Taming Multi-GPU Greedy Scheduling Through a Polyglot Runtime

Guido Walter Di Donato, *Graduate Student Member, IEEE,* Ian Di Dio Lavore, *Graduate Student Member, IEEE,* Alberto Parravicini, Francesco Sherzi, *Graduate Student Member, IEEE,* Marco Arnaboldi, Arnaud Delamare, Daniele Bonetta, Marco Domenico Santambrogio, *Senior Member, IEEE.*

***Abstract*—Multi-GPU systems are increasingly being deployed in cloud data centers, but using GPUs efficiently from high-level programming languages remains a challenge. Moreover, exploiting the full capabilities of multi-GPU systems is an arduous task due to the complex interconnection topology between available accelerators and the variety of inter-GPU communication patterns exhibited by different workloads. This work introduces a novel scheduler for multi-task GPU computations that provides transparent asynchronous execution on multi-GPU systems without requiring prior information about the program dependencies or the underlying system architecture. Our scheduler integrates with the polyglot GraalVM ecosystem and is therefore available for multiple high-level languages, providing a general framework that can significantly lower the barriers to entry to multi-GPU acceleration. We validate our work on a set of benchmarks designed to investigate scalability and inter-GPU communication. Experimental results show how our scheduler automatically achieves 80-90% peak performance against hand-optimized CUDA host code on Volta and Ampere multi-GPU systems.**

***Index Terms*—Heterogeneous Computing, Graphics Processing Units, Multi-GPU, Task Scheduling, Run-time Systems, Polyglot.**

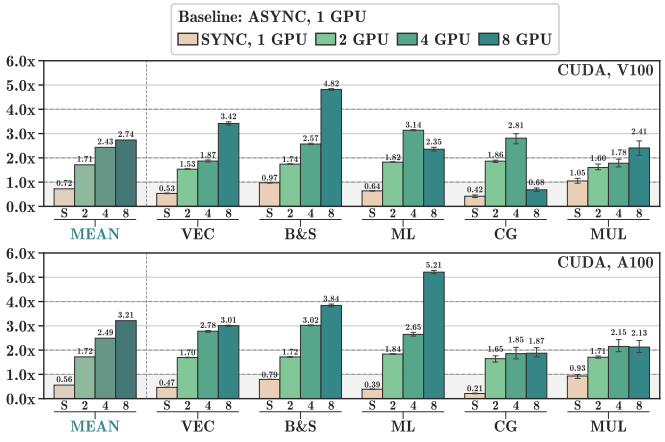

Fig. 1. Performance scaling (speedup) of GPU benchmarks whose scheduling and data-transfer has been hand-optimized in C++ CUDA, when running on up to 8 Nvidia Tesla V100s and A100s. While some of them do not fully benefit from 8 GPUs, all present a speedup when using multiple GPUs.

I. Introduction

MODERN computing systems increasingly rely on hardware accelerators like Graphics Processing Units (GPUs) to achieve performance goals. Indeed, their massive multi-threading ability can provide extremely high throughput to applications from different domains, such as Deep Learning and graph analytics [1]. Moreover, the increasing demand for faster computation from data-intensive workloads has driven the deployment of multi-accelerator servers in shared multi-tenant environments, such as cloud data centers [2]. Multi-GPU systems provide an additional level of parallelism that can be leveraged to push the performance of accelerated applications even further. As an example, Figure 1 illustrates the speedup achieved in different CUDA [3] benchmarks (presented in §IV-B) by hand-optimizing inter-GPU data transfer and the execution of independent computations, when running on up to 8 GPUs. The experimental results show how all the benchmarks present performance improvements when using multiple GPUs, achieving up to $5.21\times$ speedup when running on 8 GPUs. These results clearly capture the benefit of spreading the computation across multiple accelerators.

Today, major cloud providers such as Amazon, Google, Oracle, and others provide multi-GPU systems to their users [4]. In such systems, GPUs are increasingly interconnected in complex topologies. Despite the development of new interconnection technologies, such as NVLink [5] and NVSwitch [6], the interconnection bandwidth (up to $\sim 900\,\mathrm{GB\,s^{-1}}$) is still much slower than the global memory bandwidth of modern GPUs (up to $\sim 3900\,\mathrm{GB\,s^{-1}}$) [7]. Moreover, compute-intensive workloads are exhibiting a wider variety of inter-accelerator communication patterns. While exploiting the full capabilities of a GPU is already an arduous task requiring expert knowledge of asynchronous programming, writing efficient multi-GPU code is even more complex. In fact, although employing multiple GPUs can deliver many advantages, it presents new challenges in workload management and scheduling to obtain optimal performance [2], [4].

To date, the adoption of GPUs is often limited to specific domains offering libraries or Domain-Specific Languages (DSLs) that mask the GPU computation [8], [9]. This is mainly due to the limited integration of GPUs with high-level programming languages like Java or Python. More-

G.W. Di Donato, I. Di Dio Lavore, A. Parravicini, and M. D. Santambrogio are with the Department of Electronic, Information and Bioengineering, Politecnico di Milano, Milano, IT, 20133. E-mail: {guidowalter.didonato, ian.didio, alberto.parravicini, marco.santambrogio}@polimi.it

F. Sgherzi is with the XXX, Barcelona Supercomputing Center, Barcelona, Spain. E-mail: ,

M. Arnaboldi and A. Delamare are with Oracle Labs, Zurich, Switzerland. D. Bonetta is with Oracle Labs, Amsterdam, Nederland.
E-mail:{marco.arnaboldi, arnaud.d.delamare, daniele.bonetta}@oracle.com

over, APIs offering complete control over the GPUs require tremendous efforts to unleash the full hardware potential, including extensive debugging and profiling. In the last few years, important efforts have been made to integrate GPUs with high-level programming languages, to lower the barriers to entry to GPU acceleration [10], [11]. A worthy example is GrCUDA [12], [13], a polyglot CUDA API based on the GraalVM ecosystem [14]–[16]. Implemented as a Truffle [17] DSL, GrCUDA enables GPU acceleration in all the languages supported by GraalVM.

Parravicini et al. [18] recently extended the open-source GPU scheduler of GrCUDA to support asynchronous CPU and GPU execution. The cornerstone of their scheduler is a data-flow Directed Acyclic Graph (DAG) built at runtime, representing relationships between computations that involve the GPU. Their work focused on space-sharing and transfer-computation overlap, and it is only suitable for single-GPU computations. In this work, we tackle the problem of scheduling on multi-GPU systems, masking the complexity of inter-GPU data-transfer management. The problem is significantly harder than the single-GPU counterpart, as it requires computing data placement and migration costs at run time to identify the optimal scheduling.

We propose a novel methodology to schedule multi-GPU computations without requiring prior information about the program dependencies or the underlying system architecture. Our work is implemented as an extension of the asynchronous scheduler proposed in [18], and it enables GrCUDA to become a general framework that can significantly lower the barriers to entry to multi-GPU acceleration.

In summary, the main contributions of this paper are:

1) A **greedy multi-GPU scheduler** that integrates directly with the original GrCUDA architecture, extending the lower-level of the runtime without changing the API and the dependency DAG computation (§III-A).
2) **Four multi-GPU scheduling policies** able to handle multi-task workloads of increasing complexity (§III-B).
3) **Five multi-GPU benchmarks** designed to highlight the performance profile of our new scheduler, focusing on scalability, inter-GPU communication, and the benefit of transfer-computation overlap (§IV-B).
4) An evaluation of how our scheduler automatically achieves an average of 80–90 % peak performance against hand-optimized CUDA C++ host code (§IV-C, §IV-D, §IV-E).

II. Background and Motivation

Multi-GPU programming is hard: runtime scheduling is often sub-optimal, and the lack of support for high-level programming languages makes it difficult for data scientists to leverage the power of modern hardware. In this section we summarize the complexity of multi-GPU computations (§II-A), and we introduce the GrCUDA language binding (§II-B) and its runtime extension proposed in [18] (§II-C). Then, we describe the rationale behind the development of an asynchronous and transparent multi-GPU scheduler (§II-D).

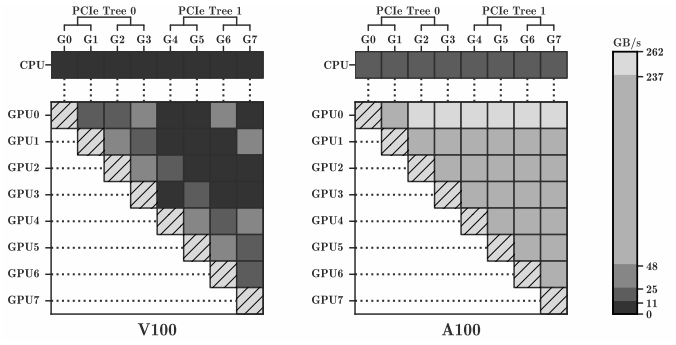

Fig. 2. Heatmap of the GPU-GPU and CPU-GPU interconnection bandwidth of a system equipped with 8 Nvidia V100s or 8 Nvidia A100s. Same-GPU interconnection bandwidth is not shown, and it amounts to the GPU's memory bandwidth.

A. Multi-GPU Computations

Given the rapid evolution of GPU hardware, it is very easy for existing tools and frameworks to have limited portability as hardware and programming languages change. For example, modern GPU interconnections such as NVLink [5], and even more so NVSwitch [6], have made heavy GPU-GPU communication feasible. In the past, instead, it was necessary to resort to the limited bandwidth of PCI Express (PCIe) [19], limiting multi-GPU computations to processing independent data partitions and using GPU-GPU communication mostly for control-flow data [20]. Consequently, tools that optimized inter-GPU transfer assuming PCIe interconnections might no longer be relevant since the introduction of faster interconnections. In this fast-moving context, GPU runtimes should be able to *adapt to different hardware in a transparent way*. This is the main motivation of our work, which is agnostic to the underlying interconnection technology and automatically infers information about inter-device transfer performance by dynamically profiling the system, as discussed in §III-B. Developers can expect *high performance in any deployment*, not just on the system where they developed their code. This high portability is especially valuable in cloud settings, where the underlying hardware is often changing based on availability, performance needs, and other non-deterministic factors.

To capture the complexity of GPU and CPU interconnections in modern multi-GPU systems, Figure 2 shows a heatmap of the GPU-GPU and CPU-GPU interconnection bandwidth of an NVLink-V2 machine with 8 Nvidia V100s and of an NVSwitch machine with 8 Nvidia A100s, as available on the Oracle Cloud Infrastructure (OCI) [21]. In detail, it shows the unidirectional bandwidth between each couple of devices, obtained experimentally. The bandwidth of any device pair is symmetric. For both the systems, the bandwidth between CPUs and each GPU appears to be uniform, even though we use dual-socket servers where CPUs are connected through QPI [22] (NVLink-V2 machine) or Infinity Fabric [23] (NVSwitch machine). Bandwidth on NVSwitch is significantly higher and more uniform ($\sim 250\,\mathrm{GB\,s^{-1}}$, unidirectional) than NVLink ($\leq 48\,\mathrm{GB\,s^{-1}}$), as all GPUs are fully connected to the same switch instead of providing a number of NVLink

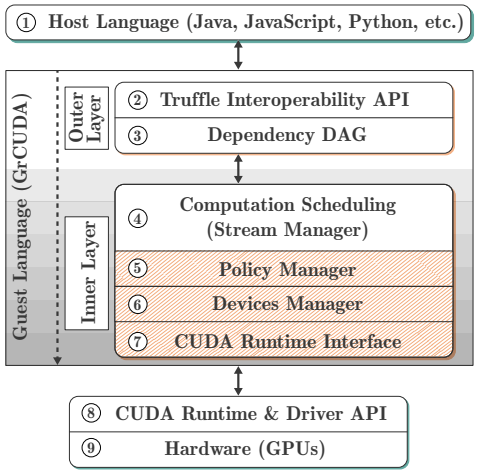

Fig. 3. The two-layer architecture of GrCUDA and interaction with the host language and CUDA runtime. We highlight the components that we added or extended to provide multi-GPU scheduling.

connections that depends on the GPU pair. With the abundant bandwidth of NVSwitch, one can aggressively parallelize tasks between devices, even if strong GPU-GPU communication is present. On the other hand, when using a system based on NVLink, one has to consider the heterogeneous interconnection topology and the affinity between devices: GPUs are not equivalent, and using a specific set of GPUs for a given computation might be better than another. Indeed, depending on where data is located, transferring the same amount of data results in different transfer times depending on where the GPUs are located in the system architecture. Thus, modern techniques for multi-GPU scheduling must be aware of the interconnection topology between devices [2], [4].

B. The GrCUDA Language Binding

GrCUDA [12] is a GPU programming framework initially developed by Nvidia targeting the GraalVM platform [14]. It consists of a CUDA language binding implemented as a language-independent library using the Truffle language implementation framework [17].

Figure 3 shows a high-level representation of GrCUDA's components. Thanks to its design, GrCUDA is used as a domain-specific guest language and can be exposed to any of the host languages supported by GraalVM (including JavaScript, Python, Ruby, WASM, etc.) transparently. The Truffle's interoperability layer (Figure 3, ②) exposes to the host language the basic elements of GrCUDA, such as GPU arrays or executable GPU kernels as *interoperable objects*, so that the host language can manipulate these elements as if they were objects of the host language itself. For example, the host language can declare and access arrays used by the GPU, or execute GPU kernels like any other function.

At its lowest level, GrCUDA directly interacts with the NVRTC compiler [24] and with CUDA's runtime and driver APIs [25] ⑧, which schedule and manage the execution of computations on the GPUs available in the system ⑨.

C. The GrCUDA Scheduler and Runtime

The original open-source Nvidia GrCUDA runtime has been extended in [18] to provide asynchronous and transparent single-GPU scheduling. The main design choice of this extended GrCUDA runtime is to have two separate layers. The outer layer provides the interface with the host language (Figure 3, ②), the computation of the dependency DAG ③, and wraps individual computations supported by GrCUDA (e.g. GPU kernels) in a way that only details relevant to the data dependency computations are exposed. GPU kernels, memory accesses by the CPU to *device arrays* (backed by CUDA's Unified Memory), and functions from libraries such as cuBLAS [26] or cuSPARSE [27] are all wrapped into *computational elements*. The input (and output) arguments of each computation may introduce task or data dependencies that may result in excessive contention and inefficient synchronization. Such data and task inter-dependencies may result in severe performance degradation, with a very negative impact on the overall efficiency (parallelism degree). GrCUDA uses a data-flow DAG to track these dependencies and guarantee the maximum amount of task-level parallelism while preserving correctness. It automatically infers data dependencies and models them through a DAG, instead of having the user manually specify dependencies. The managed execution environment of GrCUDA makes inferring dependencies possible. Inputs are encapsulated into objects, removing the risk of pointer aliasing typical of native languages (multiple pointers that refer to the same memory area). GrCUDA's object-oriented memory model also ensures that all valid inputs to a CUDA kernel in GrCUDA are tracked correctly: one cannot pass arbitrary pointers to other objects.

The inner layer (Figure 3, ④ to ⑦) interacts with the CUDA runtime, and it leverages the information extracted from the dependency DAG and the status of the available devices to assign streams to GPU computations, synchronize computations when necessary, and transparently provide asynchronous execution. GrCUDA's original layers, providing space-sharing and transfer-computation overlap for single-GPU computations, are extensively explained in [18]. §III-A presents the enhancements we propose to enable the efficient scheduling of multi-GPU applications by masking the complexity of inter-GPU data-transfer management.

D. The Case for Greedy Multi-GPU Scheduling

Scheduling, in our context, means identifying the best GPU on which to execute the GPU kernels of a target application, and maximize asynchronous computations and data movements to minimize the overall execution time of the application. This optimization process requires knowledge of the target architecture and infrastructure, and an understanding of the data dependencies in the target application and possibly of the computations themselves. For this reason, writing multi-GPU programs by hand is error-prone, with frequent non-deterministic issues introduced by asynchronous behaviors that are hard to debug.

We say that a scheduling process is *greedy* if it does not require the user to specify in advance the structure of the com-

putations (in terms of data dependencies, data sizes, and other constraints). Non-greedy schedulers need prior information specified at compile-time or with some explicit *compilation* function in the code, like CUDA Graphs [28] or the original TensorFlow [8]. When the structure of the computation is fully defined in advance, this information can be leveraged to produce high-quality schedules and holistic optimizations. However, greedy scheduling provides clear advantages overall.

First, greedy scheduling offers **flexibility** with respect to the structure of the program not being known in advance. Control flow can change dynamically, and there is no need to know in advance the value of conditional expressions or loop iterations to optimize the scheduling. We observe how a similar approach has made PyTorch [9] the prevalent Deep Learning (DL) framework in research, where flexibility is paramount, and that TensorFlow has introduced eager execution to address the same needs [29]. However, these libraries provide parallelism mostly in the form of dividing data across devices, instead of relying on more advanced scheduling optimizations.

Second, greedy scheduling opens the door to **just-in-time scheduling optimizations**. For example, if the output size of a computation is not known in advance (as it occurs in database workloads), it might be impossible to optimize fully the scheduling of other computations that require this output, since the transfer time of this data to other devices depends on its own size. While this situation is less prevalent in DL applications, GrCUDA is not specialized for a single domain and aims to be a *solution for general-purpose GPU programming*.

This work extends the GrCUDA low-profile runtime to automatically leverage multiple GPUs in multi-task computations. Our scheduler aims to transparently provide speedups comparable to what an expert programmer can achieve by hand, making multi-GPU computations easier to approach while minimizing performance compromises.

III. Workload Scheduling Across Multiple GPUs

To simplify the approach to multi-GPU programming, this work brings efficient transparent multi-GPU scheduling to GraalVM. In this section, we describe the architecture of the GrCUDA multi-GPU scheduler we obtained by extending the asynchronous scheduler proposed in [18] (§III-A). Then, §III-B presents the methodology we employ in our scheduler to select the right GPU for each computation, by tracking data locality and employing multiple scheduling policies that implement different heuristics.

A. The Multi-GPU GrCUDA Architecture

Our multi-GPU scheduler for GrCUDA integrates directly with the original architecture, extending the lower level of the runtime without changing the API and the dependency DAG computation. As such, users do not have to modify their existing code to leverage multi-GPU scheduling. Moreover, any future improvement to GrCUDA's outer layer (e.g. the DAG computation or the user-facing API) can be done transparently to multi-GPU scheduling, resulting in higher flexibility and forward compatibility.

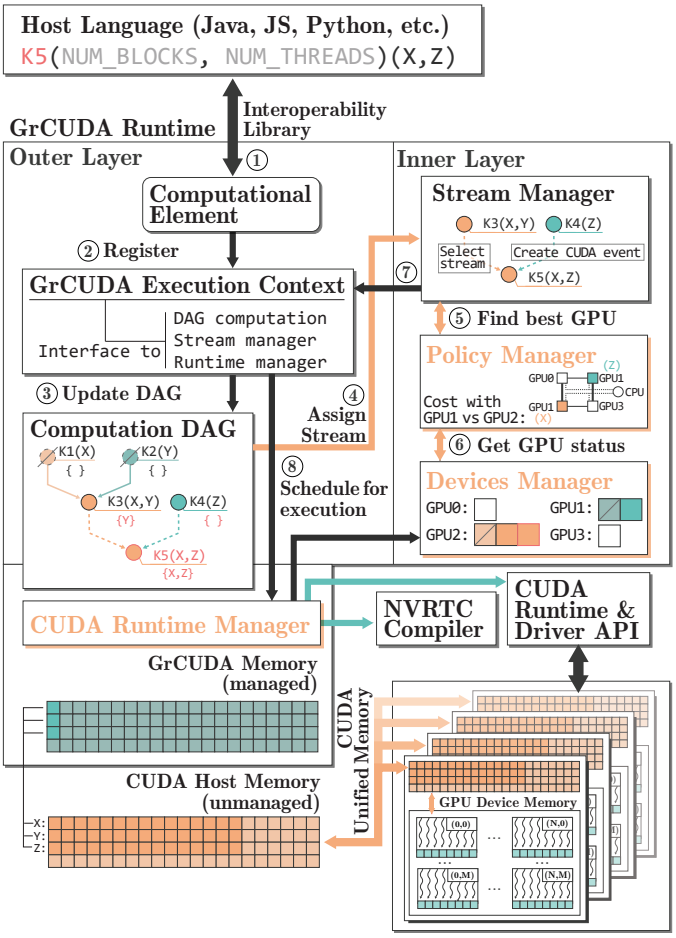

Fig. 4. Detailed architecture of the multi-GPU GrCUDA architecture. New or extended components are highlighted.

The main components of our multi-GPU scheduler for GrCUDA, as shown in Figure 3 and Figure 4, are the following:

Dependency DAG: GrCUDA tracks data dependencies between computations using a DAG. For each computation, we infer what input and output arguments can create data dependencies and update the DAG as computations finish execution or new computations are scheduled.

Computation scheduling (Stream manager): GrCUDA achieves asynchronous execution by assigning GPU computations to different CUDA streams [30]. This component selects the target stream with heuristics provided by the *policy manager*, and it coordinates streams using CUDA events [31].

Policy manager: CUDA streams are uniquely associated to a GPU. As such, we modified the scheduling heuristics of GrCUDA to select the best device for each new computation, using information such as data locality and the current load of the device. This component encapsulates all these heuristics, further defined in §III-B3.

Devices manager: this new component encapsulates the status of the multi-GPU system. It tracks the currently active GPUs, the streams and the currently active computations associated with each GPU, and what data is up-to-date on each device (including the CPUs)

CUDA runtime interface: this component provides access to the CUDA API. We extended it with APIs for selecting and managing multiple GPUs.

Figure 4 provides a more detailed view of Figure 3 by showing the execution flow of a GrCUDA computation scheduled from the host language. The outer layer of GrCUDA is unmodified from the original single-GPU asynchronous scheduler proposed in [18]. Users call a GrCUDA computation in the host language like a regular function (Figure 4, ①). This computation is wrapped into a computational element and passed to the *execution context* for scheduling ②. The execution context infers data dependencies for the new computation based on its input and output arguments, and updates the DAG that tracks the global execution status ③. To infer data dependencies, GrCUDA associates with each computational element a dependency set that initially contains all arguments of the computational element. An argument is removed from the set if a subsequent computation uses and modifies that argument, defining a data dependency on it. Once a set is empty, the corresponding element can no longer introduce dependencies to subsequent computations. The scheduler uses optional argument annotations (i.e. `input`, `const`) to optimize computations with read-only arguments, which are ignored in the dependency computations, if possible. GrCUDA accomplishes these steps without any notion of the underlying GPUs status. Instead, the DAG is provided as input to GrCUDA's inner layer, which is responsible for mapping the abstract data dependencies into a concrete stream and device assignment ④, and for synchronizing previous computations if necessary to guarantee correct results. The stream and device used for the computation are chosen by the policy manager ⑤, which combines information about the status of each GPU ⑥ with information about the data required by the computation (from the DAG). Details about the implemented scheduling policies are provided in §III-B3. Finally, the stream is assigned to the computation ⑦, and the computation is scheduled for execution using the CUDA API ⑧.

B. Selecting the Right GPU

As introduced in §III-A, in our scheduler, each CUDA stream is uniquely associated with a GPU. When using a single GPU, all streams are functionally equivalent, while in the case of multi-GPU scheduling, the choice of a CUDA stream also implies the selection of a GPU. In this subsection, we first describe the challenges we had to face while developing our greedy multi-GPU scheduler, providing observations about how the behavior of the Unified Memory (UM) impacts the device selection [32]. Then, we explain how our system tracks data locality and handles data transfers to optimize an application's performance while guaranteeing correct results. Finally, we illustrate how our scheduler assigns a stream to a computation, leveraging different device selection policies.

1) Challenges of Multi-GPU Scheduling: Given a *multi-task workload*, i.e. a GPU application composed of multiple computations, and a multi-GPU system, our goal is to schedule each computation on a specific GPU and minimize the total execution time of the workload. This optimization problem demands balancing *inter-device data-transfer* and *computational load*, two faces of the same coin. On the one hand, we need to minimize data transfer between devices, as interconnection bandwidth is at least one order of magnitude lower than the GPUs' memory bandwidth. On the other hand, we want to maximize the number of GPUs that we use to evenly distribute the computational load in our system. Moreover, there is not a single universal scheduling strategy that can provide optimal results for every workload and multi-GPU system. As such, we propose an array of different automated multi-GPU scheduling policies, and provide heuristics that can balance the optimization of transfer time and computational load.

Using UM simplifies maintaining data coherence between devices and lifts us from the burden of scheduling explicit data transfers for small synchronization updates. However, CUDA's heuristics to synchronize data between devices are opaque and cannot be completely relied upon if the goal is to maximize performance. Moreover, data transfer through UM relies on page faults and virtual address translation, adding a small overhead to each transaction between devices. This overhead can become significant when transferring small amounts of data repeatedly, for example, when synchronizing results between computations partitioned on multiple devices [33]. The choice of the right GPU for a given computation is thus extremely important, as even in situations where the device selection seems to be irrelevant, the heterogeneous topology of the system's interconnections mixed with the black box CUDA's heuristics, can greatly skew the result. For example, if two devices require to transfer the same amount of bytes, it could still happen that one has better bandwidth thanks to its faster interconnections with the other devices where data are present. To overcome the lack of details about CUDA's heuristics to synchronize data between devices, we made our scheduler able to prefetch data transparently to the user, leveraging the inferred data dependencies. This choice aims to minimize the number of data transfer and synchronization events relying on page faults. To achieve that, CUDA API calls that manage and migrates memory region (i.e. `MemPrefetchAsync`, `MemAdvise`, and `StreamAttachMemAsync`) are added to the computational DAG by the runtime. Moreover, we also expose those CUDA APIs to GrCUDA developers, so they can explicitly leverage them to eventually improve their applications' performance.

2) Tracking Data Locality: The optimal computation-device mapping is strongly dependent on maximizing data locality, i.e. scheduling computations on GPUs that already contain the up-to-date data required as input for the computation. Maximizing data locality has the effect of minimizing the time spent transferring data between devices, a major bottleneck of multi-GPU workloads. This observation holds true as well for data transfer between the CPUs and the GPUs. The structure of the PCIe tree penalizes concurrent transfers of the same data to multiple devices as the total PCIe bandwidth is split among devices. Moreover, multi-socket GPU systems require data transfer between CPUs when moving data from a CPU to a GPU connected to another CPU.

As such, for each argument involved in the computation, we track which devices (including the CPU) have up-to-date values through a *custom MSI-like coherence protocol*. Through

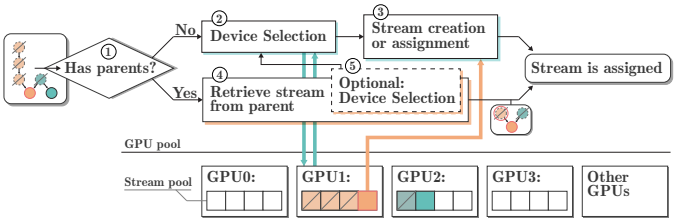

Fig. 5. Sequence of operations taken by our multi-GPU scheduler to assign a device and a CUDA stream to an input GPU computation.

this information, the policy manager can optimize the device selection by minimizing the amount of data to transfer or the estimated transfer time. If an argument (e.g. an array) is used by a computation, the device where the computation is scheduled is the only one having up-to-date values for the argument. If the argument is marked as *read-only* (e.g. when the CPU reads the result of a kernel), we do not reset the list of existing up-to-date locations, but add the device where the computation is scheduled to the list. We update the data locality status of arguments as soon as the computation is scheduled for execution, not after the computation has finished execution. The choice of updating the status as early as possible ensures that we avoid unnecessary repeated transfers, as consecutive computations have immediate access to the current location status instead of having to wait for a synchronization that might happen much later in the execution. Independently of its size, each argument is treated as a whole, always considering the entire allocation block associated with that argument. This is motivated by the fact that CUDA does not expose the status of individual pages, and tracking them ourselves would be too expensive. Moreover, such an approach aligns with our choice of letting users control the logic of data partitions in their applications, which is better motivated in §IV-E. Finally, it is worth mentioning that the device manager does not need to explicitly handle which data to evict when a device runs out of memory since CUDA's UM automatically handles that issue with page fault handling [34].

3) Multi-GPU Scheduling Policies: For any given GPU computation, our scheduler has to select the device that executes this computation and the CUDA stream (on this device) to which the computation is assigned. While selecting the device requires considerations about data locality and workload distribution, the choice of the stream also requires inspecting the *parents* of the computations, i.e. the computations from which there is a data-dependency into the current computation (the in-neighbors of the computation in the DAG). That's because computations scheduled on the same stream (and thus, on the same device) are implicitly synchronized (i.e. a new computation starts only after the previous one on the same stream has finished), while computations on different streams require the GrCUDA runtime to perform synchronizations using CUDA events [31]. Moreover, we know that the computation can start only after its parents have finished running to avoid data conflicts and that GrCUDA always schedules independent computations on different streams. As such, it is likely that parents' streams (and their associated devices) will be free by the time the new computation starts, and not reusing one of the existing streams would not provide any benefit. Finally, creating new streams instead of reusing existing ones also introduce small overheads. While the overheads of these operations are in the order of milliseconds at worst and are often negligible, it is better to avoid them whenever possible by reusing the parents' streams [30].

Thus, in our multi-GPU scheduler, assigning a stream to a computation requires the following operations, presented in Figure 5. First, we distinguish between computations with parents and computations with no parents (Figure 5, ①). If the computation does not have parents, we first select the GPU where the computation will run, using one of the policies described below ②. After selecting the device, we select one of its unused streams (if it exists), or create a new stream otherwise ③. If the computation has parent computations, we can leverage information about them and possibly reuse one of their streams ④. For this step, we adopt the following heuristic. We retrieve the set of GPUs associated with the parents' streams. Then, we compute the optimal device among the retrieved GPUs using the same policy for GPU selection, explained below. We might not be able to reuse existing streams, for example, if other children computations are already waiting to start on those same streams (e.g. two computations that share the same three children; only two of them can reuse the parents' streams). In this case, we employ the same strategy used for computations without parents ⑤. Regardless of which policy is chosen, the computational element is provided with a CUDA stream and an associated GPU where the computation will run. It is worth noticing that the assignment to a CUDA stream happens when the task is first scheduled. We schedule workloads based on the estimated transfer size (or time), which is inferred by looking at the input and output arguments' sizes that are known at schedule time (without waiting for dependee tasks to complete). This approach also allows to prefetch data that are required by the dependent task but not by the dependee, while the dependee task is still running.

We implemented 4 different policies for device selection to guarantee a certain level of flexibility to our scheduler, and to investigate the impact of more informed heuristics on the runtime performance of various workloads. In detail, we provide the following policies:

- **Round-Robin:** simply rotate the scheduling between GPUs. Used as a baseline and as an initialization strategy of other policies.
- **Stream-Aware:** assign the computation to the device with the fewest busy streams. As each stream can run at most a computation at a time, this policy is equivalent to selecting the device with fewer ongoing computations. It tries to distribute the workload evenly between devices.
- **Minimum Transfer Size:** select the device that requires the least amount of bytes to be transferred, maximizing data locality.
- **Min-Max Transfer Time:** considering just the amount of data to be transferred is often misleading due to the devices' *heterogeneous interconnection topology*. Even if a GPU requires more bytes to be transferred than other

GPUs, this device could still be optimal if it has faster interconnections to the devices from which data is copied. As the heuristic used by CUDA to transfer data in UM is not publicly known, we assume that data is copied from the data sources with the lowest bandwidth, and we select the device for which the total transfer time would be minimum. In other words, the policy minimizes the maximum possible transfer time. Assuming that data is copied from the data source with the largest bandwidth did not show performance differences.

To formalize this last policy, consider two devices $d_i, d_j \in \mathcal{D} = \{CPU, GPU_0, GPU_1, \ldots, GPU_N\}$. B_{ij} is the bandwidth between d_i and d_j, while $\mathcal{A} = \{A_1, A_2, \ldots, A_N\}$ is the set of arguments required by the computation to be scheduled. S_a is the size in bytes of arguments $a \in \mathcal{A}$. Assume $B_{ii} = \infty$, and $B_{ij} = B_{ji}$. The *Min-max Transfer Time* policy selects a device $d_j \neq CPU$ such that $\text{argmin}_{d_j \in \mathcal{D}} \{\sum_{a \in \mathcal{A}} S_a / \min_{d_i \in \mathcal{D}} \{B_{ij}\}\}$. Information about interconnection topology and speed is firstly computed during the installation of GrCUDA on a specific multi-GPU system and then retrieved at runtime. Such information is encoded in a dense interconnection graph with bandwidth-weighted edges between all couples of devices in the architecture, and it can be updated upon the user's request, even at runtime. Indeed, we support multiple architectures and interconnection topologies (e.g., NVLink, NVSwitch) without any explicit optimization.

All these new scheduling policies integrate smoothly with the existing GrCUDA scheduling options, such as the ability to reuse free streams or prefetch data to optimize large data transfer. It is worth noticing that scheduling is a serial process. Thus, policies aiming at reducing data transfer size or time could suffer from early convergence and, consequently, load imbalance. To avoid such a scenario, we have implemented an **exploration-exploitation heuristic** in the two data-aware policies, where we do not consider devices with an amount of already available data inferior to a threshold percentage. In particular, we used a threshold value of 10% in our experiments, described in §IV. This means that if a device has less than 10% of the total amount of data required by the computation we are scheduling, we consider that it has no data available at all. This choice prevents the early saturation of the GPU(s) where the first computational elements were scheduled, which would result in the under-utilization of the available devices and, consequently, in sub-optimal performance.

To help users to alleviate hot spots and congestion that some applications might introduce, we allow developers to export the computation DAG of their applications, as obtained with the selected scheduling policy. This information can be leveraged to better understand the achieved performance and to compare the schedules derived from different policies. Moreover, independently of the selected policy, poorly written applications will result in DAGs with low-level of task parallelism, suggesting designers to change their applications' logic.

Another important aspect is that we offer users the ability to profile the execution and track historical information to help the creation of *novel optimizations* on top of our scheduler. Users can easily add different policies to the *policy manager* to create new streams and associate them with computations. For example, one could introduce domain-specific memory management policies (e.g., for sparse and graph computations), if it is known that some classes of applications present distinctive movement patterns that can be leveraged.

IV. Experimental Evaluation

Our performance evaluation relies on a set of multi-GPU benchmarks exhibiting task-level parallelism and leveraging CUDA kernels taken or derived from open-source implementations. We explicitly designed such a benchmark suite to analyze the performance of our novel multi-GPU GrCUDA scheduler under different workloads. In §IV-B, we provide a workload characterization for each benchmark, measuring the amount of data transferred between different devices (both CPUs and GPUs) available in the systems. Then, we evaluate the scalability of both CUDA C++ and GrCUDA implementations (using Python and Java as host languages) when running on up to 8 GPUs, and we compare the performance of GrCUDA against the CUDA C++ API, showing how we can achieve 80–90 % of the peak performance while significantly lowering the development effort (§IV-C). In §IV-D we analyze the impact of the proposed scheduling policies on the achieved performance, showing the advantages of more refined policies for workloads exhibiting complex data movement patterns. Finally, we demonstrate the benefits of separating logical from physical data partitioning, supporting our design choice of letting users control the partitioning logic but lifting them from the onerous task of managing individual devices.

A. Experimental Setup

All experiments are conducted on two different machines available on the OCI platform. The two servers were chosen because they feature *different GPU architectures* and - more importantly - *different interconnection technologies*, allowing us to evaluate the capabilities of our scheduler to adapt to various system topologies. In detail, the first machine is equipped with 8 Nvidia Volta V100 GPUs (84 SMs, 16 GB global memory) paired with dual Intel Xeon Platinum 8167M CPUs and 768 GB of RAM. Since each V100 GPU only has 6 NVLinks, the GPUs are connected in a hybrid cube-mesh network topology, where each GPU is connected to two GPUs through a double NVLink-V2 ($\sim 50\,\text{GB}\,\text{s}^{-1}$ of unidirectional bandwidth), and to other two GPUs through a single NVLink-V2 ($\sim 25\,\text{GB}\,\text{s}^{-1}$). The communication with other GPUs in the system relies on PCIe 3.0 ($\sim 7\,\text{GB}\,\text{s}^{-1}$). The second machine is equipped with 8 Nvidia Ampere A100 GPUs (128 SMs, 40 GB global memory) paired with dual AMD EPYC 7542 CPUs and 2048 GB of RAM. Here, each GPU is connected to the CPU through PCIe 4.0, and the 8 GPUs are fully connected via NVSwitch, leveraging the 12 NVLinks ($\sim 300\,\text{GB}\,\text{s}^{-1}$) in each A100. We used Ubuntu 20.04 LTS, CUDA 11.7, and GraalVM CE 21.2.0 for all our experiments. All experiments were repeated 30 times. Our plots report results for each benchmark in terms of arithmetic mean over 27 executions, excluding the first 3 executions for each benchmark (we use them for warm-up). Instead, the

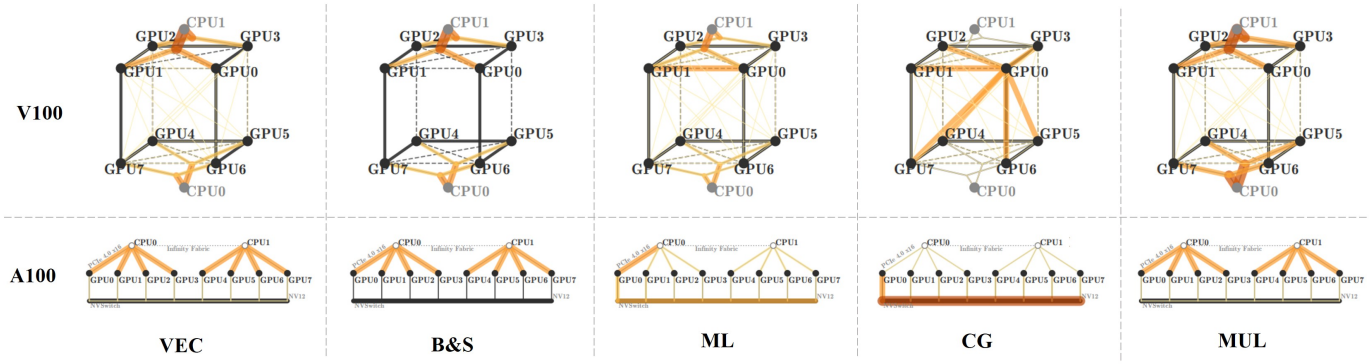

Fig. 6. Amount of bytes transferred between GPUs for each employed benchmark (hand-optimized C++ CUDA version), running on 8 Nvidia Tesla V100s (top) and A100s (bottom), with visual representations inspired by Li et al. [20]. Data transfer is represented by orange lines, where thicker and darker lines identify more data transfer. For the V100s, solid and dashed lines represent dual and single NVLink connections, respectively, whereas in the A100s, the solid line is an NVSwitch network.

MEAN section in the plots reports the geometric mean, which is more appropriate for combining the results of different benchmarks.

In the rest of the paper, we will use the GPU model (A100 and V100) to refer to the respective server configuration.

B. Workloads Characterization

In our experiment, we employed 5 different multi-GPU benchmarks that we implemented to evaluate different aspects of a multi-GPU architecture. First, we extended three of the benchmarks proposed in [18] to exploit all the GPUs present in the system. In particular, we adapted the **Vector Squares (VEC)** benchmark, which computes the sum of differences of two squared vectors and requires intensive data transfer between CPUs and GPUs; the **Black & Scholes (B&S)** equation, a streaming benchmark with multiple independent kernels that can be overlapped; and the **Machine Learning Ensemble (ML)**, a realistic ML pipeline that shows execution-time imbalance in its computation graph. Then, we extended our suite with two additional benchmarks: a **Conjugate Gradient (CG)** solver, a traditional HPC workload with heavy communication between different GPUs (inspired by [35]), and a **Matrix Vector Multiplication (MUL)** routine, which performs a multiplication between a dense matrix and a dense vector, efficiently leveraging data partitioning.

Our benchmarks leverage both open-source kernels and kernels written by us. We wrote the host code for all the benchmarks, and we employed UM both in the CUDA C++ and the GrCUDA versions to have a fair comparison between the two implementations. The CUDA C++ code was hand-optimized with CUDA API calls that manage and migrate memory regions to replicate - as much as possible - what would happen with manual memory management. For what concern the kernels' configuration, the number of blocks and number of threads per block have been optimized for each benchmark to get the best performance in the CUDA baselines. In benchmarks with partitioned data, the number of partitions is constant across input sizes and numbers of GPUs, and it has also been chosen to optimize the CUDA baselines' performance. Indeed, for each benchmark, the same settings have been used for the CUDA baseline and the GrCUDA implementation.

The heterogeneity of the benchmarks is shown in Figure 6, where we present the amount of data transfer for each benchmark running on the full set of 8 GPUs, for both the V100 and A100 systems. The data underlying Figure 6 are taken from the baseline CUDA C++ implementations, so they are the "optimized" data movements. We can classify the benchmarks by looking at which devices the majority of data transfer is addressed: B&S has no inter-GPUs transfer, while CG is at the opposite end of the spectrum with negligible transfer from and to the CPUs. The other benchmarks present a combination of the two behaviors, requiring both CPU-GPU and GPU-GPU communication to different extents. Figure 6 also highlights how the system equipped with V100s has an an heterogeneous GPU-GPU interconnection based on NVLink, while the A100s provide a more homogeneous interconnection based on NVSwitch. Since our automatic scheduler for GrCUDA is designed to work with all the possible interconnection scenarios, we obtain positive results with both the multi-GPU systems used for our tests.

C. Scaling the Number of GPUs

To establish a baseline, we first analyzed the performance scalability of the hand-optimized C++ CUDA version of the benchmarks, when running on up to 8 GPUs. Figure 1 depicts the speedups of the 5 benchmark with respect to an asynchronous version on a single-GPU system. The first bar of each group (SYNC, 1 GPU) also provides the performance of an unoptimized synchronous CUDA version of the benchmarks. The MEAN section for both the A100 and the V100 configurations in Figure 1 shows that most of the considered benchmarks can scale across multiple GPUs. However, the GPU architecture, as well as the communication patterns, have a substantial impact on the achieved performance. When using the V100 system, ML and CG were not capable of taking advantage of the full 8 GPUs, resulting in slowdowns due to the required data transfer between GPU couples with low

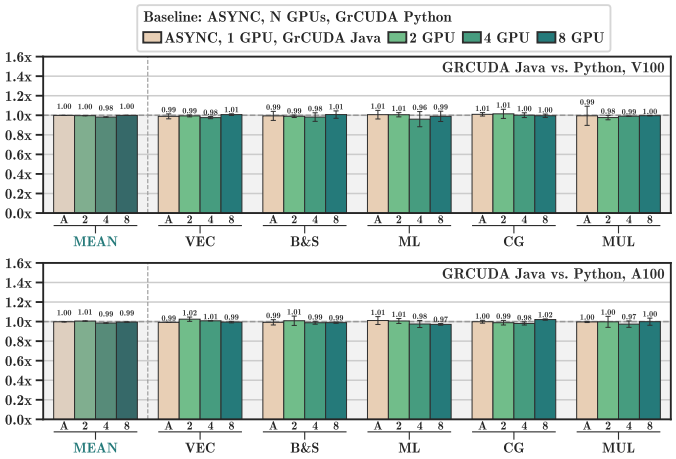

Fig. 7. Relative performance (speedup) of our automatic GrCUDA scheduler from Python vs. Java host code. Our scheduler shows consistent results in all benchmarks when employed from both host languages.

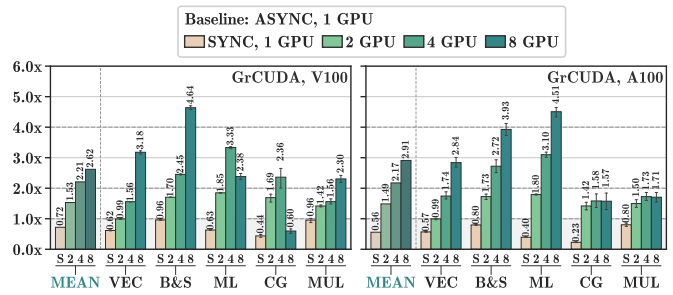

Fig. 8. Multi-GPU scaling of benchmarks when using our automatic GrCUDA scheduler, on up to 8 Tesla V100s and A100s. Speedups closely resemble the ones for hand-optimized scheduling C++ CUDA (Figure 1).

interconnection bandwidth. Instead, the homogeneous high-bandwidth interconnection provided by the NVSwitch enables the ML benchmark to perform significantly better with the full set of GPUs on the A100 system, achieving a 5.2 × speedup. On the other hand, when using the full set of A100s, CG only achieved the same performance as with fewer GPUs. This is due to the fact that CG is bound by heavy inter-GPU communication involving always the same device. Consequently, once the band of that device is saturated, the benchmark cannot benefit from a higher number of GPUs.

Given that our novel scheduler targets GrCUDA, we compared the performance of GrCUDA against the CUDA C++ API. We also compared the performance obtained when scaling the number of GPUs with both the Python and Java versions of the GrCUDA benchmarks. Figure 7 shows that there are only negligible differences in the achieved results, demonstrating that our solution provides **consistent behavior from different host languages supported by GraalVM**. In the remainder of the Section, we will refer to results obtained with Python host code, but our considerations hold for the Java implementations as well.

Figure 8 shows the results of scaling the number of GPUs when using our multi-GPU GrCUDA scheduler with the most informed scheduling policy (Min-max Transfer Time). We compare the timings obtained with our solution to the asynchronous extension of GrCUDA in [18], which supported only a single GPU and is used as the baseline for speedup computation. We also provide the performances of the original synchronous single-GPU version of GrCUDA, which always performed worse than the asynchronous one, with up to ∼80 % slowdown in the case of CG on the A100 architecture. Figure 8 shows that our novel scheduler can efficiently leverage the available accelerators: when running on 8 GPUs, it achieves **up to 4.7 × speedup on the V100** server, and **up to 4.6 × speedup on the A100** one. More importantly, the speedups obtained by our multi-GPU scheduler closely resemble their hand-optimized CUDA C++ counterpart (Figure 1), for both the tested systems.

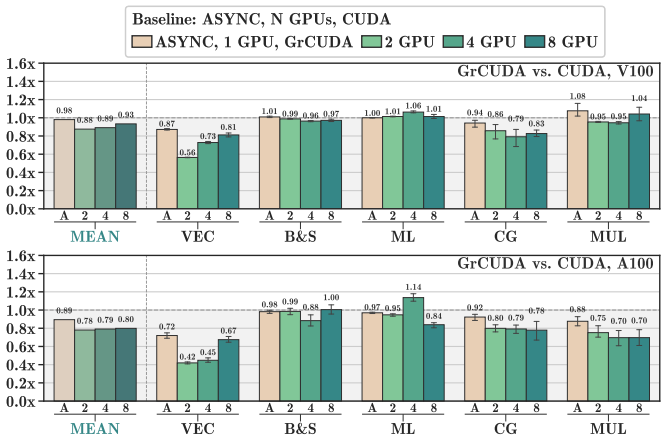

Fig. 9. Relative performance (speedup) of our automatic GrCUDA scheduler vs. hand-optimized C++ CUDA. We automatically achieve 80–90% of the C++ CUDA peak performance, with only VEC showing a noticeable performance gap.

We further explore this aspect in Figure 9, where we compare the relative speedup of our novel GrCUDA scheduler against the pure CUDA versions, when employing a certain number of GPU on both the V100 and A100 systems. Here a speedup close to 1.0 × highlights that our solution introduces minimal overhead compared to a meticulously optimized native CUDA version of the software. The figure shows how, on average, we **automatically achieve 80 % of the hand-optimized C++ CUDA peak performance on the A100** system, and **90 % on the V100** system, with only VEC showing a noticeable performance gap on both the tested architectures. VEC highlights a corner case where hand-tuned optimizations can still outmatch automatic scheduling. This benchmark performs three simple but interdependent linear algebra operations and is fully interconnection-bound. Optimal scheduling requires knowledge of both the execution time and the data movement patterns of each computation. As a consequence, VEC is noticeably worse in GrCUDA because in our scheduler there is some unnecessary GPU-GPU movement, while the hand-optimized version has none. History-driven or ML-based schedulers can, in principle, address this issue, although their additional overheads and complexity might

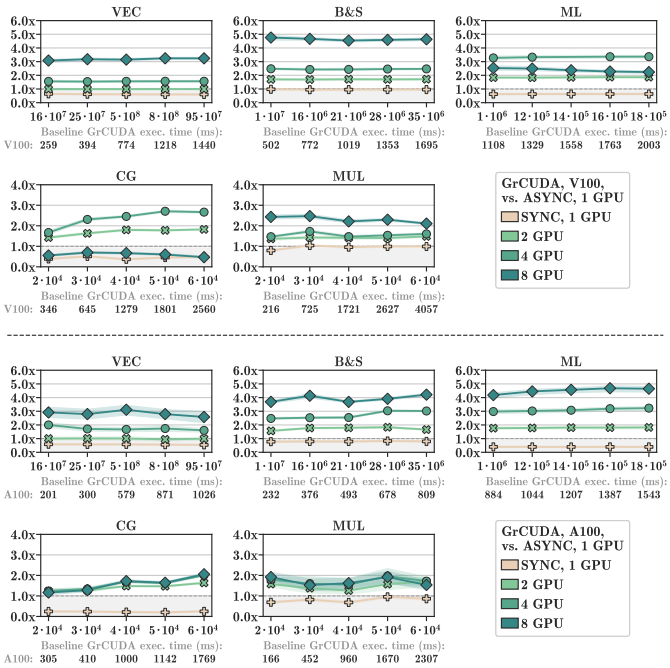

Fig. 10. Relative execution time of our automatic GrCUDA scheduler vs. hand-optimized C++ CUDA, divided by input size. Our scheduler shows consistent results when increasing the benchmarks' memory footprint from $\sim 10\,\%$ to $\sim 90\,\%$.

make them unwieldy under most circumstances.

To investigate the robustness of our scheduler with respect to the size of the input data, we studied the performance of each benchmark when scaling the input size. Figure 10 presents the results of our analysis, with input size scaling linearly from $\sim$10 % to $\sim$90 % of the memory available on each GPU. This allows us to clearly visualize if any hardware bottleneck impacts performance as input size exceeds a certain threshold. The figure shows that all the obtained results are consistent with respect to the findings presented in Figure 8. This indicates that our new greedy GrCUDA scheduler for multi-GPU computations is indeed capable of handling different types of workloads in multiple scenarios, and **can adapt to different input data sizes without performance degradation**.

Overall, our proposed scheduler is capable of relieving the developer of the time-consuming need to write optimized CUDA C++ host code to exploit a multi-GPUs system.

D. Impact of the Scheduling Policies

Having introduced a set of multi-GPU scheduling policies in GrCUDA, we now analyze how more refined scheduling techniques can result in better overall performance. In Figure 11, we compare our policies against the Min-Max Transfer Time policy, our best-performing policy on average. Results are measured on an 8-GPUs system. The simple Round-Robin policy obtains a *33 % slowdown on average*, and shows significant performance degradation for benchmarks with significant GPU–GPU communication, such as ML and the CG. Their complex data movement patterns, shown in Figure 6, require more refined scheduling policies. The Stream-Aware policy shows a *18 % slowdown on average*, mainly due

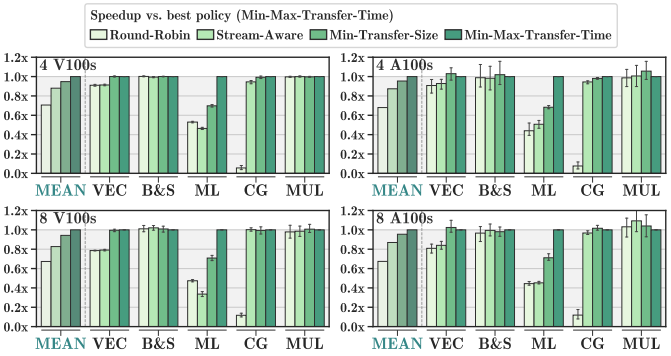

Fig. 11. Ablation study for our scheduling policies in GrCUDA, with speedup (higher is better). Computations with complex dependency DAGs, such as ML, greatly benefit from transfer-aware policies, with the simplest Round-Robin policy being 33 % slower on average.

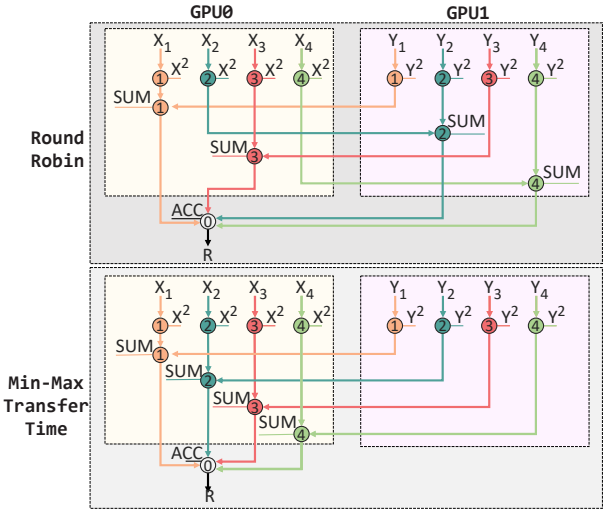

Fig. 12. The DAGs obtained when executing the VEC benchmark on 2 GPUs, with two different scheduling policies (Round-Robin and Min-Max Transfer Time). For the sake of simplicity, in this example, we divide the data into 4 partitions only.

to the performance degradation of the ML benchmark. For such a benchmark, the Stream-Aware policy performs better than Round-Robin on the A100 system with homogeneous interconnection, while it performs worse on the V100 system with heterogeneous interconnection. The two simplest policies evenly distribute the workload across GPUs but are unaware of the data locality and of heterogeneous connections between the devices in the systems. They cannot efficiently handle workloads with complex data dependencies like VEC and ML. These results highlight the *need for more informed scheduling techniques*, like our Min Transfer Size and the Min-Max Transfer Time policies. Figure 11 shows that, on average, **the two data-aware policies performed better than the simpler ones**. Moreover, they performed comparably in all the benchmarks, except for the ML one, where the Min-Max Transfer Time is significantly better.

To better understand the performance obtained with our ablation study, Figure 12 shows the schedules obtained when executing the VEC benchmark (with four partitions) on two

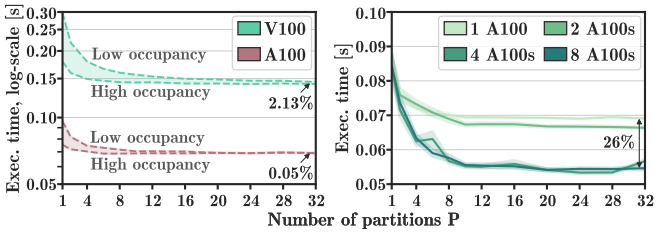

Fig. 13. Benefits of transfer-computation overlap in matrix-vector multiplication (MUL), for increasing number of data partitions and GPU configurations with different occupancy (*left*), and using multiple GPUs (*right*).

GPUs, using the Round-Robin and Min-Max-Transfer-Time policies. We only report these two schedules for the sake of brevity, given that the Stream-Aware policy produces the same schedule as Round-Robin, while the Min-Transfer-Size produces the same schedule as Min-Max-Transfer-Time. We can see how the two simpler policies simply try to distribute the computation across the two available accelerators evenly. On the other hand, the two data-aware policies tend to maximize the usage of `GPU0` to better leverage the data locality. This behavior explains the better performance obtained when using the two more refined policies, as illustrated in Figure 11.

E. Disjoining Logical and Physical Data Partitioning

One could say that having to manually specify data partitions limits the level of abstraction provided by the multi-GPU scheduling technique in this work. Intuitively, data partitioning is linked to the number of devices: with four devices, it sounds reasonable to have four data partitions. We argue that this is often not the case, and providing *logical separation between data partitioning and the underlying number of devices*, instead of hiding partitioning from the user, is beneficial.

Let's consider the MUL benchmark, performing a multiplication between a dense matrix $\mathbb{R}^{n \times m}$ and a dense vector $\mathbb{R}^m$. A simple data partitioning scheme is to divide the matrix by rows, in P submatrices each of size $\mathbb{R}^{(n/P) \times m}$. In principle, if we have D devices, it makes sense to have $P = D$, as $P > D$ would not provide any additional parallelism between tasks. However, the picture is different if data transfer and hardware utilization are kept into account. Even on a single device, having $P > 1$ gives better performance than $P = 1$.

We illustrate this behavior in Figure 13, using a 20000×20000 dense matrix. On the left, we show how **data partitioning reduces the need to hand-optimize the GPU grid configuration**. We compare the performance of a low-occupancy GPU configuration that underutilizes the GPU resources with a hand-optimized configuration that exploits the GPU processors fully, even with a single partition. For the V100 GPU, the performance gap shrinks within 2 % as the number of partitions increases (from the initial ∼50 %), as low occupancy kernels are overlapped, and the computation is masked by data transfer. Indeed, the high-occupancy configuration cannot overlap multiple kernels as each one is already able to fill the GPU processors. The speedup, in this case, comes from overlapping transfer and computation, hiding the cost of the latter. A comparable behavior is shown for the A100 GPU, which presents a smaller performance gap between low-occupancy and high-occupancy configurations, independently of the number of partitions. On the right, we show the execution time when parallelized across multiple GPUs. Using 8 GPUs and 12 to 32 partitions provides $\sim 65\,\%$ speedup versus a single-partition, single-GPU configuration. However, 8 GPUs are only 26 % faster than a single GPU using the same number of partitions. The bandwidth of PCIe is shared between GPUs, preventing higher speedups in this transfer-bound computation. On the other hand, data partitioning on a single GPU provides almost the same gain as moving from 1 to 8 GPUs without partitioning input data.

Letting users control the logic of data partitioning, and lifting them from the onerous task of managing individual devices, provides the *optimal balance between control and abstraction*. Automatically overlapping computations when the GPU is underutilized due to sub-optimal grid configurations helps reducing the burden of GPU code optimization, with less effort required to reach satisfactory performance.

V. Related Works

Automatic multi-GPU scheduling has been previously investigated in the context of domain-specific programming framework. This resulted in the expansion of DSLs to support scaling the computation on multiple devices, especially in the domain of DL, with frameworks such as TensorFlow [8], [29] and PyTorch [9]. As our work provides a domain-agnostic solution for increasing the performance of multi-GPU applications, we do not compare directly against those solutions that have design choices strictly connected to their domains.

Several researchers proposed workload-independent solutions that offer support for multi-GPU computing, like XKaapi [36]. Opposed to our approach, it calculates the dependencies only when it encounters an idle thread, and it does not take advantage of the UM [32] to manipulate data. The project is also no longer maintained, and it supported only pure C++ code. Like XKaapi, OmpSs [37] is a programming model based on C++ that supported multi-GPU automatic scheduling of tasks on CUDA/OpenCL devices. Its development has been discontinued since its underlying structure was not proving to be anymore fast enough with respect to other solutions. A new version of the back-end, OmpSs-2 [38], was released but misses the multi-GPU support of the original one. The famous OpenMP API enables shared-memory parallel programming on many architectures, including GPUs. It has bindings for C/C++ and Fortran languages, and it allows multi-GPU paradigms [39] but requires abstracting the computation, as it does not support the direct usage of CUDA code.

Another worthy solution based on GraalVM is TornadoVM [40], [41]. It is an additional layer of virtualization that can be added on top of GraalVM to allow specifically annotated Java programs to run on heterogeneous hardware. It has been recently extended to support Multiple-Tasks on Multiple-Devices (MTMD) for OpenCL compatible devices [42], but their solution does not include refined topology-aware scheduling policies. Moreover, their focus is around the

optimization in the usage of consumer available devices with respect to classical resources (e.g. integrated GPUs and multi-core CPUs in laptops), rather than optimizing the scheduling of compute-intensive application in a multi-GPU system.

To the best of our knowledge, the extension of GrCUDA we propose provides the first multi-language solution to increase the performance of CUDA programs by automatically handling scheduling on multi-GPU architectures without any additional know-how required to the developer. By taking advantage of the GraalVM runtime we support a wide range of programming languages providing a solution that can be easily adopted in real-world scenarios.

VI. Conclusion and Future Work

This paper presented a novel scheduler for multi-task computations that provides transparent asynchronous execution on multi-GPU systems. Our scheduler computes data location and migration costs at run time to identify the optimal scheduling without requiring prior information about the program dependencies or the underlying system architecture. Our solution directly integrates with the GrCUDA environment, a polyglot CUDA API based on GraalVM that provides easy access to GPU acceleration to languages such as Java, JavaScript, Python, and R. Thus, it provides a general framework that can greatly simplify the approach to multi-GPU programming.

We validated our scheduler on five benchmarks and a total of 24 GPU kernels. In our experiments, our new scheduler provided a speedup of up to 4.7 $\times$ over single-GPU asynchronous scheduling when running on 8 GPUs. More importantly, it automatically achieved 80–90 % of peak performance against hand-optimized CUDA host code, on two different state-of-the-art multi-GPU systems. We focused on Volta and Ampere GPU architectures, currently available on the OCI platform. Nonetheless, our solution could be effortlessly deployed on systems featuring the latest Hopper GPUs by Nvidia [7] to take advantage of technologies such as PCIe 5.0 and NVLink 4. The recent industrial focus on fast interconnections and unified memory space, exemplified by the Nvidia Grace Hopper Superchip [43], further remarks the importance of our analyses. In future work, we plan to analyze the performance of our scheduler on these new architectures and extend our benchmark suite to achieve a deeper performance characterization. We also plan to optimize scheduling through additional run time information, such as choosing the ideal device for a certain computation based on previous executions. Finally, we aim to extend our work to support the concurrent execution of multiple applications to make GrCUDA a viable solution to handle multi-accelerator servers in shared multi-tenant environments.

Acknowledgment

We thank Oracle Labs, Oracle Cloud Infrastructure, and Oracle for Research for their support and contributions to this work. This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle for Research program. The authors from Politecnico di Milano are funded in part by a research grant from Oracle. Oracle, Java, and GraalVM are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] H. Anzt, Y. M. Tsai, A. Abdelfattah, T. Cojean, and J. Dongarra, "Evaluating the performance of nvidia's a100 ampere gpu for sparse and batched computations," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 26–38, 2020.

[2] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder, "Topology-aware gpu scheduling for learning workloads in cloud environments," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[4] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

[5] NVIDIA, "NVIDIA NVLINK - High-Speed GPU Interconnect." [Online] Accessed: 2023-04-20. Available at: https://www.nvidia.com/en-us/design-visualization/nvlink-bridges.

[6] NVIDIA, "NVLink and NVSwitch." [Online] Accessed: 2023-04-20. Available at: https://www.nvidia.com/en-us/data-center/nvlink/.

[7] NVIDIA, "NVIDIA H100 Tensor Core GPU." [Online] Accessed: 2023-04-20. Available at: https://www.nvidia.com/en-us/data-center/h100/.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[10] A. Klöckner, N. Pinto, B. Catanzaro, Y. Lee, P. Ivanov, and A. Fasih, "GPU Scripting and Code Generation with PyCUDA," 2013.

[11] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in *European Conference on Parallel Processing*, pp. 887–899, Springer, 2009.

[12] NVIDIA, "A Polyglot Language Binding for CUDA in GraalVM." [Online] Accessed: 2023-04-20. Available at: https://developer.nvidia.com/blog/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm.

[13] R. Mueller and L. Stadler, "grCUDA: Polyglot GPU Access in GraalVM." [Online] Accessed: 2023-04-20. Available at: https://github.com/NVIDIA/grcuda.

[14] Oracle, "GraalVM." [Online] Accessed: 2023-04-20. Available at: https://www.graalvm.org.

[15] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pp. 187–204, 2013.

[16] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, "Graal IR: An extensible declarative intermediate representation," in *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[17] C. Wimmer and T. Würthinger, "Truffle: a self-optimizing runtime system," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pp. 13–14, 2012.

[18] A. Parravicini, A. Delamare, M. Arnaboldi, and M. D. Santambrogio, "DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 111–120, IEEE, 2021.

[19] R. Budruk, D. Anderson, and T. Shanley, *PCI express system architecture*. Addison-Wesley Professional, 2004.
[20] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.
[21] Oracle, "Oracle Cloud Infrastructure - NVIDIA." [Online] Accessed: 2023-04-20. Available at: https://www.oracle.com/cloud/partners/gpu.html.
[22] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *2010 18th IEEE Symposium on High Performance Interconnects*, pp. 1–6, IEEE, 2010.
[23] AMD, "AMD Infinity Architecture: The Foundation of the Modern Datacenter." [Online] Accessed: 2023-04-20. Available at: https://www.amd.com/system/files/documents/LE-70001-SB-InfinityArchitecture.pdf.
[24] NVIDIA, "NVRTC (Runtime Compilation)." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/nvrtc/index.html.
[25] NVIDIA, "CUDA Runtime API." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.
[26] NVIDIA, "cuBLAS." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/cublas/index.html.
[27] NVIDIA, "cuSPARSE." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/cusparse/index.html.
[28] NVIDIA, "CUDA Graphs." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html.
[29] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga, and S. Cai, "TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning," 2019.
[30] J. Luitjens, "Cuda streams: Best practices and common pitfalls," in *GPU Techonology Conference*, 2015.
[31] NVIDIA, "CUDA Toolkit Documentation - Event Management." [Online] Accessed: 2023-04-20. Available at: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.
[32] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, pp. 1092–1098, IEEE, 2015.
[33] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for GPU accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.
[34] N. Sakharnykh, "Everything you need to know about unified memory," 2018. [Online] Accessed: 2023-04-20. Available at: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.
[35] Nvidia, "Conjugate gradient um." [Online] Accessed: 2023-04-20. Available at: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/4_CUDA_Libraries/conjugateGradientUM.
[36] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Locality-aware work stealing on multi-CPU and multi-GPU architectures," in *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.
[37] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.
[38] B. S. Center, "OmpSs-2 Programming model." [Online] Accessed: 2023-04-20. Available at: https://pm.bsc.es/ompss-2.
[39] V. Kale, W. Lu, A. Curtis, A. M. Malik, B. Chapman, and O. Hernandez, "Toward supporting multi-GPU targets via taskloop and user-defined schedules," in *International Workshop on OpenMP*, pp. 295–309, Springer, 2020.
[40] J. Clarkson, J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, C. Kotselidis, and M. Luján, "Exploiting high-performance heterogeneous hardware for Java programs using graal," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pp. 1–13, 2018.
[41] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis, "Dynamic application reconfiguration on heterogeneous hardware," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 165–178, 2019.
[42] M. Papadimitriou, E. Markou, J. Fumero, A. Stratikopoulos, F. Blanaru, and C. Kotselidis, "Multiple-tasks on multiple-devices (MTMD): exploiting concurrency in heterogeneous managed runtimes," in *Proceedings of the 17th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, pp. 125–138, 2021.
[43] Nvidia, "Nvidia Grace CPU." [Online] Accessed: 2023-04-20. Available at: https://www.nvidia.com/en-us/data-center/grace-cpu/.

Guido Walter Di Donato (Graduate Student Member, IEEE) received his M.Sc. degree in BioEngineering from the University of Illinois at Chicago in 2019; in 2020, he got his M.Sc. degree in Biomedical Engineering from the Politecnico di Milano, where he is a Ph.D. Candidate in Information Technology - Computer Science and Engineering. His research mainly focuses on parallel computing and computer architectures, but his interests also include graph analytics, bioinformatics, and health informatics.

Ian Di Di Lavore (Graduate Student Member, IEEE) is a Ph.D. Student in Information Technology - Computer Science and Engineering at Politecnico di Milano. He holds an M.Sc. (2022) and B.Sc. (2020) in Computer Science and Engineering from Politecnico di Milano. His research mainly focuses on parallel and distributed computing and computer architectures, with a particular interest in HPC scenarios.

Alberto Parravicini got his Ph.D. in Information Technology - Computer Science and Engineering at Politecnico di Milano in 2022, where he also got his Laurea (M.Sc. equivalent) degree in computer engineering. His research covers high-performance computing for graph analytics and numerical analysis, with a focus on scalable GPU computation and making heterogeneous computing more accessible.

Francesco Sgherzi (Graduate Student Member, IEEE) got his M.Sc. degree in Computer Science and Engineering from Politecnico di Milano in 2021, and he is currently a Ph.D. Student in XXX at the Barcelona Supercomputing Center. His research covers XXX.

Marco Arnaboldi (Member IEEE?)

Arnaud Delamare is a senior member of technical staff at Oracle Labs since 2018. His research covers different projects, mainly related to distributed graph processing cloud applications.

Daniele Bonetta (Member IEEE?)

Marco Domenico Santambrogio (Senior Member IEEE) received the Laurea (MSc equivalent) degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2004, the MSc degree in computer science from the University of Illinois at Chicago, Chicago, IL, USA, in 2005, and the PhD degree in computer engineering from the Politecnico di Milano, in 2008. He was a post-doctoral fellow with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA. He has been with the NECST Laboratory, Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design project, in 2004 and the CHANGE (self-adaptive computing system) project, in 2010. He is currently an associate professor with the Politecnico di Milano. His current research interests include reconfigurable computing, self-aware and autonomic systems, hardware/software co-design, embedded systems, and high-performance processors and systems.