus of our study is on these latter features.

a) CUDA: – CUDA (Compute Unified Device Architecture) is a widely adopted API and programming model for GPGPU computing. It is a NVIDIA proprietary standard firstly released in 2007. Like all the other APIs for programmable GPUs, writing a CUDA application implies describing the interaction between host (CPU) and one or more device accelerators (GPUs). On the *host* side, the programmer can elect to use the *CUDA Driver API* or the *CUDA Runtime API*, which are mutually exclusive in their usage. The *CUDA Runtime API* eases *host* code development by providing implicit context initialization and a simplified syntax for launching compute kernels compared to the *CUDA Driver API*. The *CUDA Driver*

API offers a higher degree of control over these aspects, at the cost of a more involved coding style. On the *device* side, parallel computations are described as a grid of *threads* grouped in *blocks*, adopting a C++ like syntax enriched with specific keywords used for thread indexing, dynamic kernel invocations, synchronizations etc.

b) OpenCL: – OpenCL (Open Computing Language)⁵ is an industry standard for heterogeneous computing for massively parallel architectures created by Apple in 2009 but nowadays maintained by the Khronos Group. OpenCL presents two major differences compared to CUDA. First, its open nature allows to target generic devices other than NVIDIA GPUs. Second, unlike CUDA, OpenCL provides a more generic API that creates an abstraction layer suitable for a variety of compute accelerators (e.g: FPGA, DSPs, multicore CPUs besides GPUs). This makes developing an OpenCL application slightly harder compared to a CUDA equivalent. Just like CUDA, an OpenCL program is divided between *host* and *device* code. On the *device*-side, the OpenCL specifications allow the developer to describe parallel computations in a high level language (the OpenCL Kernel language) derived from C/C++ and a standard intermediate binary format (SPIR/SPIR-V⁶, Standard Portable Intermediate Representation). Performance-wise, it has been shown that OpenCL and CUDA behave quite similarly [26], with small performance gaps given by the different compilation heuristics employed by the different drivers. Other authors investigated the portability issues of specific kernels from CUDA to OpenCL in [27].

Although OpenCL and CUDA can share buffers to graphic contexts in a seamless manner, they both were specifically designed to be compute-only APIs (i.e., to describe general-purpose computation).

c) OpenGL: – Among the APIs for real-time graphics rendering, the most representative example is probably OpenGL (Open Graphic Language), a cross-platform standard released in 1992 by Silicon Graphics and currently maintained by Khronos. OpenGL went through a significant evolution over the course of the years. The biggest evolutionary step coincides with the introduction of *programmable shaders* (to match the hardware evolution from fixed-functionality graphics co-processors to programmable GPUs). Since version 4.3, OpenGL includes the concept of *compute shader*, namely a pipeline stage targeting the execution of general-purpose compute code, which logically runs alongside rendering stages (e.g., vertex and fragment processing). OpenGL features *device vendor extensions*, which allow for the development of hardware vendor-specific functionalities not included in the OpenGL standard, and that can be exposed to the application developer. The language for device code development is called GLSL (GL Shading Language). In mid 2003 the Khronos group proposed OpenGLES, a subset of modern OpenGL functionalities specifically designed to run on mobile and embedded systems, hence becoming the most widespread GPU API in history. Starting from version 3.1, OpenGLES fully supports compute shaders.

⁴<https://www.oneapi.com/>

⁵<https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf>

⁶<https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf>

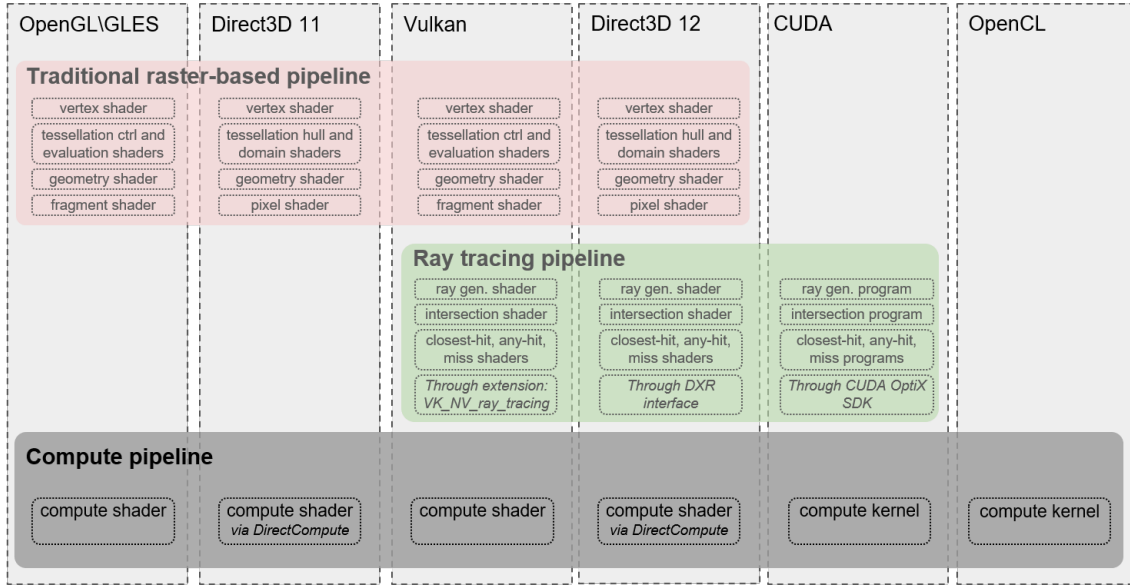


Fig. 1: Summary of GPU APIs programmable functionality set.

d) **Direct3D**: – Direct3D is a GPU API from Microsoft for graphics rendering on certified Windows platforms. It can be seen as the Windows-only OpenGL counterpart and, compared to the Khronos standard, it exposes a thinner abstraction layer. Moreover, it does not feature the equivalent of OpenGL extensions. The Direct3D equivalent for a GL *compute shader* exploits the *DirectCompute* technology. Device code within a Direct3D program is written using HLSL (High Level Shading Language). In this work we examine two major Direct3D releases: Direct3D11 and the recently released Direct3D12. The differences between the Direct3D11 and Direct3D12 are major, to the point that Direct3D 12 cannot be considered the Windows counterpart of OpenGL anymore, but rather that of another recently released Khronos Standard; *Vulkan*.

e) **Vulkan**: – *Vulkan* is a recent open and cross-platform standard for GPU programming. It was proposed and now maintained by the Khronos group. It is described as both a graphics and a compute API as the Vulkan programming model is agnostic to which of these two pipelines will be mostly used in an application. Vulkan is getting widely supported across different hardware and operating systems. In a recent announcement, Google stated that a working Vulkan implementation will be required on all 64-bit devices running Android Q. This suggests that Vulkan has been designed to be the successor of both OpenGL and OpenGL ES.

Vulkan and Direct3D12 differ from the rest of the APIs covered here in the way they handle *driver* interactions. Rather than relying on a constant *driver* interaction for error checking, hidden optimizations and other operations (for which the application developer has little to no control), Vulkan and Direct3D12 are much more explicit in the way in which commands are submitted to the GPU driver. In such model, all the low-level details of computation offloading (including synchronization, error checking, etc.) are exposed to the programmer, which has to take care of these aspects manually.

B. A note on the adopted terminology

In the remainder of this paper we refer to CUDA, OpenCL, OpenGL and Direct3D11 as the *Traditional APIs*, whereas Vulkan and Direct3D12 are referred to as the *Command List-based APIs*. Different APIs have different names for the same software artifacts, hence we establish here common terminology to avoid ambiguities.

First, we define the concept of *context*. A *context* represents the abstraction of an API-specific resource manager. A *context* has to be created once at the beginning of the application and *released* upon completion. A *context* also represents the interface in which GPU-related software artifacts might be read and modified.

Concerning the code to be executed on the GPU, the terms *compute shader*, *compute kernel* or *compute program* are indistinctly chosen, and sometimes abbreviated in *shader*, *kernel* or *program*. When a *shader*, *program* or *kernel* is launched on the GPU we refer to this operation as *invocation* or *dispatch* indistinctly.

A *kernel* has input and output data in the form of *arguments* and *symbols*. To understand the difference, let us consider the following CUDA⁷ kernel function signature:

```
__global__ void vector_add(const int* A,
                          const int* B, int* C, int N);
```

The *device* function `vector_add` takes four inputs: two pointers A and B to the input vectors (also called *data buffers*), a pointer C to the output vector, and the integer N, that represents the vectors' size. While OpenCL works in a very similar manner to this CUDA example, other APIs not only adopt different terminology to distinguish between pointers to buffers and constant values, but they also need special operations for binding inputs to one or more specific programs. In our chosen terminology, A, B and C are called *arguments* or *resources* and N is called a *symbol*. *Arguments*

⁷The concepts illustrated are not specific to CUDA.

TABLE I: API specific Host-side operations for the considered APIs

Operation - API	CUDA Driver API	OpenCL	OpenGL Compute Shaders	DirectCompute over Direct3D 11	Vulkan	DirectCompute over Direct3D 12
Context Creation	cuInit cuDeviceGet cuCtxCreate	clCreateContext FromType	<i>Platform specific sys. calls</i>	D3D11CreateDevice	vkCreateInstance vkCreateDevice	D3D12CreateDevice
Kernel Compilation	nVRTCCreateProgram nVRTCCompileProgram nVRTCGetPTX cuModuleLoadDataEx cuModuleGetFunction	clCreateProgram WithSource clBuildProgram clCreateKernel	glCreateShader glCompileShader glCreateProgram glAttachShader glLinkProgram	D3DCompileFromFile ID3D11Device::CreateComputeShader	vkCreateShaderModule	D3DCompileFromFile ID3D12Device::CreateComputeShader
Arguments allocations	cuMemAllocHost cuMemAlloc	clCreateBuffer clEnqueueMapBuffer	glGenBuffer glBindBuffer (SSBO)	ID3D11Device::createBuffer, SRV or UAV	vkCreateBuffer	ID3D12Device::Create*Resource
Symbols	const values in kernel invocation	const values in kernel invocation	OpenGL uniforms	Constant Buffers	Push Constants	Constant Buffers
Buffer Synchron.	cuMemCpy*Async	clEnqueue*Buffer	glBufferData glMap/Unmap & memcpy	D3D11_MAPPED_SUBRESOURCE map/unmap & memcpy	vkCmdCopy*	ID3D12GraphicsCommandList::CopyBufferRegion
Arguments binding	in kernel invocation cuLaunchKernel	clSetKernelArg	glBindBufferBase	ID3D11DeviceContext1::CSSetShaderResources CSSetUnorderedAccess Views	Descriptor Sets and Layouts	Root Signature Descr.
Launch Config.	in kernel invocation cuLaunchKernel	in kernel invocation clEnqueueNDRangeKernel	in dispatch call glDispatchCompute GroupSize (ARB ext.) glDispatchCompute	in dispatch call ID3D11DeviceContext1::Dispatch	Specialization Consts. dispatch call vkCmdDispatch	In dispatch Call ID3D12GraphicsCommandList::Dispatch
Device Wait For Idle (WFI)	cuCtxSynchronize	clFlush clFinish	glFlush glFinish	blocking wait on a ID3D11Query	vkDeviceWaitIdle	blocking wait on a ID3D12Fence

can be *read-only*, *write-only* or might allow both operations. *Symbols* represent *read-only* data to the *device* (only written by the *host*). Every API has to bind sets of *arguments* and *symbols* to one or more kernels: we refer to this operation as *setting the kernel layout*.

Also related to *kernels*, a program invocation needs a *launch configuration*, which logically describes the degree of parallelism in which the work must be computed over a *grid* of parallel *threads*. According to the dimension of the compute program data set, such grid might be designed to expand along 1, 2 or 3 dimensions. Every API therefore exposes a way to dispatch a *kernel* using a specific configuration of *threads* and *groups of threads* over each dimension. To this respect, we adopt the CUDA terminology: a GPU computation is divided into *threads* and threads are grouped into *blocks* or *groups*.

Buffers hosting data passed via function *arguments* might be allocated both on the *host* side and on the *device* side. When a *host* pointer has a corresponding *device* pointer, communication might be more or less explicitly *synchronized*. This implies a *device-to-host* or a *host-to-device* copy. Under the *Unified Memory Model*, a paradigm aimed at simplifying heterogeneous programming by hiding the existence of distinct address spaces (CPU and GPU), a *buffer* pointer can be flagged to allow unified access from both the *host* and *device*, without requiring explicit copies/synchronization [28]. We further discuss *unified memory* in Section III-C3.

Specific to the *Command List-Based APIs* we define the terms *Pipeline State Object* (PSO) and *command buffer*. A PSO is a pre-compiled description of settings for specific kernels, and this includes *arguments* and *symbols bindings* as well as *launch configurations*. A *command buffer* is a pre-recorded set of commands (PSO selection, kernel invocations and data buffer movements). *Command List-Based APIs* require that low-level offload operations are explicitly handled in advance to minimize driver interactions during the runtime execution of the applications (see Section III-C2).

C. API Constructs for Host Code

In this section we isolate and study the “lowest common denominator” for *host-side* operations performed in all the GPU APIs, from *context initialization* to *kernel dispatch*.

1) *Traditional APIs*: Besides *context* creation, in traditional APIs the developer has to take care the following aspects: *kernel* compilation, allocation of *buffers* for *arguments*, definition of *symbols*, buffer synchronization (i.e., data transfers), arguments binding (kernel layout) and the actual dispatch computations with their related launch configurations. The first four columns of Table I summarize how all the considered *traditional* APIs perform these operations.

Table I maps each *host-side* operation (from *context* creation to dispatch call) and other relevant software constructs to respective API-specific terminology and function calls. For CUDA, we consider the *Driver API* rather than the *Runtime API*, as – functionally speaking – the latter is merely a C++ wrapper around the C-based *Driver API*. For all these APIs, a *kernel* source code can be extracted from a file or might be hosted in a *string* variable.

Context creation exploits API-specific function calls, OpenGL being the only exception. The OpenGL specification states that *context* creation is not regulated by the Khronos standard: this implies calling platform-specific system calls as opposed to regular `gl*` function calls. Allocation functions regulate memory allocations for both *host* and *device* data, according to configurable access flags. A *device-side* buffer is stored as an API-specific data structure, like `CUDevicePtr` for CUDA or `cl_mem` for OpenCL. In OpenGL, generic *device-visible* data are *Shader Storage Buffer Objects*⁸ (SSBOs). In Direct Compute on Direct3D11, *Shader Resource Views* (SRVs) are used for read-only *device* side buffers and *Unordered Access Views* (UAVs) for those buffers that need to be read back by the *host*.

⁸https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

Symbols are trivially implemented for both CUDA and OpenCL; the OpenGL equivalent are called *uniforms*, whereas on Direct Compute-based APIs the functional equivalent are *Constant Buffers* (CBs).

Other functionality shown in Table I is straightforward to address, with the exception of the *launch configuration* part, which requires additional remarks. While none of the APIs has any limitation in dynamically specifying the number of *blocks* for each *grid* dimension, problems arise if the user also wants to dynamically specify the number of *threads* within a *block*. For CUDA and OpenCL such functionality is trivially implemented; for OpenGL, this is achieved by invoking a *compute shader* with `glDispatchComputeGroupSizeARB` instead of `glDispatchCompute` function. For Direct Compute over Direct3D11, dynamically specifying variable *block* sizes is not supported: a possible workaround for this is to create the *compute shader* at runtime, specifying the *block* size with the HLSL `numthreads` builtin macro and deferring the *shader* compilation until just before its first dispatch call.

2) *Command List-Based APIs*: *Command list-based* APIs in addition allow (i) to mark specific code sections for constructing PSOs and *command buffers* and; (ii) the actual submission of the *command buffer(s)* to the GPU. The last two columns of Table I show the operations and primitives required from *context* creation to *command list* submission for the Direct3D 12 and the Vulkan API. Focusing on Direct3D-based APIs, the biggest difference in version 12, compared to version 11 is the necessity of pre-compiling a PSO and populating a *command list* before the actual dispatch call. Other than that, *device* code is compiled and described in the same manner. Copy and compute commands are therefore deferred, as all the function calls from the `ID3D12GraphicsCommandList` interface are only allowed during a *command buffer* recording stage. The Vulkan equivalents of copy and dispatch operations use the `vkCmd*` prefix. These functions can also only be invoked during the recording of a *command buffer* (`VkCommandBuffer` data type). This significantly adds to the complexity of writing a general-purpose program.

Vulkan *device* code is written using a low-level intermediate representation (SPIR-V), which can be compiled via the `vkCreateShaderModule` function. Third-party toolchains can be used to allow the use of higher level languages (such as HLSL or GLSL) augmented with specific Vulkan extensions. In the following, we assume that an additional compilation step is put in place to allow the description of *device-side* computations in a Vulkan using GLSL. This allows to treat Vulkan and OpenGL *device* code as conceptually equivalent for our purposes. We will further elaborate on this assumption.

3) *Unified memory*: Unified memory models are present in all the examined APIs. CUDA UVM (Unified Virtual Memory) describes a model in which specific *host-visible* pointers are managed by the CUDA *driver* through an on-demand page migration mechanism between the CPU and the GPU address spaces [29]. Recent CUDA versions allow the developer to suggest to the GPU *driver* the actual residency aspects of each

allocations (`cudaMemAdvise`) and even to control memory pages prefetching (`cudaMemPrefetch*`).

In recent OpenCL versions, different implementations of the SVM (Shared Virtual Memory) abstraction allow the developer to exploit different granularity for resources shared by the CPU and the GPU. More specifically, starting from OpenCL version 2.0 three types of SVM allocations are available: (i) *coarse-grained buffer SVM*: entire buffers that reside in *device-local* memory might be accessed by the CPU; (ii) *fine-grained buffer SVM*: individual loads and stores within OpenCL buffers residing in *device-local* memory are shared between CPU and GPU; (iii) *fine-grained system SVM*: sharing between CPU and GPU occurs at the granularity of individual loads/stores within *host* memory allocations.

Both CUDA and OpenCL in their most recent versions allow the developer to allocate buffers in CPU and GPU space explicitly (`cudaMallocManaged` in CUDA and `clSVMAlloc` in OpenCL) or implicitly, i.e. using regular *host-side* allocation functions (e.g. `new` and `malloc`). CUDA and OpenCL offer very fine grained control over page migrations, CPU-GPU cache coherency and over-subscription of video memory: OpenGL and Direct3D11 present a limited control over such aspects.

OpenGL and Direct3D 11 rely on `glMapBuffer*` and `ID3D11DeviceContext::Map`, respectively, to enforce CPU-only access to a given *buffer*. The dual unmapping operation allows the sole GPU to (coherently) access that same *buffer*. *Unified memory models* delegate to the GPU *driver* the details of the CPU-GPU coherency among shared buffers. This allows for supporting *unified memory* also in absence of dedicated hardware. In this case, the GPU *driver* provides a safe fallback strategy, where different address spaces are transparently managed via implicit *buffer* copies.

Command list-based APIs introduce the concept of *explicit memory management*. The programmer is responsible for selecting the appropriate *heap* for allocating CPU or GPU *buffers*, thus controlling specific memory *types* and alignment rules. Different *heaps* are usually identified by numbers, and have specific residency and coherency rules. Although naming and numbering might slightly change between Vulkan and Direct3D12, without loss of generality we can identify three heaps: (i) *Heap 0 – Standard device-only access*: allocations that require explicit synchronization and memory copying; (ii) *Heap 1 – Device memory accessible by the CPU*: allocations that reside on GPU local memory, but that can be made visible to and coherently shared with the CPU; (iii) *Heap 2 – CPU memory accessible by the GPU*: allocations that reside on *host* memory and that are occasionally accessed by the GPU. According to the selected memory type, different cache coherency mechanisms are in place.

For *heaps* 0 and 1, Direct3D12 and Vulkan allow *over-subscription*, but the success of these operations ultimately depends on the actual GPU *driver* implementation. The GPU hardware and *driver* might not support all the combinations of *heaps* and memory types: attempting to allocate a specific type on a specific *heap* within a system that does not support such a combination will fail. API-specific unified memory features are further detailed in Table II.

TABLE II: Unified Memory model implementations for the considered APIs

API	Unified Memory model	Description / supported features
CUDA	Kepler and Maxwell GPU uArch	Very basic implementation. Moves dirty pages at kernel launch
	post Maxwell	Prefetching, residency hints, concurrent access, oversubscription on-demand page migration and system-wide atomics
OpenCL	Coarse-Grained buffer SVM	Entire buffers that resides in device memory are shared. Oversubscription
	Fine-Grained buffer SVM	Individual loads and stores in device memory are shared. Oversubscription
	Fine-Grained system SVM	Individual loads and stores in buffers residing in host memory are shared
OpenGL and Direct3D 11	API-Specific Mapping functions controlled by access flags	Allows oversubscription and limited control over residency and coherency
Vulkan and Direct3D 12	Heap 1: Device memory	Memory type 1: Oversubscription, CPU writes are write-combined and write directly into GPU memory, whereas CPU reads are uncached.
	Heap 2: Host memory	Memory type 2: CPU writes are write-combined, CPU reads are uncached. Memory type 3: CPU reads and writes go through CPU cache hierarchy, whereas GPU is able to snoop CPU cache

TABLE III: *Device-side constructs for all the considered GPU APIs*

Keyword - Functionality	CUDA	OpenCL	GLSL (Vulkan and GL)	HLSL (D3D11 & 12)
local Thread indexing	threadIdx.x,y,z	get_local_id(0,1,2)	gl_LocalInvocationID.x,y,z	SV_DispatchThreadID.x
Group-Thread indexing	blockIdx.x,y,z	get_group_id(0,1,2)	gl_WorkGroupID.x,y,z	SV_GroupID.x,y,z
Group-Thread size	blockDim.x,y,z	get_work_dim(0,1,2)	gl_LocalGroupSizeARB.x,y,z	<i>statically specified</i>
Local Memory	__shared__	__local	shared	groupshared
InterThread synchr.	__syncthreads	barrier (CLK_LOCAL_MEM_FENCE) (CLK_GLOBAL_MEM_FENCE)	memoryBarrierShared	GroupMemoryBarrier WithGroupSync
Atomic Add	atomicAdd	atomic_add	atomicAdd	InterlockedAdd

D. API constructs for Device code

Any GPGPU application needs *device* code to describe parallel computation. Different APIs provide different approaches to specifying such code and, similarly to what we highlighted for the *host* code, it is possible to identify the underlying common semantics to various APIs, concerning abstractions such as thread indexing, local memory management and synchronization (memory barriers, fences and other *device*-only synchronization points). Table III shows an overview of the most preminent language features that control such semantics for the various APIs.

This table is of course far from being complete; however, the listed constructs and keywords constitute the minimum subset required for efficiently implementing all the parallel primitives typically used to build a full-fledged heterogeneous application [30].

IV. ON THE BENEFITS OF A UNIFIED SPECIFICATION: A CASE STUDY

In this section, we aim at assessing the benefits of a unified specification across all of the most widespread GPU APIs, that we have already analyzed. This has the potential to provide the most flexible support for heterogeneous cross-platform development. To practically conduct this type of assessment, we first propose a unified specification for *device* and *host* code programming, and then provide a proof-of-concept implementation.

We call this unified specification **GUST** (GPGPU Unified Specification and Translation), which in addition to defining the common API also mandates the translation rules that allow to target each of the considered low-level APIs as a compilation backend.

A. Introducing **GUST**

GUST exposes an interface that defines a function call for each of the basic operations we identified in section III. Due to the substantially different development philosophy characterizing the two API categories introduced in section III (*Traditional APIs* and *Command List-Based APIs*), the basic interface exposed by **GUST** does not define a common layer between the two, as we believe the expert user should not be prevented from explicitly leveraging the low-level constructs and optimization knobs exposed by *Command List-Based APIs*. We will briefly discuss in Section V how a unification layer between *Traditional APIs* and *Command List-Based APIs* could be achieved practically. Starting from Tables I and III it is in many cases straightforward to remap keywords and function calls across APIs for each entry of the tables. On the *host*-side, the GUST API derived from the taxonomy in Table I is shown in Table IV. The operations described in Table IV can be easily mapped to the previous table entries. Note that the last three columns of table IV refer to the *command list-based APIs*. More specifically, the functionality extracted from Vulkan and Direct3D 12 allows the developer to specify code sections that mandate where specific **GUST** wrapper functions can be called: setting launch configurations, arguments and symbol bindings to specific *shaders* can only occur within a code block that starts with `startCreatePSO` and ends with `finalizePSO` function calls, whereas kernel launches, PSO selection and data transfer operations can only be called in a `startCreateCmdList - finalizeCmdList` code block. We refer to these code blocks as *pipeline creation* and *command list recording* blocks. The pipeline creation code block outputs a PSO handle in which a description of a kernel launch is prepared in advance. The command list recording block stores in advance the actual commands to be later executed by the GPU, as this latter command buffer will

TABLE IV: Host-side *GUST* unified semantics from translation rules

Host side operation	Context Creation	Compile Compute Prog.	Resource Allocation and synchronization	Argument binding	Symbol copy	Launch Conf. definition	Launch Program	PSO rec. block	Cmd Buf. rec. block	Submit cmd buf.
GUST API func. call	createCtx	compileProg	allocate synchBuffer	setArg	copySymbol <T>	setLaunch Configuration	launchKernel	startCreatePSO finalizePSO	startCreateCmdList finalizeCmdList	submitWork

TABLE V: API specific functions and terminology for *GUST* unified semantics for device code: preamble definition

Preamble Definition	CUDA	OpenCL	GLSL (GL)	GLSL (Vulkan)	HLSL (D3D11 & 12)
Entry Point	<code>__global__</code> void kernelname (args...)	<code>__kernel void</code> kernelname (args...)	void main()	void main()	void kernelname (launch configuration)
Local memory static allocation	Inside kernel func.	Inside kernel func.	Outside kernel func.	Outside kernel func.	Outside kernel func.
Symbols and buffers	In kernel args	In kernel args	in layout definition	in layout definition	in layout definition
Symbols are defined as	Single values as kernel input arguments	Single values as kernel input arguments	uniforms	Push constant	Elements of a constant buffer
Buffers are defined as	Device buffers	Device buffers	SSBOs	Storage buffers	Structured Buffers
Symbols/Buffers Location	Index of an array of args	Index of an array of args	Uniform Location and SSBO bindings	Descriptor sets and layout bindings	Root signature/shader resource slot

be offloaded with the `submitWork` member function. All the other functions (resource allocation, kernel compilation, etc.) might be called anywhere outside these blocks. Violating these constraints should return explanatory errors to the user.

Clearly, Table III alone cannot capture all the different mechanics for the observed APIs. Different *device*-side languages define different entry points, have distinct ways to define data structures and respective links to the *host* side data. CUDA and OpenCL, for instance, define entry points to *device* functions that can be called from the *host* using specific keywords (`__global__` for CUDA, `__kernel` for OpenCL); the kernel layout and related data structures are simply the arguments of such functions. The same aspect is treated differently in all the other APIs: GLSL for instance typically imposes a `main` function as the entry point for a particular *compute shader*, forcing the developer to define the layout (data structure description of input and output resources and symbols) outside the *shader* function. A practical implementation of *GUST*'s *device*-side abstractions must deal with the handling of symbols' and resources' *locations*. Regarding layout locations, CUDA and OpenCL kernel invocation functions take an array of arguments as input, hence in these cases an argument location is simply their index within the array. In *device* languages derived from graphic APIs, on the contrary, the concept of input and output argument of a *shader* is decoupled for the concept of *compute program*, forcing the programmer to reason about the idea to utilize similar data layouts for different compute programs. Such information constitutes the *preamble* of *compute shaders* in GLSL and HLSL, with minor variations among OpenGL/Vulkan and D3D11/D3D12. This is shown in Table V, which summarizes the translation rules for correctly generating a preamble in a cross-platform translation layer for *device*-side code.

B. A *GUST* Reference Implementation

We present a proof-of-concept implementation of *GUST*. This implementation serves to conduct our case study, and is thus far from being complete. We cover all the key *host*- and *device*-side operations previously described. Implementing

the *GUST* interface for the *host*-side interactions can be conveniently achieved in the form of a *runtime* library or the methods of a C++ class. We choose the latter. Listings 1 and 2 show an example 2D histogram computation coded in *GUST*.

1) Host-Side Operations:

a) *Context Creation*: *Context* creation is straightforwardly implemented for each GPU API as indicated in Table I. There are some additional implementation details to cover for OpenGL and the *command list*-based APIs. For the OpenGL wrapper, a *context* is established with the `glfw` library⁹, while for core profile library function loading, `gl3w` is used¹⁰. We do this as the OpenGL standard does not regulate the creation of a GL *context*, leaving this issue to be solved by the individual operating systems. *Context* creation for both D3D12 and Vulkan requires to specify in advance the memory footprint of the application (in terms of resource usage). This allows to pre-allocate Persistent Staging Buffers (PSBs). A PSB is a buffer whose dimension is set at *context* creation time and is persistently mapped (until *context* destruction) within the GPU driver: it represents a staging area in which data to and from the *device* can be read or written by the application, exploiting the maximum available memory bandwidth without wasting time in mapping and unmapping different buffers when data needs to be moved. When a *device*-side allocation is requested, a pointer within the range of possible addresses from the PSBs is returned. Compared to traditional APIs, this implies managing a local allocation table with its segmentation logic. *GUST* can take care of this transparently. For all the APIs, the *GUST* initializes the context with the *createCtx* function call, that initializes with default values any API-specific setting (e.g. PSB size for command list based APIs or preferred device type for OpenCL). *Context* creation is where *GUST* selects the corresponding API wrapper; no other *GUST* function calls are permitted before the successful creation of a *context*.

b) *Compute program compilation*: The *compileProg* function call takes care of compiling *GUST* *device* code.

⁹<http://www.glfw.org/>

¹⁰<https://github.com/skaslev/gl3w>

Listing 1: Host-side GUST code for 2D Histogram.

```
//HOST CODE
ComputeInterface *ctx = //API specific constructor.
ctx->createContext(); //context creation

//Compilation
//substitute "CUDA" with your target API
MLTL mltl(MLTL_TO_CUDA);
mltl.translateFromFile("path to mltl src file");
std::string devStr = mltl.generateSourceString();
ctx->loadAndCompileShader(devStr, "histo");

/*Application params settings*/
uint32_t bins, W, H ...

//Host + device allocation
int* img_ch=ctx->deviceSideAllocation(W*H*sizeof(int));
int* histo=ctx->deviceSideAllocation(bins*sizeof(int));

//Host-side init for the buffers
[...]

//host to device memcpy
ctx->synchBuffer((void*)&img_ch, HOST_TO_DEVICE);
ctx->synchBuffer((void*)&histo, HOST_TO_DEVICE);

//Symbols and arguments setting
ctx->setArg((void*)&img_ch, "histo", 0);
ctx->setArg((void*)&histo, "histo", 1);
ctx->copySymbolInt(W, "histo", 2);

//launch compute kernel
ctx->setLaunchConfiguration(
    ComputeWorkDistribution_t(W / 32, H / 32),
    ComputeWorkDistribution_t(32, 32));
ctx->launchComputation("histo");
ctx->synchLaunch();

//data copy-back
ctx->synchBuffer((void*)&histo, DEVICE_TO_HOST);
ctx->deviceSynch();

[...] //use data

//freeing host and device resources
ctx->freeResources();
```

Device code can be stored in a *string* or in a file and might be specified using a tag-based language implementation of an API-specific *device* code. We call our instance of such a language the *MLTL* (Metal Layer Translation Language). Internally, *GUST* translates MLTL code into API-specific code, as API selection already occurred during *context* creation. Once *device*-specific code is available, *GUST* wraps compiling functions as described in table I. Instead, using the recently released NVRTC¹¹ (NVIDIA Runtime Compilation Library) we are able to create and compile kernels at runtime, hence uniforming the CUDA behavior to all the others APIs. For *command list-based APIs*, compiling a kernel in D3D12 is done in a very similar manner as seen in D3D11, but for Vulkan an additional step is needed. Using the Valve's LunarG Vulkan SDK¹² and by exploiting its *glslangValidator* executable the user is able to feed *GUST* with a *string* or a file containing a GLSL *compute shader* with Vulkan-specific extension and obtain a SPIR-V source. From that, a *vkShaderModule* can be created.

c) *Resource Management*: *GUST* offers an *allocate* function to wrap backend API-specific allocation functions.

Listing 2: Device-side GUST code (MLTL) for 2D histogram.

```
[LAYOUT_DEF]
ARG: bininput, R, int
ARG: histogram, RW, int
SYM: W, int
SHM: shmem, int, 256
[END]
[PROGRAM][histo]
int local_index = _GRSX*_LITY_ + _LITX_;

if(local_index < 256) SHM(shmem,local_index)=0;

_LOCMEMBAR_

int index =
    ARG(bininput,SYM(W)*(_LITY+_GIDY*_GRSY_) +
        (_GIDX*_GRSX+_LITX_));
_ATADD_(SHM(shmem,index),1);

_LOCMEMBAR_

if(local_index < 256)
    _ATADD_(ARG(histogram,local_index),
        SHM(shmem,local_index));
[END]
```

In our example implementation, an allocation call always returns a pointer able to be read and written by the *host*. Internally, *GUST* maintains an allocation table in which for each *host* pointer a corresponding *device* pointer is present: in this way, the complexity of dealing with two address spaces (GPU and CPU) is partially hidden. However, resources must be explicitly synchronized (*synchBuffer*) and released¹³. Once a resource is created, it can be used as input argument to *setArg*, to be accessed by the GPU during kernel execution.

d) *Kernel invocation*: Before any kernel invocations, all the arguments must be set. In addition to that, *symbols* and launch configuration must also be specified (resp. *copySymbol<T>* and *setLaunchConfiguration*) Actual kernel invocation occurs through the *launchKernel* call. Internally, our implementation of *GUST* uses tables to keep track of all the previously compiled kernels, related launch configuration and layout. We recall that *command list-based APIs* must defer actual copy and kernel invocation as described in Sections III-C2 and IV.

e) *Bookkeeping*: As we highlighted in the previous paragraphs, allocations, PSOs and compiled kernels are stored within internal data structures related to a compute context. Such structures are implemented as *std::maps* and software cache for frequently utilized resources/kernels. More specifically, the context cache remembers recently launched kernels and respective arguments to avoid table look-ups.

f) *Error checking*: *GUST* API calls are internally checked for errors. This might occur by delegating error checking to the target GPGPU API or by exploiting the bookkeeping structures. For instance, context creation failure is handled via API-specific error checking. Conversely, since compiled kernels are stored in a *std::map*, attempting to call a kernel that was not previously compiled will be easily detected by simply looking up that map in our implementation.

¹¹<http://docs.nvidia.com/cuda/nvrtc/index.html>

¹²<https://www.lunarg.com/vulkan-sdk/>

¹³Due to space constraints these operations are not discussed

TABLE VI: SLOC for basic compute operations. *GUST* compared to seven GPU APIs

Host Operation/ API SLOC	CUDA-Runtime API	CUDA Driver API	OpenCL	OpenGL	D3D11	D3D12	Vulkan	GUST
Context Creation	(implicit)	5	60	47	5	60	242	1
Memory Allocations	2	2	3	4	10	60	30	1
Memory Transfer	1	1	1	4	10	(in cmd buf recording) 40	(in cmd buf recording) 27	1
Set Kernel Argument	(in kernel call)	(in kernel call)	1	2	2	(in pipeline creation) 20	(in pipeline creation) 10	1
Copy Symbol	(in kernel call)	(in kernel call)	1	2	3	(in cmd buf recording) 2	(in cmd buf recording) 27	1
Set Launch configuration	(in kernel call)	(in kernel call)	(in kernel call)	(in compute dispatch)	(in compute dispatch)	(in pipeline creation) 2	(in pipeline creation) 20	1
Launch Kernel	1	1	1	1	1	(in cmd buf recording) 2	(in cmd buf recording) 2	1
Wait for completion	1	1	1	2	7	10	2	1
Kernel compilation	uses nvcc	90 (with nvtc)	43	90	50	44	(uses glslangValidator) 75	1
Submit Work	-	-	-	-	-	1	1	1
Pipeline creation	-	-	-	-	-	90	146	2
Cmd Buf recording	-	-	-	-	-	4	4	2

2) *Device-side Operations*: In our reference implementation, we opted to manage *device*-side translation rules and preamble definition by relying on a tag-based language that is processed by means of a source-to-source compiler as part of the compute program compilation process described in the previous section¹⁴. From the translation rules defined in tables III and V, our reference implementation of *GUST* provides a Meta Language in which the programmer specifies arguments, symbols and local scratchpad memory allocations in a tag-based language. Then, a standard C program enriched with specific keywords is used to describe the parallel algorithms executed by the kernel. A source file constructed with such rules is named a MLTL (Meta Language Translation Layer) source file. The user can choose two ways of writing a compute kernel: (1) using API-specific *device* code; or (2) exploiting the above mentioned MLTL language. Choosing the latter requires an additional step for the compilation process, where the MLTL code is parsed and translated into the functional equivalent for the target API. Listing 2 shows a MLTL implementation of a histogram calculated over a 2D matrix of integer values. The tag based structure is evident. We can identify two tags: a first tag detailing the layout definition (*LAYOUT_DEF*) and a tag in which the behavior of the kernel is expressed as a standard C program (*PROGRAM*). Tags are delimited by the keywords between square brackets. The *LAYOUT_DEF* tag is composed of a forward declaration of arguments (one argument per line, starts with *ARG*: with the syntax `<name>, <usage> ∈ {R, W, RW}, <type>`), symbols (one per line, starts with *SYM*: with the syntax `<name>, <type>`) and static local memory allocation (called *shared*, starts with *SHM*: with the syntax `<name>, <type>, <elements>`). Layout definition is part of what is known in *GUST* as *preamble definition*. The *PROGRAM* tag contains standard C code defined by an entry point (a kernel identifier able to be referred from *host* code, in the example in Listing 2 “histo” is used). Keywords delimited by underscores refer to MLTL-specific syntax: *_GTSX_* stands for block size over the X dimension, *_LITY_* refers to the local thread index ID over

the Y dimension and so on. Fetching data from the inputs defined in the (*LAYOUT_DEF*) tag is done with the syntax *ARG|SHM*(*name*, *index*); for symbols, *SYM*(*name*) is used. *SHM* refers to local memory. Atomic operations and local memory barriers can be instantiated with the *_ATADD_* and *_LOCMEMBAR_* tags, respectively. For space constraints we do not list all the possible combinations of thread indexing and synchronization currently implemented for the MLTL. The MLTL layer in *GUST* is able to substitute the MLTL specific keywords in order to recreate the kernel in every API-specific language: in doing so, our reference implementation can issue a warning or error if a violation of the MLTL-specific syntax was detected during source translation. Non-MLTL-related errors will be flagged by the chosen API library during native compilation.

C. An Early Assessment of *GUST*

We assess the benefits of the proposed unified specification in terms of ease of development and performance penalty (overhead characterization). Performance hit is also qualitatively discussed for the unification of *traditional* and *command list-based* APIs.

1) *Ease of development*: The abstraction provided by *GUST* translates in a simplified coding style, which we quantify by means of source lines of code (SLOC). SLOC for each API compared to *GUST* is visible in Table VI. Here, we highlighted how many lines of code are needed for each operation abstracted by our reference *GUST* implementation: for each API, the line count includes basic error checking and bookkeeping operations. D3D12 and Vulkan are intuitively expected to require a similar line code count, but this is not observed in table VI. This is due to the fact that most of D3D12 structure initialization occurs through a helper library function (D3DX12) which moderates coding verbosity in Direct Compute applications.

From Table VI is evident that *GUST* is very effective at reducing the amount of code to be written for GPU application development (in the compute pipeline), as most of the functionalities are wrapped within a single function call. We also include numbers from a seventh API, the *CUDA*

¹⁴For an integrated compilation process, custom keywords could also be implemented in the form of language extensions (e.g., C, C++) or compiler annotations (OpenMP-style *#pragmas*).

runtime API, which is not wrapped by our implementation of **GUST**, but provides an interesting term of comparison since it is in practice the most used GPGPU API, due to its ease of use. Compared to *command list-based APIs* the reduction in SLOC for **GUST** is major. On the *device* side, for MLTL there is basically a 1:1 translation for every construct to any back-end API, which implies no increase in SLOC. Only what we call the MLTL *preamble* slightly increases the instruction count compared to CUDA and OpenCL. This is because while CUDA and OpenCL describe kernel inputs and output buffers in the device function signature, in GLSL and HLSL (used for the other APIs) inputs and outputs are specified in a separate code block: for simplicity our MLTL prototype translator matches the GLSL/HLSL specification.

In **GUST**, porting one application from an API to another is simple, provided that both the starting API and the destination API belong to the same category (traditional and command list-based). The only part that would require a light programming effort is instantiating the compute context: some API-specific settings might be necessary to be specified as context constructor parameters. However, default constructors passing default settings can be easily put in place. Moreover, it is trivial to automate the selection of the most appropriate API backend in **GUST** during context creation as information such as device vendor and installed drivers can be easily queried at runtime. Device code written in MLTL does not require any modification. The possibility to port the host code of an API belonging to a category to a target API belonging to a different category is discussed in section IV-C3.

2) *Overhead characterization*: The overhead characterization w.r.t. the APIs wrapped by **GUST** depends of the design choices of the **GUST** implementer; our reference implementation heavily relies on red-black tree-based maps to perform state tracking and bookkeeping of *host-side* constructs like compiled *shaders* and allocation tables. Every time the **GUST** user attempts to use a kernel with specific resources, look-up operations on all these maps are triggered. Overhead given by the look-up operations is therefore related to how many kernels and allocations are distinctively used within the same application. Since we used regular *std::maps*, look-up complexity is logarithmic in size; moreover, recently used kernels and layouts are cached to local variables to amortize look-up costs for complex applications. On the *device* side, our reference implementation adds no overhead other than the compilation time from MLTL to API-specific *device* code.

However, it is still interesting to measure how **GUST** performs in terms of overhead compared to the only API that is not directly targeted: the *CUDA Runtime API*. For this experiment, **GUST** is set to wrap the CUDA driver API. Results on a small set of benchmarks are reported in Figure 2. The benchmarks include a Vector Addition (VADD), Single-Precision $A \times X + Y$ (SAXPY), 2D histogram computation (HISTO) and a parallel reduction on an array of integers (MINREDUX). Dataset size is 512K elements for each buffer in SAXPY, VADD and MINREDUX. Histogram is computed on a 256-colors image (1024×768). Although all the benchmarks call all the GUST functions listed in Tables IV and V we

also provide an experiment with MINREDUX where we study the effect of varying the input dataset size, and we provide a fifth benchmark, a 2D Finite Difference Time Domain solver¹⁵ (FDTD2D). FDTD2D represents a more complex application composed of different *kernels* that are iterated multiple (500) times. The dataset consists of 2048x2048 floating point values. All The tests are executed on a Intel i7 x86_64 platform featuring a GTX860M NVIDIA discrete GPU.

The leftmost plot in Figure 2 shows execution times for **GUST** and the CUDA runtime API (normalized to the latter). Execution times include allocations, data initializations, memory transfers (any direction), kernel launches and *host-device* synchronization operations. In all the tested benchmarks **GUST** performs on par with the CUDA runtime API (within $\pm 0.04\%$). This is not unexpected, as both **GUST** and the CUDA runtime eventually make the same calls to the underlying CUDA Driver API functions, as it's easily confirmed running the CUDA profiler, *nvprof*.

Overhead impact is also independent of the dataset size, as shown by our experiment with MINREDUX in the center plot in Figure 2). Varying the dataset size from 512K to 10M integer elements does not significantly affect execution time. This is also intuitively explained by the fact that bookkeeping operations are only sensitive to the number of buffers used in a program, not their size. The larger the dataset, the longer the kernel processing time and thus the less relevant the already negligible host-side overheads.

The rightmost plot in Figure 2 shows the overhead implied by the MLTL compilation step occurring before native, API-specific back-end compilation. Specifically, this overhead is calculated as $\frac{T_{GUSTcompile} + T_{NVRTCcompile}}{T_{NVRTCcompile}}$ where $T_{GUSTcompile}$ is the time to translate MLTL to CUDA and $T_{NVRTCcompile}$ is the native CUDA compilation time using NVRTC. Again, this overhead is largely implementation-specific, but even with our proof-of-concept translator it is easy to see that this overhead is negligible, as MLTL only supports a small subset of device-side constructs and only operates simple line-by-line translation, leaving more sophisticated optimizations via full-blown compiler intermediate representation (IR) to the following API-specific compilation steps.

3) *Unifying API categories*: In the methodology we have presented we kept a distinction between *traditional* APIs and *command list-based* APIs, due to the different application development philosophy underlying the two. However, non-expert developers dealing with an application port that targets a *command list-based* API might still find it very useful to stick to a simple, unified specification layer. To support this, it is possible to define mechanisms for just-in-time creation of PSOs and command lists starting from a program that exploits the *traditional* API coding style. We discuss two strategies: (i) lazy PSO and command buffer creation and (ii) command batching from static analysis.

¹⁵From the Polybench benchmark suite: <http://web.cs.ucla.edu/~pouchet/software/polybench/>.

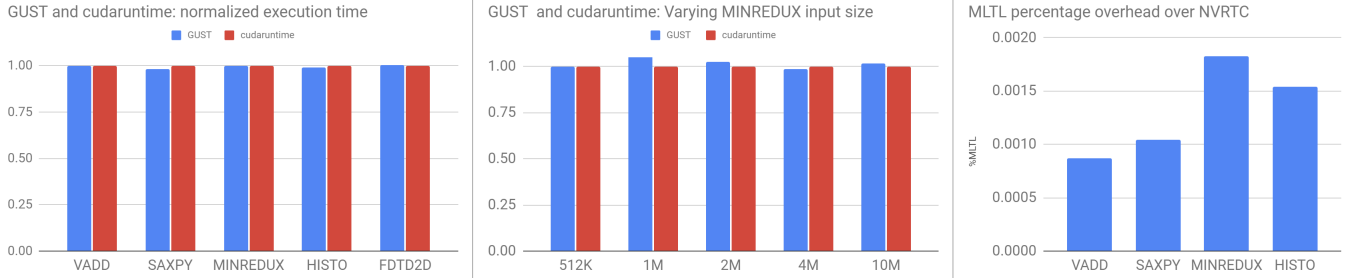


Fig. 2: GUST prototype implementation vs. CUDA Runtime API. Average execution time (left). Effect of varying dataset size in the MINREDUX benchmark (center). MLTL compilation overhead over total compilation time (right).

a) *Lazy PSO and command buffer creation*: – Starting from a *GUST* application written using the *traditional* API coding style, each kernel call is intercepted at runtime: a corresponding PSO is created on the fly according to the kernel invoked, its layout and its launch configuration. Such PSO is hashed and stored in a cache, so to avoid recreating the same PSO in case of reuse. *vuda*¹⁶, a coding effort that enables the user to write CUDA code over a Vulkan implementation, follows this blueprint. Memory transfers and actual kernel launches will have to be recorded in a command buffer: this introduces the problem of *when* to start the recording phase and when to stop it, to then submit the command buffer to the GPU. This is because the workload might be composed of a sequence of an unknown number of *copy* and *compute* commands. This can be solved by recording a command buffer for each operation within the sequence, and offloading the work as soon as the recording phase terminates. Another strategy for managing the command buffer is to start the recording upon the first kernel launch or copy invocation and terminate the recording phase when the application calls a GPU-CPU synchronization point (e.g: wait for idle) or when a read-back of a resource used by the GPU is requested by the *host*. While the first strategy is very easy to implement, the resulting performance hit can be dramatic for long GPU command sequences, especially if the GPU commands submitted present small execution times. The second strategy allows *GUST* to batch many commands within a single command buffer recording phase so to minimize submission operations. This is optimal, but implementation-wise a variety of problems arise: in GPGPU computing calls to CPU-GPU synchronization points are not mandatory, and the read back resources by the *host* can occur via multiple *device-to-host* copy commands. These are the uncertainties to face when deciding to close the recording phase of a command buffer. Heuristics based on time windows might be used to mitigate these issues: the first copy or dispatch command signals the beginning of a recording phase and all the commands invoked within a pre-determined time window will end up in the same command submission. A third possible strategy exploits CUDA *graphs*: a feature introduced in CUDA 8. CUDA graphs enqueue GPU commands with pre-defined precedence constraints. Building a CUDA graph is semantically very similar to recording a command buffer, as intended

in next-generation APIs. Although this feature is designed for reducing submission overheads, offloading commands in such a way might still result in higher overheads than submitting a command buffer using next-generation APIs [31].

b) *Command batching from static analysis*: – Static code analysis can theoretically improve upon the cost for just-in-time creation of PSOs and help in the selection of the right heuristics for command buffer recording. Creating a PSO for each kernel (including layout definition and launch configuration) takes 20 ms in an NVIDIA Jetson AGX embedded board using Vulkan. In the same board, recording a command buffer takes a constant time of 2.5 ms for sequences of less than 100 *copy* and *compute* commands; after that, the time needed for recording a command list scales linearly [31]. Trivially, for the examined board, recording and submitting a command list for each *copy* or *compute* command is convenient for GPU execution times larger than the time needed to finalize the command list for each command. In this way, CPU execution times for the recording phase can proceed in parallel with GPU execution. If using an approach based on time windows it is instead convenient that the length of the time window is sized as a negligible value compared to the sum of the execution times of the commands batched within the time window. Command lists can be hashed and cached just like PSOs for optimizing submission operations in case of periodic workloads.

V. FINAL DISCUSSION AND CONCLUSION

In this paper we presented an exhaustive overview on the most commonly used GPU APIs for general purpose computing. From there we derived a taxonomy with two macro categories: *traditional* APIs (CUDA, OpenCL, OpenGL compute shaders, DirectCompute over Direct3D11) and *command list-based* APIs (Vulkan and DirectCompute over Direct3D12 APIs). Among APIs belonging in the same category we identified the subset of programming constructs that describe the same functionality, both for *host*- and *device*-side code. This allowed us to define a unified specification across the various APIs, which is sufficiently expressive to capture the basic blocks of any heterogeneous programming application targeting a GPGPU as an accelerator. We provide a reference implementation of such specification, called *GUST*, which we use to validate the benefits of the approach. Our unified specification and the *GUST* tool proved to be very effective at porting

¹⁶<https://github.com/jgbit/vuda>

applications onto different platforms, simplifying both application development and extensive cross-benchmarking [31]. Performance-wise, we have observed minimal overheads for the abstraction layer.

GUST is meant as a cross-platform API specification, yet some of the APIs remain hardware- or operating system-specific (e.g., D3D wrappers only work on Windows; CUDA is specific to NVIDIA hardware). Trivially, there is no workaround for this: the only APIs known to be completely cross-platform are the ones defined by the Khronos specifications. However, the proposed methodology supports all such standards, and identifies at least two alternatives for all the common operating system and hardware configurations found in the market, for both desktop and embedded systems.

As of now, **GUST** is far from being feature-complete. In particular, some relevant differences between the APIs complicate the unification of the specification. Today, our reference implementation manages this functionality gap by allowing the experienced developer to write algorithms in MLTL syntax, so to have a translation to all the other APIs specific device-code. Then, through manual tuning, the developer is still able to use API-specific features for optimization.

As future work, we plan to incrementally add to our reference implementation additional features, such as support for unified memory models, parallel submission of command buffers (in the form of multiple queue of commands), multi-GPU support and the implementation of a unifying layer between traditional and command list-based APIs (as discussed in section IV-C3). Our reference implementation of **GUST** will soon be released as an open source contribution. The interested reader can find **GUST** interface sources together with a Vulkan implementation at this link https://git.hipert.unimore.it/rcavicchioli/cpu_gpu_submission/-/tree/master/vkcomp.

ACKNOWLEDGEMENT

This work has received funding from EU projects H2020 CLASS (780622) and ECSEL JU NEWCONTROL (826653) and COMP4DRONES (826610).

REFERENCES

- [1] A. Sridhar, A. Vincenzi, M. Ruggiero, and D. Atienza, “Neural Network-Based Thermal Simulation of Integrated Circuits on GPUs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 23–36, 2012.
- [2] E. Schneider and H. Wunderlich, “SWIFT: Switch-Level Fault Simulation on GPUs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 122–135, 2019.
- [3] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, “DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.
- [4] B. Shi, Y. Zhang, and A. Srivastava, “Accelerating Gate Sizing Using Graphics Processing Units,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 160–164, 2012.
- [5] M. Ujaldon and U. V. Catalyurek, “High-performance signal processing on emerging many-core architectures using CUDA,” in *2009 IEEE International Conference on Multimedia and Expo*. IEEE, 2009, pp. 1825–1828.
- [6] S. R. Upadhyaya, “Parallel approaches to machine learning—A comprehensive survey,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 284–292, 2013.
- [7] S. Mittal and J. S. Vetter, “A survey of CPU-GPU heterogeneous computing techniques,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 69, 2015.
- [8] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez, “Experiences with mobile processors for energy efficient HPC,” in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 464–468.
- [9] P. Faber and A. Größlinger, “A comparison of GPGPU computing frameworks on embedded systems,” *IFAC-PapersOnLine*, vol. 48, no. 4, pp. 240–245, 2015.
- [10] S. Schaetz and M. Uecker, “A multi-GPU programming library for real-time applications,” in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2012, pp. 114–128.
- [11] J. Enmyren and C. W. Kessler, “Skepu: a multi-backend skeleton programming library for multi-gpu systems,” in *Proceedings of the fourth international workshop on High-level parallel programming and applications*, 2010, pp. 5–14.
- [12] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryuji, and T. R. Scogland, “Raja: Portable performance for large-scale scientific applications,” in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 71–81.
- [13] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, “Hpx—an open source c++ standard library for parallelism and concurrency,” *Proceedings of OpenSuCo*, vol. 5, 2017.
- [14] H. C. Edwards and C. R. Trott, “Kokkos: Enabling performance portability across manycore architectures,” in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.
- [15] S. Memeti, L. Li, S. Pillana, J. Kołodziej, and C. Kessler, “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption,” in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, 2017, pp. 1–6.
- [16] A. Acosta, C. Merino, and J. Totz, “Analysis of OpenCL Support for Mobile GPUs on Android,” in *Proceedings of the International Workshop on OpenCL*. ACM, 2018, p. 27.
- [17] S. Kim and S.-K. Kim, “Comparison of OpenCL and RenderScript for mobile devices,” *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14 161–14 179, 2016.
- [18] R. Membarth, O. Reiche, F. Hannig, and J. Teich, “Code generation for embedded heterogeneous architectures on Android,” in *2014 Design, Automation & Test in Europe*

- Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.
- [19] M. Bourgoïn, E. Chailloux, and J.-L. Lamotte, “Efficient abstractions for GPGPU programming,” *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 583–600, 2014.
- [20] A. Mazaheri, J. Schulte, M. W. Moskewicz, F. Wolf, and A. Jannesari, “Enhancing the Programmability and Performance Portability of GPU Tensor Operations,” in *European Conference on Parallel Processing*. Springer, 2019, pp. 213–226.
- [21] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou, “A unified optimizing compiler framework for different GPGPU architectures,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 2, p. 9, 2012.
- [22] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, “GPU programming in rust: Implementing high-level abstractions in a systems-level language,” in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 315–324.
- [23] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, “Compiling a high-level directive-based programming model for gpgpus,” in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 105–120.
- [24] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. De Sande, “accULL: an OpenACC implementation with CUDA and OpenCL support,” in *European Conference on Parallel Processing*. Springer, 2012, pp. 871–882.
- [25] D. Sharlet, A. Kunze, S. Junkins, and D. Joshi, “Shevlin park: Implementing c++ amp with clang/llvm and opencl,” in *General Meeting of LLVM developers and users*, 2012.
- [26] J. Fang, A. L. Varbanescu, and H. Sips, “A comprehensive performance comparison of CUDA and OpenCL,” in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 216–225.
- [27] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.
- [28] A. Rao, A. Srivastava, K. Yogesh, A. Douillet, G. Gerfin, M. Kaushik, N. Shulga, V. Venkataraman, D. Fontaine, M. Hairgrove *et al.*, “Unified memory systems and methods,” Jul. 23 2015, uS Patent App. 14/601,223.
- [29] M. Knap and P. Czarnul, “Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs,” *The Journal of Supercomputing*, pp. 1–21, 2019.
- [30] J. Nickolls, I. Buck, and M. Garland, “Scalable parallel programming,” in *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE, 2008, pp. 40–53.
- [31] R. Cavicchioli, N. Capodieci, M. Solieri, and M. Bertogna, “Novel Methodologies for Predictable

CPU-To-GPU Command Offloading,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.



Nicola Capodieci is an associate researcher at the HiPeRT-Lab of the University of Modena and Reggio Emilia. His main research interests range from distributed systems to languages, architectures and programming models for GPUs.



Roberto Cavicchioli Roberto Cavicchioli is a Post-Doctoral Researcher at the HiPeRT-Lab (University of Modena and Reggio Emilia) whose research is focused on parallel computing for heterogeneous systems, optimization algorithms, machine learning and real time scheduling. He received his Ph.D. in 2014.



Andrea Marongiu received the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. He has been a postdoctoral research fellow at ETH Zurich, Switzerland. He currently is an associate professor at the University of Modena and Reggio Emilia. His research interests focus on programming models and architectures in the domain of heterogeneous multi- and many-core systems-on-chip. In this field, he has published more than 100 papers in peer-reviewed conferences and journals.