PARALiA: A Performance Aware Runtime for Auto-tuning Linear Algebra on Heterogeneous Systems

Citation for the original published paper (version of record):
Anastasiadis, P., Papadopoulou, N., Goumas, G. et al (2023). PARALiA: A Performance Aware Runtime for Auto-tuning Linear Algebra on Heterogeneous Systems. Transactions on Architecture and Code Optimization, 20(4). http://dx.doi.org/10.1145/3624569

PARALiA: A Performance Aware Runtime for Auto-tuning Linear Algebra on Heterogeneous Systems

PETROS ANASTASIADIS, Cslab, National Technical University of Athens, Greece
NIKELA PAPADOPOULOU, Computer Science and Engineering, Chalmers University of Technology, Sweden
GEORGIOS GOUMAS and NECTARIOS KOZIRIS, Cslab, National Technical University of Athens, Greece
DENNIS HOPPE and LI ZHONG, HLRS, University of Stuttgart, Germany

Dense linear algebra operations appear very frequently in high-performance computing (HPC) applications, rendering their performance crucial to achieve optimal scalability. As many modern HPC clusters contain multi-GPU nodes, BLAS operations are frequently offloaded on GPUs, necessitating the use of optimized libraries to ensure good performance. Unfortunately, multi-GPU systems are accompanied by two significant optimization challenges: data transfer bottlenecks as well as problem splitting and scheduling in multiple workers (GPUs) with distinct memories. We demonstrate that the current multi-GPU BLAS methods for tackling these challenges target very specific problem and data characteristics, resulting in serious performance degradation for any slightly deviating workload. Additionally, an even more critical decision is omitted because it cannot be addressed using current scheduler-based approaches: the determination of which devices should be used for a certain routine invocation. To address these issues we propose a model-based approach: using performance estimation to provide problem-specific autotuning during runtime. We integrate this autotuning into an end-to-end BLAS framework named PARALiA. This framework couples autotuning with an optimized task scheduler, leading to near-optimal data distribution and performance-aware resource utilization. We evaluate PARALiA in an HPC testbed with 8 NVIDIA-V100 GPUs, improving the average performance of GEMM by 1.7× and energy efficiency by 2.5× over the state-of-the-art in a large and diverse dataset and demonstrating the adaptability of our performance-aware approach to future heterogeneous systems.

CCS Concepts: • **Computer systems organization** → *Interconnection architectures*; • **Mathematics of computing** → **Mathematical software performance**; • **Software and its engineering** → Application specific development environments; • **Theory of computation** → **Parallel computing models**; • **General and reference** → *Empirical studies*

Additional Key Words and Phrases: Graphics processing units, BLAS optimization, performance prediction

This work has been supported by the project CATALYST, which is funded by the Ministry of Science, Research and Arts (MWK), Baden-Württemberg, Germany.

Authors' addresses: P. Anastasiadis, G. Goumas, and N. Koziris, National Technical University of Athens (NTUA), School of Electrical and Computer Engineering, Heroon Polytexneiou 9, Polytexneioypolis Athens, Zografou, 15773 Greece; e-mails: panastas@cslab.ece.ntua.gr, goumas@cslab.ece.ntua.gr, nkoziris@cslab.ece.ntua.gr; N. Papadopoulou, Department of Computer Science and Engineering, Chalmers University of Technology, Chalmersplatsen 4, Gothenburg, 412 96 Sweden; e-mail: nikela@chalmers.se; D. Hoppe and L. Zhong, High-Performance Computing Center Stuttgart (HLRS), Nobelstr. 19, 70569 Stuttgart, Germany; e-mails: hoppe@hlrs.de, li.zhong@hlrs.de.

1544-3566/2023/12-ART52
https://doi.org/10.1145/3624569

ACM Reference format:
Petros Anastasiadis, Nikela Papadopoulou, Georgios Goumas, Nectarios Koziris, Dennis Hoppe, and Li Zhong. 2023. PARALiA: A Performance Aware Runtime for Auto-tuning Linear Algebra on Heterogeneous Systems . *ACM Trans. Arch. Code Optim.* 20, 4, Article 52 (December 2023), 25 pages.
https://doi.org/10.1145/3624569

1 INTRODUCTION

Dense linear algebra operations occur very frequently in **high-performance computing (HPC)** applications, making their performance critically important for their scalability. The standardization of the **Basic Linear Algebra Subprograms (BLAS)** [1] in the early days of HPC eased the development of scientific code, allowing domain experts to rely on standardized and performance-optimized building blocks to implement more complex simulations at scale. The ample regular parallelism of BLAS routines made them a good fit for GPUs, hence the existence of many BLAS libraries for GPUs, the most common being *cuBLAS*, a CUDA-based BLAS library for NVIDIA GPUs [2] which offered highly-optimized primitive BLAS operations with the constraint that the input data must reside on GPU memory.

The success of GPUs resulted in the widespread adoption of multi-GPU nodes, with custom interconnects between the GPUs. However, even the most GPU-friendly BLAS routines like **general matrix-matrix multiplication (GEMM)** initially struggled to exploit the compute capabilities of these nodes, only achieving small fractions of peak performance. The addition of multiple workers to a single problem decreased data reuse and required additional communication, including **device-to-device (d2d)** transfers over links with various bandwidths, while the domain decomposition and engineering complexity for developing multi-GPU BLAS increased considerably. This led the scientific community to the extension [3–6] or development [7–10] of many BLAS libraries for multi-GPU systems where the input data can reside on host memory, on GPU memory, or a combination of both. Despite supporting all these data configurations, the optimization of these libraries focused only on the *homogeneous* case meaning that: a) all data reside on the CPU memory (henceforth *full offload*) and b) all GPUs were considered equal and always used regardless of their performance contribution.

While this homogeneous case leads to a programmable drop-in replacement of legacy CPU code with GPU-enabled code, it suffers from severe performance penalties and energy inefficiencies in the general case. More specifically, it fails to provide a solid solution in applications that rely on a series or a workflow of BLAS invocations, as is for example the case of iterative solvers or machine learning pipelines. In these cases data that are produced by a BLAS kernel in the GPU(s) may be consumed by subsequent BLAS kernel(s) again on the GPU(s), instead of always being updated on the CPU. Additionally, deploying each single kernel execution on all GPUs is not always efficient, since either the kernel itself may not be scalable, or a deployment of multiple concurrent kernels would lead to a much more scalable or energy-efficient execution.

Figure 1 shows the performance of state-of-the-art multi-GPU libraries, executed with various data placement configurations. BLASX and XKBLAS, the state-of-the-art multi-GPU level 3 libraries, perform well for GEMM in the full-offload cases, but their performance drops significantly in all other data configurations, where a part of the data is stored in GPU memory before execution. This is particularly noticeable for smaller problem sizes where the execution is more communication-bound. Additionally, since both BLASX and XKBLAS use all available hardware (i.e., all eight GPUs in our case) for all problem sizes, they result in low energy efficiency in the cases where they cannot achieve high performance.

In this paper we present A Performance-Aware Runtime for Auto-tuning Linear Algebra, an end-to-end solution for near-optimal performance-aware multi-GPU BLAS by utilizing auto-tuning

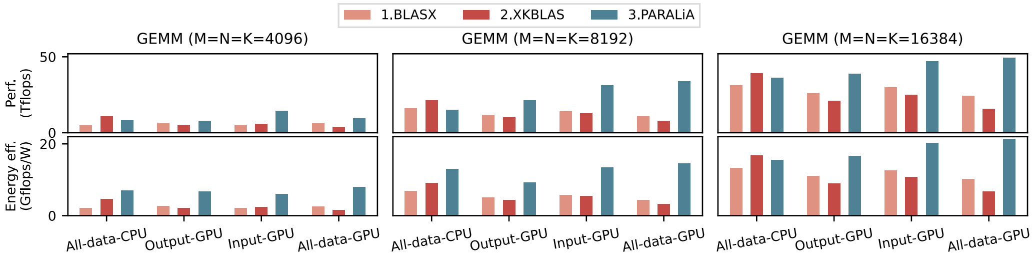

Fig. 1. The GEMM performance (top) and energy efficiency (bottom, using the power-delay product) of the state-of-the-art multi-GPU BLAS libraries BLASX and XKBLAS compared with our work, PARALiA, in a multi-GPU cluster with 8 NVIDIA-V100 GPUs, for three problem sizes and four different data placements. BLASX and XKBLAS offer competitive performance for the first placement but fail to adjust to the other three more complex ones resulting in serious performance degradation, while PARALiA adjusts well to all scenarios and offers increased performance. PARALiA also offers higher energy efficiency through device selection with a negligible trade-off in performance.

and performance modeling during runtime in order to 1) optimize communication and avoid bottlenecks deriving from data placement, and 2) select which devices to use and in what degree for problem-efficient execution. Figure 1 shows that PARALiA provides robust performance regardless of data placement, resulting in superior performance over the state of the art in the three mixed data configurations. Additionally, in the smaller problem sizes where the GPUs are underutilized, PARALiA adapts and uses fewer devices, achieving similar performance coupled with higher energy efficiently. Overall, this paper makes the following contributions:

(1) It introduces a portable multi-GPU communication optimization method, that encodes system characteristics and adjusts communication routing during runtime in order to better fit to different problem layouts (Section 3.2).
(2) It explores performance-aware workload distribution and device selection for multi-GPU BLAS (Section 3.1), using performance modeling with a variety of target metrics (Section 3.3) fueled by empirical micro-benchmarks (Section 3.4).
(3) It combines the above with a runtime tile scheduler into *PARALiA*, an end-to-end multi-GPU BLAS framework offering device selection, coupled with performance-aware runtime task scheduling, which demonstrates an average 1.7× performance and 2.5× energy efficiency improvement over state-of-the-art libraries. (Section 4).

2 BACKGROUND

We define multi-GPU libraries as libraries that allow input data to reside on host memory, GPU memory, or a combination of both and internally manage data distribution and computation on multiple devices. Most existing multi-GPU BLAS libraries target Level-3 BLAS routines [3–10], relieving the programmer from the complex optimizations required on multi-GPU systems, with the optimization of level-1 and level-2 BLAS still left to the programmer due to their usually smaller impact on total application performance. In this section, we present the performance bottlenecks of multi-GPU Level-3 BLAS, as well as the algorithms and optimizations used by the current state-of-the-art libraries to alleviate them. We additionally discuss the limitations of current optimization approaches concerning the initial data distribution and the interconnect heterogeneity. Finally, we discuss the absence of consideration for the resource utilization in existing multi-GPU BLAS libraries, and the relevant efficiency and heterogeneity challenges.

The key architectural features that influence the performance, and thus, library design, in multi-GPU setups are the distinct GPU memories and the increasingly complex underlying

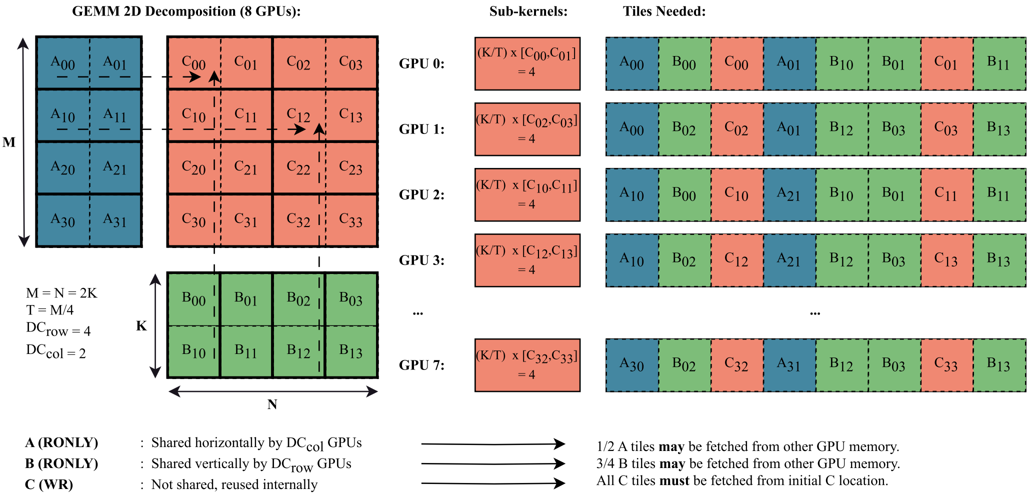
GEMM 2D Decomposition (8 GPUs):
Sub-kernels:
Tiles Needed:
A_{00} A_{01} A_{10} A_{11} A_{20} A_{21} A_{30} A_{31}
M
C_{00} C_{01} C_{02} C_{03} C_{10} C_{11} C_{12} C_{13} C_{20} C_{21} C_{22} C_{23} C_{30} C_{31} C_{32} C_{33}
B_{00} B_{01} B_{02} B_{03} B_{10} B_{11} B_{12} B_{13}
K
N
M = N = 2K
T = M/4
DC_{row} = 4
DC_{col} = 2
GPU 0: (K/T) x [C_{00},C_{01}] = 4 | A_{00} B_{00} C_{00} A_{01} B_{10} B_{01} C_{01} B_{11}
GPU 1: (K/T) x [C_{02},C_{03}] = 4 | A_{00} B_{02} C_{02} A_{01} B_{12} B_{03} C_{03} B_{13}
GPU 2: (K/T) x [C_{10},C_{11}] = 4 | A_{10} B_{00} C_{10} A_{21} B_{10} B_{01} C_{11} B_{11}
GPU 3: (K/T) x [C_{12},C_{13}] = 4 | A_{10} B_{02} C_{12} A_{21} B_{12} B_{03} C_{13} B_{13}
...
GPU 7: (K/T) x [C_{32},C_{33}] = 4 | A_{30} B_{02} C_{32} A_{31} B_{12} B_{03} C_{33} B_{13}
A (RONLY) : Shared horizontally by DC_{col} GPUs → 1/2 A tiles may be fetched from other GPU memory.
B (RONLY) : Shared vertically by DC_{row} GPUs → 3/4 B tiles may be fetched from other GPU memory.
C (WR) : Not shared, reused internally → All C tiles must be fetched from initial C location.

Fig. 2. An example of GEMM (M = N = 2K) 2D decomposition to sub-problems and data tiles (tiling size T = M/2). The eight participating devices are distributed in a 2D grid of (DC_{row}, DC_{col}) = (4, 2) to encourage horizontal and vertical device-to-device (d2d) data movement between same row/column devices, respectively. An optimized library employing software-implemented caching of RONLY tiles to GPUs can avoid 50% and 75% of h2d transfers for the A and B matrices, respectively, by using peer-to-peer d2d transfers.

interconnect. Consequently, unlike single GPU setups, where algorithmic optimizations mainly target the internals of the BLAS kernels, multi-GPU setups include multiple devices acting as parallel workers, introducing the notions of data decomposition, reuse, communication, scheduling, and load balancing. For simplicity, we categorize the optimization space for multi-GPU BLAS into a) data domain decomposition, i.e., *splitting* the initial problem into sub-problems/tasks (henceforth *sub-kernels*) and their distribution, b) communication *overlap*, *avoidance*, and *routing* and c) *load-balancing* between GPUs.

2.1 Level-3 BLAS Decomposition and Distribution

Level-3 BLAS routines operate on matrices, and typically the problem is decomposed into data-parallel tasks, following some matrix decomposition scheme. On multi-GPU setups, this is necessary to exploit the multiple devices as parallel workers, similarly to multi-core or multi-node execution [11]. Decomposition and distribution schemes are important, as they determine the required amount of communication between workers. Early multi-GPU BLAS libraries, such as the CUDA-based *cuBLASXt* [7], and its LAPACK-compatible wrapper *NVBLAS* [8], use a simple round-robin scheme to distribute 2D tile-based sub-kernels to devices, resulting in unnecessary communication. State-of-the-art libraries such as BLASX [9] and XKBLAS [10] borrow the 2D block-cyclic decomposition and distribution from distributed computing, which achieves a good homogeneous distribution of communication on a virtual 2D-grid of workers.

In this work, we also employ the state-of-practice 2D block-cyclic decomposition/distribution. Figure 2 shows an example of the 2D block-cyclic distribution used for Level-3 BLAS matrix-matrix multiplication (GEMM). The available system devices are organized in a virtual 2D grid, which should be as square as possible - for example, a 2 × 2 grid for four devices, a 4 × 2 or 2 × 4 grid for eight devices, a 3 × 3 grid for nine devices, and so on.

2.2 Communication Optimization

The 2D block-cyclic decomposition in GEMM depicted in Figure 2 results in a favorable communication pattern for the read-only tiles of matrices A and B. Every GPU requires the same number of “row” tiles and “column” tiles from each of the input matrices. This is the basis for enabling *communication optimization*, as communication is the main bottleneck in multi-GPU BLAS performance on modern systems [9, 10, 12]. While different libraries have used different approaches for improving communication performance, the optimization targets can be roughly classified into communication *overlap*, *avoidance*, and *routing*.

Overlap: Overlap refers both to computation-communication overlap, as well as communication-communication overlap, when this happens between different devices. Computation-communication overlap is a common technique in GPU offload, both in single and multi-GPU setups and it has been extensively explored in the past [12–15]. CUDA versions also frequently increase the overlap potential by enabling additional GPU copy-engines [16]. In this work, we also try to maximize overlap to improve the performance of Level-3 BLAS.

GPU Data caching: In a multi-GPU setup, the different GPUs have distinct memories. Because of the problem distribution, data that is necessary for computations need to be transferred from one device to another often. Although the effect of those transfers is mitigated by technologies like RMA, a common approach to reduce redundant communication is data caching/buffering on the GPU memory, as it also enables data reuse between subsequent subkernels. For example, in Figure 2, if GPU 0 does not cache data tiles, it needs to fetch C_{00}, A_{00}, A_{01} and C_{01} twice, resulting in 12 tile fetches instead of 8 (50% increase in communication volume), which can worsen depending on the problem size and the tiling size T.

BLAS libraries initially designed for CPUs [3–6] use simple buffers in GPU memories to support some data caching, and cuBLASXt [7] follows the same design logic. Unfortunately, simple buffering is only sufficient for internal GPU caching, which fares well on a single GPU. Its performance degrades rapidly as the number of GPUs increases, since it neglects the very fast connections between discrete GPU memories modern interconnects offer [9]. In this work, we employ a GPU data caching scheme to reduce unnecessary communication and maximize data reuse.

Communication routing: Routing in a multi-GPU setup refers to selecting the fastest routes for transfers between GPUs. In the context of Level-3 BLAS, the multiple GPUs require different data tiles, which need to be transferred, and peer-to-peer transfers between GPUs (henceforth *d2d*) usually have considerably higher bandwidth than CPU-GPU transfers (henceforth *h2d/d2h*). To enable routing optimizations for multi-GPU BLAS, two components are necessary: 1) a cache consistency-like logic for the GPU buffers, to ensure that data tiles are always up to date and their Read/Write dependencies are respected, and 2) a way to distinguish the interconnect bandwidth levels in order to select the ‘closest’ fetch location when a **read-only (RONLY)** tile is available in multiple buffers. A solution like this is implemented in BLASX [9], which provides a hardware abstraction of the underlying interconnect in a hierarchical representation, designed for communication optimization in multi-GPU systems. The hierarchical representation abstracts the system as a cache hierarchy, with the CPU RAM being the main memory and the distinct GPU memories being levels of caches. This representation favors data reuse (last-level cache ‘hits’) and the faster GPU-GPU transfers (lower-level cache ‘hits’) to CPU-GPU transfers (cache ‘misses’), while a MESI-like protocol enables sharing RONLY blocks between GPUs. Still, BLASX performance is hindered by continuously writing back to the CPU and re-fetching output (WR) blocks. Moreover, recent multi-GPU clusters from NVIDIA are connected with NVLink lanes, which may offer more than one distinguishable bandwidth ‘levels’, compared to the simple distinction between ‘h2d’ and ‘d2d’ bandwidths. XKBLAS [10] overcomes the write-back bottleneck by performing lazy write-backs of output data. It additionally considers a heuristic ‘ranking’ of distinguishable bandwidth levels

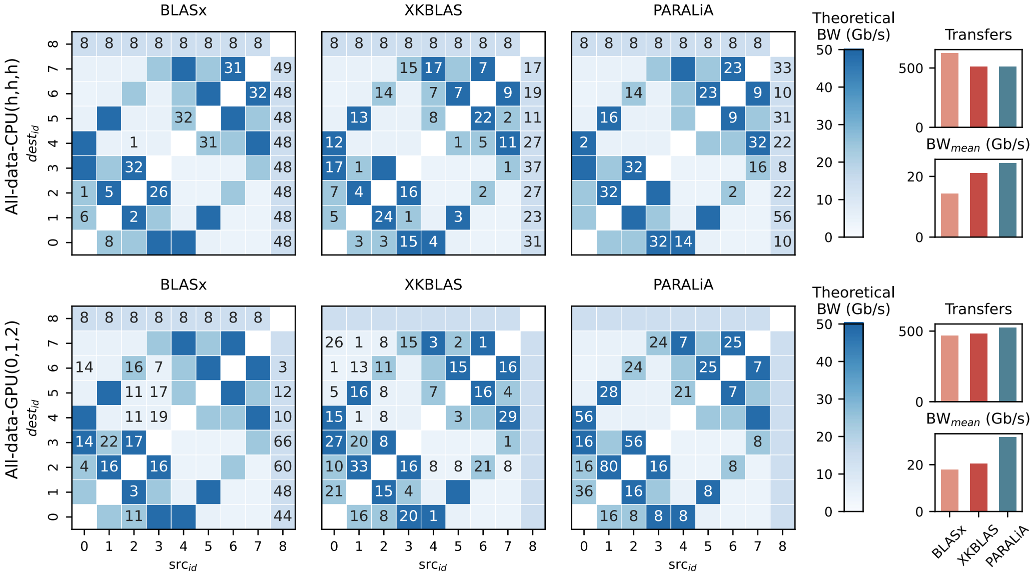

Fig. 3. The communication pattern of BLASX, XKBLAS and PARALiA for a GEMM execution (M = N = K = 16384, T = 2048) in a testbed with 8 NVIDIA V100 GPUs interconnected by a mixed grid of NVLink1 and NVLink2 (more in Section 4), for two data placements: the full-offload case (all data at host memory initially) and a case where the A, B and C matrices are initially populating the memories of GPUs 0, 1 and 2, respectively. The heatmaps visualize all communication (source GPU = x axis, destination GPU = y axis); the heat is the theoretical bandwidth of each connection and the displayed labels in each box denote the total number of (equal byte) transfers passing from this connection during execution. The *id* = 8 is assigned to the host memory. The bar plots aggregate the total transfers and their average bandwidth for each library. PARALiA achieves a much higher average bandwidth for both cases by utilizing the 'hottest' links with the highest bandwidths.

through information from the NVIDIA driver interface, and favoring transfers over device connections higher bandwidth. In our work, we also perform communication routing as a communication optimization, through a different hardware abstraction and caching scheme.

The data placement hazard: While all the aforementioned optimizations target the full-offload scenario, where all data are initially located on the CPU, BLAS multi-GPU libraries support input data on any GPU. On modern systems, CPU-GPU (h2d/d2h) transfers are inherently much slower than peer-to-peer, GPU-GPU (d2d) transfers, therefore having part of the input data available on GPUs should lead to a less communication-bound problem.

However, as shown in Figure 1, the performance of BLASX and XKBLAS drops significantly in all but the full-offload scenarios and only PARALiA provides the expected increased performance. We demystify the cause for this counter-intuitive behavior in Figure 3, which shows the communication pattern, number of transfers and average achieved bandwidth for BLASX, XKBLAS and our proposed runtime, PARALiA. BLASX suffers from excessive costly write-backs to the CPU due to its caching policy, although these h2d/d2h could be completely avoided in the scenario where all data are initially on the GPUs [10]. Additionally, BLASX is bandwidth-agnostic, with transfers passing through a variety of bandwidth levels, since its hierarchical abstraction is only capable of recognizing the difference between *h2d/d2h* and *d2d*, and is not sufficient for modern interconnects. XKBLAS, on the other hand, avoids intermediate write-backs to the CPU due to its lazy write-back design and provides a much more balanced communication map in the full-offload scenario, with a clear preference for the higher bandwidths. This desirable behavior does not extend to the scenario where data initially populate GPUs 0, 1 and 2. On the contrary, the communication map becomes more dense around these locations and creates a communication bottleneck, with a lot of extremely low-bandwidth transfers. The cause of this meltdown lies in the core of current multi-GPU Level-3 BLAS optimization - the decomposition and distribution method. All usual distribution methods (round-robin, block-cyclic, 2D block-cyclic, etc.) target a homogeneous, balanced scenario, where data - and consequently the communication volume - is distributed between workers as equally as possible. In the full-offload scenario, all data must first be fetched from the host ($id = 8$ in the heatmaps), which in our testbed - and most modern HPC clusters - has the same bandwidth for all h2d/d2h connections and therefore favors a balanced distribution. On the other hand, in the second scenario, some GPUs are *closer* (higher d2d bandwidth) and some are *further* (lower d2d bandwidth) from the data, which results in transfers passing through a variety of connections, some of which are very slow. In our work, PARALiA, we take data placement into consideration and optimize communication, under the assumption of heterogeneity in the underlying interconnect, through an effective abstraction of the hardware, modeling, and autotuning. As shown in Figure 3, our work achieves higher average bandwidth compared to other techniques regardless of the initial data placement.

2.3 Load Balancing for Heterogeneity

State-of-the-art multi-GPU libraries attempt to mitigate the aforementioned problem of imbalance between transfers, and also adapt to potential heterogeneous computational capabilities, by using task load-balancing through work-stealing [9, 10]. However, although work-stealing can improve load balancing, it disrupts the potential temporal locality offered by a good initial task decomposition and distribution, as is the 2D block-cyclic decomposition. Moreover, although work stealing may address the load imbalance coming from small performance differences between GPUs, it is not sufficient for heterogeneous workload distribution, as we will show in Section 4.4. Finally, the work stealing mechanisms integrated in existing approaches are performance-agnostic; they balance work between devices, without taking into account the computational capabilities of the devices or the communication properties of the problem. This design results in inefficient resource allocation and usage for BLAS execution, and is incompatible with future heterogeneous systems. This monolithic design does not lead to efficient execution since it can't adapt resource allocation to BLAS problem characteristics and is incompatible with future heterogeneous systems.

In this work, we argue that a homogeneous distribution coupled with work-stealing is not able to effectively handle the built-in heterogeneity of modern HPC systems for the case of BLAS, and propose a model-based approach that adjusts the decomposition, communication, and task allocation to the characteristics of 1) each different system through offline micro-benchmarks and 2) each different BLAS problem and data placement during runtime. Our approach steps away from the typical homogeneous distribution coupled with work stealing, as we believe that this approach cannot simultaneously account for the interconnect layout, problem size, and data placement, and therefore cannot effectively handle the built-in heterogeneity of modern multi-GPU HPC systems for the case of BLAS.

3 THE PARALIA FRAMEWORK

In this section, we present the main contribution of this work; the PARALiA framework. First, we demonstrate the high-level design, briefly describing the components involved in BLAS optimization and exposing the decision knobs that can be autotuned.

Figure 4 shows the complete PARALiA framework that supports the efficient execution of BLAS routines in a heterogeneous multi-GPU system. PARALiA is activated when user code

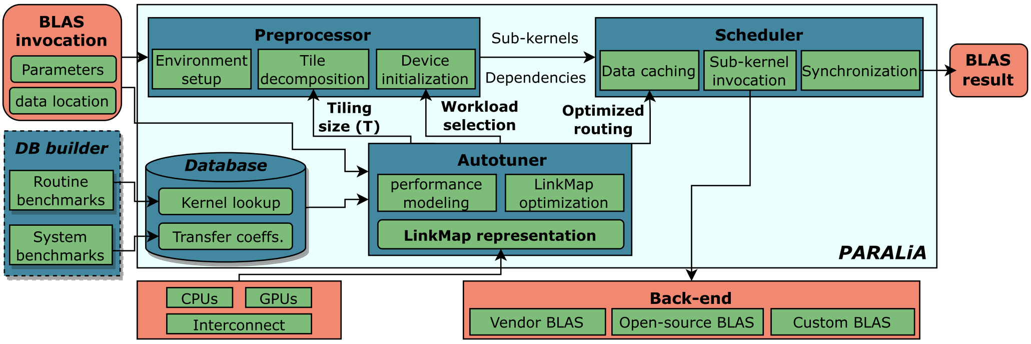

Fig. 4. An overview of the PARALiA framework and its main components.

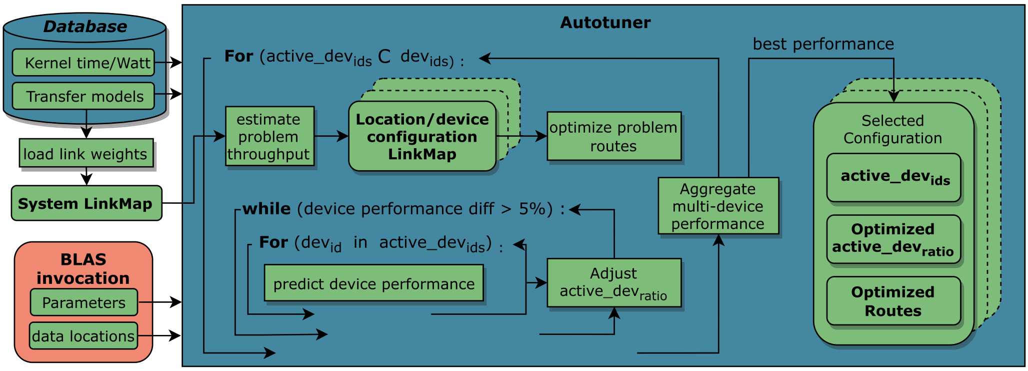
Database
Kernel time/Watt
Transfer models
load link weights
System LinkMap
BLAS invocation
Parameters
data locations
Autotuner
For (active_dev_ids ⊂ dev_ids) :
best performance
estimate problem throughput
Location/device configuration LinkMap
optimize problem routes
Selected Configuration
active_dev_ids
Optimized active_dev_ratio
Optimized Routes
Aggregate multi-device performance
while (device performance diff > 5%) :
For (dev_id in active_dev_ids) :
predict device performance
Adjust active_dev_ratio

Fig. 5. An overview of the PARALiA autotuner and its prediction pipeline.

invokes a BLAS routine with routine data residing within the memory of any of the available devices. The framework consists of three main components: a preprocessor (Section 3.5) that is responsible for preparing the framework environment for execution, a scheduler (Section 3.6) that is responsible for managing input/output data and invoking backend BLAS kernels, and an autotuner (Section 3.1) that receives system and problem parameters from a database (Section 3.4), a hardware abstraction (LinkMap, Section 3.2) and the routine invocation, and decides which devices to utilize for BLAS execution, the granularity (tiling size) of the basic computational blocks and the data transfer routing. The PARALiA framework is a publicly available open-source project.

3.1 The Autotuner Algorithm

The autotuner is the backbone of PARALiA's optimization. Its purpose is to improve (1) communication throughput and (2) workload distribution for arbitrary system/problem configurations. Due to the more generic nature of this problem, using a heuristic-based approach is bound to favor a subset of configurations, based on which the heuristics were designed, that being either specific system characteristics (e.g., number of CPUs/GPUs, inter-connectivity) or problem characteristics (e.g., data size, placement). For this reason, the autotuner uses a model-based approach instead, which looks at each configuration as a different problem, by combining its *system* and *problem* characteristics at runtime.

The autotuning algorithm that commences during each routine invocation is shown in detail in Figure 5. When a routine is invoked, the problem parameters are extracted from the routine.

The autotuner loads pre-obtained transfer coefficients from the PARALiA database and uses them to construct an abstraction of the system characteristics called LinkMap. Then, the autotuner loops over **candidate workload distributions**, estimates their total performance and selects the best one. Each *workload distribution* consists of a) a list of dev_{num} devices ($active\_dev_{ids}$), which is a subset of the total system devices, b) a list of sub-kernel ratios ($active\_dev_{ratio}$) suggested for each device, and c) a transfer routing map optimized for this specific distribution. Regarding (a) and (b), since their combined search space is very large ($active\_dev_{ratio}$ are float values), we instead decouple them by iterating on the possible device combinations $active\_dev_{ids}$ (which are discrete) and selecting their $active\_dev_{ratio}$ with a model-based method. Specifically, for each device combination we start with equal sub-kernel ratios, and iteratively adjust the ratios based on a performance prediction for each device (more in Section 3.3), until a $active\_dev_{ratio}$ with similar performance per device (within 5%) is reached. Regarding (c), the autotuner adjusts and optimizes the *LinkMap* to each aforementioned scenario using its specific problem characteristics (more in Section 3.2). Finally, the best *workload distribution* is selected by using some metric-related aggregator (e.g., maximum for time, sum for energy, etc.) on the performance of each device obtained during the estimation of (b). We note that the autotuner also selects a tiling size **T** for tile decomposition (as depicted in Figure 4), but this process is disconnected from (a), (b) and (c) and performed based on CoCoPeLiA [12] due to its small impact in multi-GPU performance.

3.2 The LinkMap Representation

Since the hardware abstractions of previous libraries target homogeneous distributions in systems with similar device and interconnect capabilities, they are not suitable for any workload distribution. To mitigate this we assume the most generic system in an abstraction called LinkMap, capable of representing any system with arbitrary devices and connections between them. The LinkMap abstraction disconnects from the notion of "CPU" and "Main memory" and treats all parts of a system similarly; any candidate data location or available computational resource is categorized as a ***device*** and is connected via ***links*** with all other devices, which are responsible for data transfers between them. In the LinkMap representation each *device* is defined by a unique id (dev_{id}). While not common in current systems, different devices can share memory, in which case the transfer link time between them is always equal to zero. Additionally, this abstraction assumes a fully-connected virtual topology; even if an actual hardware connection does not exist between a device pair. Therefore, this creates a fully-connected graph, with where devices are the nodes and the links are the edges: the dev_{num} nodes are connected via a 2D grid (dev_{num}, dev_{num}) of edges/links. The LinkMap representation is implemented in C++ as a class whose members and functions are shown in Table 1. It consists of five 2D matrices $link_{\{lat,bw,bw-shared,route,sl\}}$ that hold its values and three functions that are used during auto-tuning to update them.

The LinkMap representation by itself does not contain any insights, it just represents the most general case. Its usefulness is its adaptability to any system and problem data placement, which happens during runtime and has three basic phases, implemented in the LinkMap functions. First, once per program during the first routine invocation, $load\_link\_weights()$ loads the transfer coefficients $link_{\{lat,bw,sl\}}$ from the database. This provides a basic *System LinkMap* containing empirically obtained estimations for the system in general. Then during the autotuning of any routine, $estimate\_problem\_throughput()$ adjusts the LinkMap bandwidths ($link_{bw-shared}$) according to the current device/data configuration. Specifically, it assumes that all links that connect the dev_{num} devices ($active\_dev_{ids}$) to the $data_{num}$ data locations ($data_{locs}$) perform transfers for the entire routine execution, and apply the slowdown of simultaneous usage (Equation (6)) to the bandwidth

Table 1. `LinkMap` Member and Functions used for Communication Optimization

System-wise:	
$link_{lat}(dest_{id}, src_{id})$	The latency of each link.
$link_{bw}(dest_{id}, src_{id})$	The isolated bandwidth of each link.
$link_{sl}(dest_{id}, src_{id}, s\_dest_{id}, s\_src_{id}))$	The slowdown imposed by simultaneous usage on each pair of links.
Problem-adjusted:	
$link_{bw-shared}(dest_{id}, src_{id})$	The sustainable bandwidth of each link for a device/data configuration.
$link_{route}(dest_{id}, src_{id})$	The underlying route all transfers passing through a link must follow.
Functions:	
$load\_link\_weights()$	Initializes $link_{bw/lat/sl}$ from the database.
$estimate\_problem\_throughput()$	Estimates $link_{\{bw-shared\}}$ for a device/data configuration.
$optimize\_problem\_routes()$	Re-routes communication for 'bad' links for a for a device/data configuration.

of each such link:

$$link_{bw-shared}(dest_{id}, src_{id}) = link_{bw}(dest_{id}, src_{id}) \times \sum_{i=0}^{dev_{num}} \sum_{j=0}^{data_{num}} link_{sl}(dest_{id}, src_{id}, active\_dev_{ids}(i), data_{locs}(j)))$$

The final optimization phase is to calculate something similar to the shortest paths for this graph. In our case, we want to reroute transfers that would pass through links with low bandwidth to *series of links* of higher bandwidth. For example using three devices, if $link_{bw-shared}(0 \rightarrow 1) = 1Gb/s$, $link_{bw-shared}(0 \rightarrow 2) = 3Gb/s$ and $link_{bw-shared}(2 \rightarrow 1) = 4Gb/s$, the shortest transfer route for $0 \rightarrow 1$ would be optimized to $0 \rightarrow 2 \rightarrow 1$, since it would be faster to transfer data from device 0 to device 1 through device 2, instead of using their direct link. To avoid very long routes the re-routing algorithm ($optimize\_problem\_routes()$) uses a max_hops argument that limits the intermediate data locations (currently supported $max\_hops = \{1, 2\}$), and we use $max\_hops = 1$ in our evaluation. Performing these intermediate 'hops' during runtime has a very low overhead since PARALiA already holds tile buffers in all devices. Re-routing significantly improves performance since 1) bandwidth is increased for the otherwise slowest transfers, which are an important bottleneck, and 2) in level-3 BLAS, transferring a read-only data chunk with additional 'hops' (like device 2 in the example) also stores it to these devices for potential use.

3.3 Offload Performance Estimation

As explained in Section 3.1, the ratio adjustment and the total performance aggregation in the autotuner use an estimation of the offload performance of each device (henceforth $pred_{metric}(dev_{id})$). Performance prediction in multi-GPU setups is considerably more complex than on a single GPU, as scheduling on multiple devices involves runtime decisions regarding data caching and simultaneous resource utilization that are not static or known beforehand. For this reason, we use a performance upper bound based on the *full-overlap* model [15], instead of using more advanced overlap models [12, 14, 15]. We note that, for simplicity, all equations presented below use time as the performance *metric*, but PARALiA supports more performance metrics that are later explained in detail. Table 2 summarizes the modeling notation used in this work.

First, we combine the *full-overlap* upper bound [15] with the PARALiA database to get a routine-specific, full-overlap prediction for *each* device's total performance:

Table 2. Modeling Notation used in this Work

Empirical values (from database):	
$t_{exec}(routine, dev_{id}, D1[, D2[, D3]])$	The execution time of *routine* in dev_{id} as a function of problem size.
$W_{exec}(routine, dev_{id}, D1[, D2[, D3]])$	The average power (in Watt) of *routine* in dev_{id} during execution.
Problem parameters (from routine):	
$dims : D1[, D2[, D3]]$	Problem dimensions for BLAS level-1, 2 and 3, respectively.
$data_{num}$	The number of total matrices and vectors used by this routine.
$is_{\{R,W\}}(data_{num})$	A flag [0,1] denoting if a matrix/vector is input/output, respectively.
$data_{loc}(data_{num})$	The data placement of each participating matrix/vector.
$bytes(data_{num})$	The size in bytes of all matrices and vectors used by this routine.
Estimated (model-based):	
dev_{num}	The number of devices participating in multi-device parallel execution.
$active\_dev_{ids}(dev_{num})$	A list containing the ids for each such device.
$active\_dev_{ratio}(dev_{num})$	The percentage of the total sub-kernels assigned to each such device.
$pred_{metric}(dev_{id})$	A *metric* prediction required for dev_{id} to complete its assigned sub-kernels.
$total\_pred\_metric$	The total estimated *metric* (e.g., time, EDP) of multi-device parallel execution.

$$pred\_t_{base}(dev_{id}) = max\left(t_{exec}(dev_{id}, dims), t_{h2d}\left(dev_{id}, \sum_{i}^{is_R} bytes(i)\right), t_{d2h}\left(dev_{id}, \sum_{j}^{is_W} bytes(j)\right)\right) \tag{1}$$

where **h2d** stands for **host-to-device** and **d2h** for **device-to-host** transfers, and $\sum_{\{i,j\}}^{is_{\{R,W\}}}$ are the subsets of the $data_{num}$ matrices/vectors that are problem inputs and outputs, respectively. To adjust the model for multi-device offload, we need to replace 'h2d' and 'd2h' time with the transfer times of all links connecting $data_{locs}$ to each device. To do this, first, we calculate the transfer time for each link (t_{link}) as a function of transferred *bytes* with:

$$t_{link}(dest_{id}, src_{id}, bytes) = link_{lat}(dev_{id}, src_{id}) + \frac{bytes}{link_{bw-shared}(dest_{id}, src_{id})} \tag{2}$$

by combining each link's latency and bandwidth using the well-accepted latency/bandwidth model [12, 14, 15, 17–19]. Then, we assume the best-case scenario, where all input matrices/vectors are distributed equally between the dev_{num} devices by combining Equation (1) with Equation (2) to generalize for any initial data placement:

$$\begin{aligned} pred\_t_{over}(\ldots) = &max\left(t_{exec}(\ldots), \sum_{i}^{is_R} t_{link}\left(dev_{id}, data_{locs}(i), \frac{bytes(i)}{dev_{num}}\right),\right. \\ &\left.\sum_{j=0}^{is_W} t_{link}\left(data_{locs}(j), dev_{id}, \frac{bytes(j)}{dev_{num}}\right)\right) \end{aligned} \tag{3}$$

Equation (3) provides a more accurate prediction for the full-overlap performance of a routine, *if multi-GPU execution does not involve additional transfers/data sharing between devices*. This assumption works for level-1 and level-2 BLAS, but in level-3 BLAS decomposition each tile is reused by many sub-kernels and therefore transferred to multiple devices throughout a routine's lifetime.

Since PARALiA uses a 2D block-cyclic decomposition (DC_{row}, DC_{col}) for level-3 BLAS, we consider this baseline scenario of 1) exchanging equal portions of RONLY bytes between all decomposition rows and columns and 2) no output data sharing. We estimate the proportional increase in transfer volume for each device as:

$$extra\_transfer\_bytes = \frac{(DC_{row} - 1) + (DC_{col} - 1)}{RONLY_{num}} \cdot RONLY\_sum\_bytes$$

Where $RONLY_{num}$ is the number of matrix/vectors with $is_R = 1$ and $is_W = 0$, and $RONLY\_sum\_bytes$ is the sum of their corresponding *bytes*. This represents a lower bound of the added bytes due to multi-GPU BLAS3 data sharing *for each device*. We assume these bytes are equally distributed between devices, and use the *average bandwidth* of all links to estimate the additional transfer time:

$$t_{extra}(dev_{id}) = extra\_transfer\_bytes \cdot \frac{dev_{num}}{\sum_{idx=0}^{dev_{num}} link_{bw-shared}[dev_{id}][idx]} \quad (4)$$

in which the extra communication in bytes for each device is multiplied by the inverse of its average receive bandwidth, which serves as an average estimate for the expected bandwidth of these transfers. We finally construct the full-overlap model used for the *estimated performance of each GPU in a multi-GPU environment* by adding the extra transfer time of Equation (4) to Equation (3):

$$pred_t(dev_{id}, \ldots) = max\left(t_{exec}(\ldots), t_{extra}(dev_{id}) + \sum_{i}^{is_R} t_{link}(\ldots), \sum_{j}^{is_W} t_{link}(\ldots) \right) \quad (5)$$

Performance metrics*:* All the aforementioned models return a time prediction for the execution on a single GPU. We use the maximum predicted execution time, $total\_pred\_t = max(pred_t(dev_{id}))$ to evaluate different candidate workload distributions. PARALiA also supports utilization/energy-centric metrics, based on the total power consumption of all GPUs in the multi-GPU setup, $total\_pred\_W$, which we combine with $total\_pred\_t$ to further enhance the workload selection process. In this work, we use 1) performance ($FLOPs = \frac{FLOP}{time}$), 2) an inverse energy-delay product ($EDP_i = FLOPs^2/W$) and 3) an inverse power-delay product ($PDP_i = FLOPs/W$) for workload distribution (*PARALiA select(*$\{FLOPs, EDP_i, PDP_i\}$, respectively). As evaluation metrics we use performance ($FLOPs$, in $GFLOPs$ or $TFLOPs$) and energy efficiency (PDP_i, in $GFLOPs/W$). To support rapid experimentation with additional metrics, we have simplified the addition and benchmarking of new metrics, requiring the modification of 3-4 lines of code only.

3.4 Database

The PARALiA database stores the empirical measurements required to construct PARALiA's system abstraction `LinkMap`, and to estimate performance in the autotuner. These measurements include transfer latencies and bandwidths, which are collected only once per system, and execution time/Watt measurements, collected for each routine. The database is automatically built by PARALiA at installation time, with a set of automated micro-benchmarks (we denote this process as the *DB Builder*), for all available devices and all connections between them.

Database Builder*:* The *DB builder* is an extension of the relevant component in CoCoPeLia [12], which performs single-device BLAS routine benchmarks for all system devices, and extends the set of system benchmarks to model transfers according to the requirements of the `LinkMap` representation. In PARALiA's DB builder, we opt for ease of use, robustness and short benchmarking time. For ease of use, PARALiA provides a fully automated process for micro-benchmarking and for producing the empirical transfer models. Additionally, PARALiA is easily extensible to accommodate new backend BLAS library options, providing micro-benchmark template scripts, which can be easily modified with new routine invocations and any additional parameters. For robustness, for each benchmarked value, we repeat measurements until the 95% confidence interval of the mean falls within 5% of the reported mean value (taking at least 10 measurements to obtain these means). Finally, for short benchmarking times, we try to strike a balance between the number of measurements required for robustness, and their overall execution time. The DB builder benchmarks the computation time of the different backend BLAS routines. The minimum problem dimensions and steps are static and predefined for all benchmarks, while the maximum dimensions are calculated based on the available memory of the target device. The results of the DB builder are stored as a database and made available to the framework for all subsequent calls in the system.

Kernel lookup: The offload performance models used in the autotuner require an estimate for the routine *execution time* and *average Watts* per device. Using the same technique as in CoCoPeLia [12], we only collect measurements for the time/power of fine-grained chunks of specific, small tiling sizes, namely $\{t, W\}_{exec}(routine, dev_{id}, T[, T[, T]])$, for which we then use value lookup in the database. The average GPU Watt consumption is obtained by sampling Watt values at regular intervals with the CUDA `nvml-driver` during each routine benchmark and averaging these. Micro-benchmarks are performed per routine and per device ($dev_{num} \times routine_{num}$ times) and use separate BLAS backends depending on the target dev_{id}. Devices with $0 \leq dev_{id} < cuda\_dev\_num$ are reserved for available CUDA devices and devices with $dev\_id >= cuda\_dev\_num$ are reserved for available CPUs (usually one). The value lookup micro-benchmarks use cuBLAS for NVIDIA GPUs and OpenBLAS for CPUs, but are easily extensible with minimal adjustments to other dev_{id} ranges (e.g., for AMD devices) or for different BLAS implementations (e.g., a custom GPU implementation instead of cuBLAS) that follow the BLAS standard.

Transfer coefficients: To obtain $link_{\{lat,bw\}}$, we follow the most widely used semi-empirical approach; we measure a set of transfer times and use them to fit the coefficients of basic linear models for transfer time. We obtain $link_{lat}$ empirically as the average latency of multiple single-byte transfers. For $link_{bw}$, we run benchmarks for square transfers with $dtype = double$, for $D1 = D2 = 256 \xrightarrow{step_{ad}=256} \sqrt{max\_device\_memory/2}$, and use least square regressions on the obtained samples in the manner of [20]. Then, we estimate the slowdown $link_{sl}$ for simultaneous link usage (e.g., transfer overlap for any two links), assuming it imposes a constant throughput slowdown and does not affect latency. This slowdown is calculated with a single micro-benchmark for each link; first, for the link of interest, a large transfer ($D1 = D2 = \sqrt{max\_device\_memory/2}$) is tested isolated ($t_{link1}$), and then, it is tested overlapped ($t_{link1-link2_{over}}$) with multiple similar transfers on the other link, resulting in the slowdown:

$$sl_{link1-link2} = (t_{link1-link2_{over}})/t_{link1} \tag{6}$$

Since the method of obtaining the sl is empirical, we assume a maximum slowdown of $sl_{link1-link2} = 2.0$ (i.e., the effective bandwidth is halved), to avoid empirical errors spilling into the models. For all transfer experiments we use the PARALiA wrapped functions for transfers, which currently use the `cudaMemcpy2DAsync` routine in their back-end with pinned host memory, as required by these asynchronous calls.

3.5 Preprocessor

The PARALiA preprocessor is responsible for the framework initialization and the transformation of problem data for BLAS execution in a multi-GPU system, which is broken down into the three basic operations described next.

Environment setup: This operation is performed when a BLAS routine is invoked for the first time or with a new set of parameters. It allocates buffers, initializes data structures, and performs backend-specific actions, like creating CUDA streams and events, to be used by the scheduler.
Tile decomposition: The bulk of preprocessing in BLAS libraries involves decomposing the problem data into smaller chunks, usually referred to as *tiles*. As most similar multi-GPU libraries [6, 7, 9, 10], in PARALiA we decompose vectors to 1D tiles and matrices to 2D square tiles, using a tiling size T provided by the autotuner. After data decomposition into tiles, PARALiA identifies all sub-kernels deriving from this decomposition, generating the relevant data/task dependencies.
Device initialization: This operation initializes the devices that will participate in BLAS execution ($active\_dev_{ids}$) and distributes sub-kernels to them, proportionally to their assigned problem ratios ($active\_dev_{ratio}$). PARALiA supports multiple sub-kernel distribution patterns (Sequential, Round-Robin, 1D-cyclic and 2D-cyclic) and by default uses Round-Robin for BLAS 1 & 2 and 2D-cyclic for BLAS3. Unlike scheduler-centric multi-GPU libraries [9, 10] that rely on dynamic load-balancing, PARALiA follows a static approach since load-balancing is based on effective performance estimation performed before scheduling.

3.6 Scheduler

The PARALiA scheduler manages data caching in distinct device memories, and data transfers between memories, invokes all sub-kernels on their assigned devices, and synchronizes their execution and results, as analyzed below.
Data caching: For this operation, PARALiA uses a `Software_buffer` C++ class, similar to BLASX and XKBLAS, which represents a buffer in each device with a distinct memory, and can store 1D and 2D tiles. Each `Software_buffer` holds a number of blocks depending on the problem size and per-device memory limitations, and employs a simple block-sequential write-back policy to swap tiles during sub-kernel execution, using a MESI-like protocol similar to BLASX [9]. This `Software_buffer` in each device is initialized the first time a program calls a PARALiA BLAS routine, and is updated/extended in subsequent calls to match their device and size requirements.
Sub-kernel invocation: The scheduler spawns $active\_dev_{num}$ `POSIX threads`, that are responsible for invoking sub-kernels in their corresponding devices.

After a sub-kernel is invoked, it issues three *sub-tasks*: a) the requests for all its input tile dependencies (e.g., fetching tiles if they are not available in its device memory), b) the sub-kernel computations to be performed after dependencies are met, and c) potential data write-backs to the initial memory location for any tile it modified. The optimization of sub-kernel invocation order is important for multi-GPU BLAS scheduling, since it affects both task parallelism and the communication and data reuse pattern [9, 10]. We leave the sub-kernel order problem for future work because PARALiA focuses on model-assisted communication and workload distribution, not dynamic scheduling techniques.
Synchronization: The sub-tasks of each sub-kernel (i.e., fetch data, compute, writeback) are executed asynchronously and overlapped with sub-tasks from other sub-kernels (software pipelining) and on other devices (multi-GPU) using CUDA events to enforce data dependencies and CUDA streams to enable overlap. After all sub-kernels are invoked, the scheduler synchronizes all sub-tasks and returns the result and control to the user upon completion.

4 EVALUATION

In this section, we evaluate the performance of PARALiA and compare it with state-of-the-art libraries. First, we introduce the testbed and the evaluation dataset we use for our experiments and illustrate its corresponding *LinkMap* representation. Then, we provide a full evaluation of PARALiA's DGEMM performance and compare it against cuBLASXt [7], BLASX [9] and XKBLAS

Table 3. The Vulcan **CLX-AI** Testbed System Characteristics

Vulcan CLX-AI	CPU	GPU
Computation:	4 X Intel Xeon Gold 6240 CPU 18 cores @ 2.60GHz	8 X NVIDIA Tesla V100 FP peak 14 TFlop/s DP peak 7 TFlop/s
Memory:	768GB DDR4	32 GB HBM2 760 GB/s
Interconnect:	PCIe Gen3 ×16	NVlink 1.0/2.0
OS:	Rocky Linux release 8.7	CUDA Driver -
Kernel:	4.18.0-425.3.1.el8.x86_64	510.108.03
Compiler:	g++ 11.2.0	CUDA 11.6
Opt. flags:	-O3	-O3, -arch=sm_75

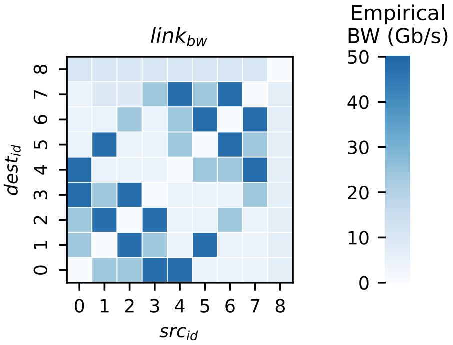

Fig. 6. The *System Linkmap* for the **CLX-AI** interconnect.

[10], using both performance (Tflops) and energy efficiency (Gflops/W) metrics. We compare three versions of PARALiA, with each version using a different approach for workload distribution, based on the estimated routine performance, inverse energy-delay (EDP_i) or inverse power-delay(PDP_i) as described in Section 3.3. Finally, we showcase that PARALiA also adapts better than previous approaches to a heterogeneous system, which we emulate using a different predefined per-device load in our testbed.

4.1 Experimental Setup

For the performance evaluation we use a single testbed: the “clx-ai” nodes of HLRS’ HPC cluster VULCAN, HPC cluster at HLRS [21]. System details are presented in Table 3, along with the interconnect bandwidths stored in the `LinkMap` for the 9 devices (8 GPUs + CPU which are illustrated in Figure 6). The interconnect utilizes a mix of NVlink-1 (24 GB/s) and NVlink-2 (48 GB/s) for inter-GPU connectivity and PCiE (12 GB/s) for all CPU-GPU communication. In addition, we note that CPUs share PCIe bandwidth in sets of two (e.g., GPU 0-1, 2-3, etc.). For time measurements we use wrapped timers based on `clock_gettime`, with device synchronization (`cudaDeviceSynchronize()`) also included; both timer and synchronization overhead were less than 1% for all benchmarks. We perform 20 executions for large benchmarks and 100 for small ones, after 10 warm-up runs, and we report the median time/performance of these runs. The allocation time needed for CPU/GPU buffers is not modeled or included in the total time, and all matrices/vectors are initialized with random values before execution. We use pinned host memory to enable asynchronous CUDA calls and the caches/buffers are flushed between runs. The above configuration is consistent for all our experiments and all state-of-the-art libraries we include in this work.

4.2 Evaluation Dataset

4.2.1 Routine Selection. While the PARALiA framework is designed to support all BLAS levels, level-1 and level-2 are rarely offloaded to GPUs/accelerators as standalone calls - they usually follow or precede level-3 BLAS invocations which can fully utilize the extreme computational capabilities of GPUs. We therefore only implement a subset of BLAS routines (`axpy`, `dot`, `gemv`) as proof-of-work with the PARALiA’s wrappers and do not include any level-1 or level-2 BLAS routines in our evaluation. On the other hand, the usual evaluation trend for multi-GPU level-3 BLAS publications is to report the performance of most or all level-3 BLAS routines they implement, for multiple supported datatypes. Due to the very high resource cost of benchmarking multi-GPU level-3 BLAS, this usually leads to small datasets with specific characteristics, which as we mentioned in Section 1, is the main problem of previous approaches. Due to this, their evaluations explore only a fraction of potential problems, resulting in potential underlying bottlenecks never brought to light. We choose a different benchmarking approach for PARALiA; we select a large, diverse dataset and focus solely on double-precision floating-point matrix-matrix multiplication (dgemm) for performance evaluation. We make this choice for three reasons. First, GEMM is by far the most common level-3 BLAS routine; all other level-3 BLAS kernels are either datatype-specialized GEMM implementations or internally perform mostly GEMM computations (68-93% according to BLASX [9], which increases further with problem size), and therefore follow similar performance trends. Second, as the most generic level-3 BLAS routine, GEMM, depending on its input/output size and shape, results in a plethora of different arithmetic intensities, which can be used to expose bottlenecks for transfer-, memory- and compute-bound problems with a single implementation. Third, because our total resources are limited, we prefer to cover a diverse dataset to expose hidden bottlenecks, instead of presenting similar results for multiple routines.

4.2.2 Dataset Characteristics. Since most state-of-the-art multi-GPU level-3 BLAS libraries use the same cuBLAS single-GPU routines at the backend, they have similar performance peaks when communication is not a bottleneck. We therefore try to include a good percentage of problems that potentially have performance differences due to communication/scheduling. The main characteristics of GEMM that change its communication/computation ratio are the problem size and problem shape, and additionally, for multi-GPU setups, the initial residing memory for each of the input/output matrices. We consequently explore 21 square problem sizes ($M_{sq} = N_{sq} = K_{sq} = (2 \xrightarrow{\text{step=1}} 22) \cdot 2^{10}$), 21 fat-by-thin problems ($M_{fat} = N_{fat} = (8 \xrightarrow{\text{step=4}} 32) \cdot 2^{10}, Kthin = \frac{M_{fat}}{r}, r \in [2, 8, 32]$) and 21 thin-by-fat problems ($K_{fat} = (12 \xrightarrow{\text{step=4}} 36) \cdot 2^{10}, M_{thin} = N_{thin} = \frac{K_{fat}}{r}, r \in [2, 8, 32]$) for 10 location combinations (more in Figure 7) for a total of 630 problems. We assume each matrix initially exists in a single location and is not pre-distributed to multiple devices, to maintain compatibility with the BLAS API standard and to be able to compare performance with existing multi-GPU BLAS libraries, which also follow the standard. For each such problem, we measure the execution time t of 1) *cuBLASXt*, 2) *BLASX*, 3) *XKBLAS* and 4) four PARALiA variants (*PARALiA comm_opt*, *PARALiA select(PERF)*, *PARALiA select(EDP$_i$)*, *PARALiA select(PDP$_i$)*). *PARALiA comm_opt* only optimizes communication without employing device selection, while the other three versions also select the best device configuration for optimizing the relevant metric. We exclude other libraries like SuperMatrix [3] and PARSEC [6] that were designed taking older GPU architectures into account, as they are outperformed by both BLASX and XKBLAS.

4.3 Comparison with State-of-the-art

4.3.1 Performance. Figure 7 shows the evaluation results for the entire dataset. As previous work also outlines, *cuBLASXt* has very low performance due to its static round-robin decomposition as well as the absence of a data caching and reuse logic. On the other hand, *BLASX* provides good performance for the full-offload (h,h,h) scenario for initial data locations, which drops considerably in all other location combinations. This pattern holds for all three data shapes, and is more evident in fat-thin and thin-fat problems because they are more communication-bound than the square shape, for which GEMM has the highest arithmetic intensity. *XKBLAS* follows a similar pattern, with only one distinguishable characteristic; it has the highest full-offload (h,h,h) performance of all libraries, but the performance degradation in all other location combinations is much larger than *BLASX*, resulting in inferior performance. We attribute this to the extra heuristics *XKBLAS* uses for limiting writebacks and task scheduling and its very lightweight scheduler, which are designed around the optimization of the full-offload scenario. While we are working on overcoming both those issues, we also believe that this would not occur in modern

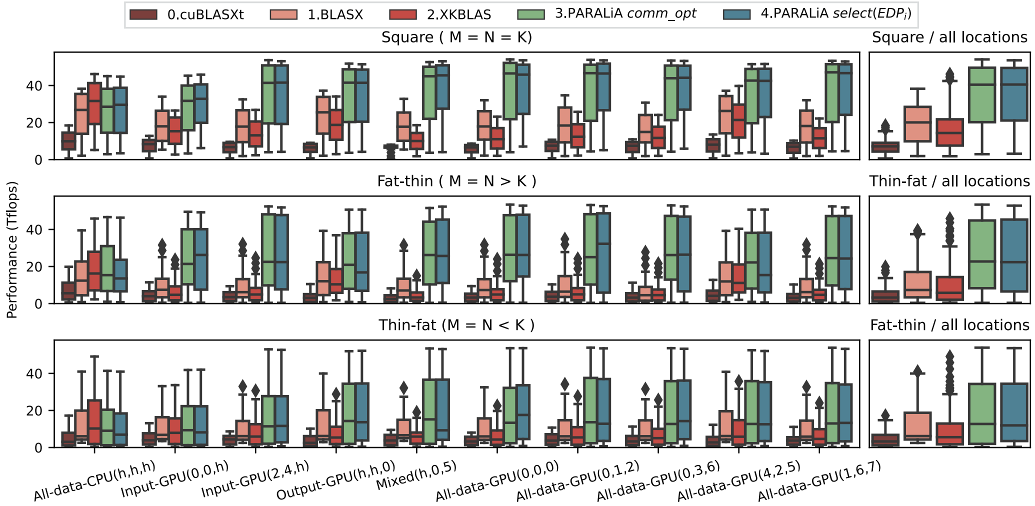

Fig. 7. Performance of dgemm with cuBLASXT, BLASX, XKBLAS, and two variants of PARALiA (one always utilizing all GPUs and one selecting the workload distribution to maximize the inverse energy-delay product - EDP_i), on the dataset described in Section 4.2. Each row corresponds to a different data shape M, N, K and each boxplot group corresponds to a different data location, with $gemm_{loc} = (A_{loc}, B_{loc}, C_{loc})$, where loc = h corresponds to data on host and loc = dev_{id} to the corresponding device's memory. The right subfigure aggregates results for each problem shape.

systems that are not as heavily bound by h2d PCIe transfers. The simpler BLASX is better in this case, since writing back to the host and then re-fetching to the GPUs with h2d/d2h transfers (PCIe bandwidth = 12 GB/s), is better than performing d2d between *distant* devices which results in extremely slow transfers through PCIe, bridges and potentially NUMA connections (bandwidth < 6 GB/s), which cannot be overlapped. Nevertheless, this is a very interesting observation - the dethroning of *BLASX* by *XKBLAS* as the state of the art for multi-GPU setups was based only on the full-offload comparison. Looking behind the curtain, *BLASX* does provide more robust multi-GPU performance in the general case - which further stresses the importance of a more diverse dataset for a fair performance evaluation. Both PARALiA implementations offer a 1.8-2X mean performance improvement over BLASX and XKBLAS and exhibit superior performance for all location and shape configurations, except full-overlap, where our choice to not use sub-kernel order selection heuristics gives XKBLAS a 5-10% performance advantage. The performance gain versus *BLASX* and *XKBLAS* varies for all other configurations, with the two PARALiA implementations displaying almost similar performance and ultimately approaching peak performance (e.g., being compute-bound) by better utilizing the faster NVLink connections due to the optimized `LinkMap`. In summary, Figure 7 illustrates that PARALiA outperforms previous approaches in terms of performance (details in Section 4.3.3) in a complete, diverge dataset, containing various transfer- and compute- bound cases, due to its better communication optimization scheme.

4.3.2 Energy Efficiency. Figure 8 presents results on energy efficiency for our dataset using the inverse power-delay product (PDP_i in Gflops/W). Both PARALiA implementations have superior PDP_i than the state of the art, which for PARALiA comm_opt versus *cuBLASXt*, *BLASX*, *XKBLAS* is due to their performance difference, since they all utilize all 8 available GPUs. On the other hand, *PARALiA select* (EDP_i) has the best PDP_i for all configurations, providing an 8% higher average PDP_i than *PARALiA comm_opt* with only 0.5% less mean performance. It is also evident

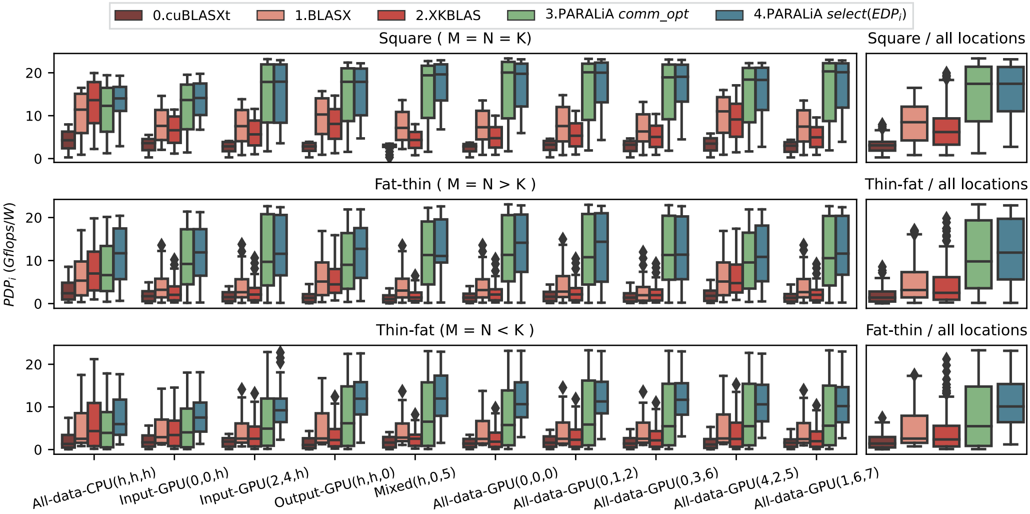

Fig. 8. Energy efficiency of dgemm (Gflops/W) for all problem configurations presented in Figure 7. cuBLASXt, BLASX, XKBLAS and PARALiA *comm_opt* have PDP_is relative to their performance (since they all utilize all 8 system GPUs), resulting in a much better PDP_i for PARALiA due to its higher performance. On the other hand, PARALiA *select(EDP_i)* also takes into account the energy-performance relation when considering how many devices to use and therefore has a much better PDP_i with only imposing a minor performance difference.

that the mean PDP_i improvement via selection mostly affects smaller problems (boxplots lower parts defer more) and depends on problem shape (Mean improvement: sq = 1%, fat-thin = 8%. thin-fat = 15%). Both these behaviors derive from the fact that device selection is only meaningful for partially communication-bound problems, since for purely computation-bound ones selecting all devices will always yield the highest EDP_i. Summing up, PARALiA provides the highest energy efficiency for all configurations, coupling better overall performance with efficient device selection for communication-bound problems.

4.3.3 In-depth Analysis. While PARALiA's communication optimizations affect most of the dataset, making their performance contribution easily distinguishable, device selection benefits only problems that still remain communication-bound after the aforementioned optimization. Consequently, since in Figures 7 and 8 such problems are overshadowed by the compute-bound portion of the total dataset, we include Table 4 to better demonstrate the effect of selection by splitting the dataset to two equal (310-320) parts, denoted small (S) and large (L), along with the total (T) dataset mean values. We also include two other versions of PARALiA selection, *select(PERF)* and *select(PDP_i)*, to showcase the effect of different optimization metrics, and make three basic observations.

First, the means show how *PARALiA comm_opt*, *PARALiA select(PERF)* and *PARALiA select(EDP_i)* vastly outperform all previous approaches in terms of performance in all (S, L, T) problems, with *PARALiA select(PERF)* having a (geo)mean performance improvement in (S, L, T) of (4.6×, 5.6×, 5.1×) over *cublasXt*, (1.6×, 2.0×, 1.8×) over *BLASX* and (2.3×, 2.5×, 2.4×) over *XKBLAS*, and *PARALiA select(EDP_i)* (4.3×, 5.5×, 4.8×) over *cublasXt*, (1.5×, 2.0×, 1.7×) over *BLASX* and (2.2×, 2.5×, 2.3) over *XKBLAS*. For all the above cases, the communication optimization yields similar results in (S, L, T), with slightly better speedup on large problems due to previous libraries struggling with the interconnect optimization, while PARALiA already reaches compute-bound

Table 4. A Summary of the Performance of dgemm for the Whole Dataset for Each Implementation, using the $\frac{mean(Gflop)}{mean(metric)}$ [22]

Implementation	Performance (Gflops)			PDP_i (Gflops/W)		
	Small (S)	*Large (L)*	***Total (T)***	*Small (S)*	*Large (L)*	***Total (T)***
cuBLASXt	1827	8516	7484	0.79	3.68	3.23
BLASX	4913	24755	21486	2.12	10.69	9.28
XKBLAS	3569	18572	15895	1.54	8.02	6.86
PARALiA comm_opt	9396	45840	39996	4.06	19.79	17.27
PARALiA select(PERF)	10641	46066	40933	5.32	19.94	18.06
PARALiA select(EDP_i)	9453	45433	39804	6.30	20.16	18.64
PARALiA select(PDP_i)	3970	6700	6530	13.71	23.14	22.56

Small problem (S), large problem (L) and total dataset (T) percentages are displayed separately for extra clarity regarding the underlying performance. *PARALiA comm_opt*, *PARALiA select(PERF)* and *PARALiA select(EDP_i)* vastly outperform previous approaches, with *PARALiA select(PERF)* offering the best performance and *PARALiA select(EDP_i)* being more balanced between performance and energy efficiency as intended. *PARALiA select(PDP_i)* leads to relatively low performance coupled with the best PDP_i.

performance earlier. *PARALiA select(PDP_i)* on the other hand leads to vastly inferior performance, since PDP_i alone is a bad metric in multi-GPU due to often selecting 1 GPU to provide the most flops/W.

Second, all PARALiA implementations also outperform previous approaches in terms of energy efficiency, with *PARALiA select(PERF)* having a (geo)mean PDP_i improvement in (S, L, T) of (7.8×, 5.6×, 6.6×) over *cublasXt*, (2.7×, 2.0×, 2.3×) over *BLASX* and (4.0×, 2.5×, 3.2×) over *XKBLAS*, *PARALiA select(EDP_i)* (9.0×, 5.7×, 7.1×) over *cublasXt*, (3.1×, 2.0×, 2.5×) over *BLASX* and (4.6×, 2.6×, 3.4×) over *XKBLAS* and *PARALiA select(PDP_i)* (17.0×, 7.0×, 10.8×) over *cublasXt*, (5.8×, 2.5×, 3.8×) over *BLASX* and (8.6×, 3.2×, 5.2×) over *XKBLAS*. Unlike performance which is mainly driven by the LinkMap optimization, the additional PDP_i improvement derives from device selection, which is evidently higher in the small (S) problems where most selection occurs. As anticipated, *PARALiA select(PDP_i)* offers the best energy efficiency by far, since the selection target is also the evaluation metric, *select(PERF)* improves *PARALiA comm_opt* PDP_i as a byproduct of using fewer devices when they provide similar performance and *select(EDP_i)* provides a solid PDP_i in between the other two (leaning towards performance) as intended.

Third, we consider *PARALiA select(EDP_i)* to provide the best performance-PDP_i trade-off, focusing on performance but also accounting for energy efficiency in order to avoid very inefficient choices (like for example using 8 devices to get a 5% speedup from using 2), resulting in huge PDP_i improvement in small problems (1.5× over *PARALiA comm_opt*) with a similar performance. This energy efficiency improvement is *virtually free* to the user, deriving solely from performance awareness, and is the main motivation behind our work. Additionally, the means for *select(PERF)* and *select(EDP_i)* show that selection can also lead to better performance even disregarding energy whatsoever, depending on the communication-boundedness of the problem. Summing up, PARALiA vastly outperforms previous approaches in terms of both performance and energy efficiency, with *PARALiA select(EDP_i)* offering near-optimal performance due to communication optimization coupled with superior PDP_i due to performance awareness delivered from the auto-tuning runtime.

4.4 Applicability to Heterogeneous Platforms

Device selection in homogeneous systems is meaningful for communication-bound problems but can be even more beneficial in heterogeneous systems, where devices can have different

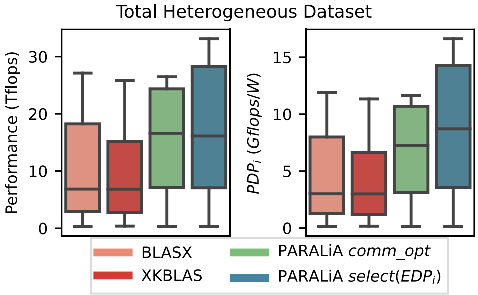

Fig. 9. DGEMM performance (Tflops) and energy efficiency (Gflops/W) for half of the problem configurations presented in Figure 7. BLASX and XKBLAS schedulers do not adjust well to different computation devices, while PARALiA still provides improved performance and PDP_i, which is boosted by better workload distribution.

Table 5. A Table Summarizing the GEMM Performance for the Whole Dataset for Each Implementation, using the $\frac{mean(Gflop)}{mean(metric)}$ [22] for Half of the Small (HS), Large (HL) and Total (HT) Problems of Table 4 Ran on a Heterogeneous Emulated System

Implementation	Performance (Gflops)			PDP_i (Gflops/W)		
	(HS)	*(HL)*	***(HT)***	*(HS)*	*(HL)*	***(HT)***
BLASX	5036	18621	17316	2.21	8.2	7.6
XKBLAS	5379	16782	15520	2.36	7.4	6.8
PARALiA comm_opt	12056	24892	24146	5.5	10.9	10.6
PARALiA select(PERF)	12160	27221	28117	6.2	14.7	13.2
PARALiA select(EDP_i)	9453	26154	25645	7.9	15.3	15.1

PARALiA outperforms all multi-GPU scheduler-based approaches both in performance and energy efficiency, further boosted by a better workload selection.

computation capabilities. While heterogeneous multi-device systems are not common nowadays, computational heterogeneity will probably be more commonplace in the future. Heterogeneous-like execution scenarios can also appear in current homogeneous multi-device systems, by applying different power management policies or sharing devices between users/processes. For this reason, we include a proof-of-concept application of our approach to an artificial heterogeneous system, which we emulate by loading the GPUs of the Vulcan clx-ai cluster with different computation workloads running in other processes. We configure these workloads empirically to represent GPUs with lower performance, resulting in the following double FP peak adjustments: $GPU_{\{0,1,4,6\}} = 3.5$ Tflops, $GPU_{\{2,3\}} = 5$ Tflops and $GPU_{\{5,7\}} = 7$ Tflops (original peak). We also do not adjust the power consumption of each device, resulting in different energy efficiency for each device category (e.g., $GPU_{\{5,7\}}$ are more energy efficient than $GPU_{\{2,3\}}$, etc.). This leads to a total system DP peak of 38 Tflops (vs 56 Tflops for the original system), and a PDP_i peak of 17.5 (vs 26 in the original system). We also note that a homogeneous-distribution DP peak (without load-balancing) is $3.5 \cdot 8 = 28$ Tflops for future reference. Additionally, we limit the dataset to less than half the problems, by doubling the data size iteration steps and the minimum size for a total of 250 problems, and exclude *cublasXT* due to extreme benchmark times and having no load-balancing mechanism whatsoever. The results for this heterogeneous-emulated system are shown in Table 5 and Figure 9.

Figure 9 contains the aggregated performance (left) and energy efficiency-PDP_i (right) boxplots for the entirety of the aforementioned heterogeneous dataset of 250 problems (HS - 90, HL - 160), which now leans more towards computation-bound problems since the peak performance has lowered considerably while the interconnect is the same. *BLASX*, *XKBLAS* and *PARALiA comm_opt* follow similar performance and PDP_i patterns with the homogeneous system, albeit at lower peak as expected. Communication optimization is still an issue for *BLASX* and *XKBLAS* resulting in superior performance for *PARALiA comm_opt*, but all three approaches are limited by the aforementioned homogeneous-distribution peak around 28 Tflops. On the other hand, *PARALiA select(EDP_i)* manages to provide a better workload distribution, which results in better performance by approaching the peak of 38 Tflops for large problems, and has far superior PDP_i both due to better performance and due to awareness of the different characteristics of each emulated device. Summing up, *PARALiA select(EDP_i)* vastly outperforms previous approaches both in terms of performance and energy efficiency in the heterogeneous system as well, now further boosted by a better workload distribution in different devices.

Table 5 shows the achieved mean performance in a similar layout with Section 4.3 for the small (HS), large (HL) and total (HT) dataset also displaying results for *PARALiA select(PERF)*. *PARALiA select(PDP_i)* is omitted due to always selecting from $GPU_{\{5,7\}}$ as expected without adding any additional insights. Since the performance is already visualized in Figure 9 we use Table 5 for problem size-related insights and mean comparison, making the following observations. First, PARALiA versions still outperform previous approaches in all subsets, but with a smaller performance difference added to the baseline *PARALiA comm_opt* from communication optimization. Specifically, PARALiA select(PERF) has a (geo)mean performance improvement in (S, L, T) of (2.2×, 1.7×, 1.8×) over BLASX and (2.0×, 1.9×, 1.9×) over XKBLAS, and PARALiA select(EDP_i) has (1.8×, 1.6×, 1.6×) over BLASX and (2.0×, 1.7×, 1.7×) over XKBLAS. This is expected, since by reducing peak performance and removing smaller problems from the total dataset, the impact of communication optimization is limited since many problems now become compute-bound. Second, the impact of workload selection increases performance and energy efficiency both for *PARALiA select(PERF)* and *PARALiA select(EDP_i)* in respect to *PARALiA comm_opt*, since devices with lower computational power are used for a smaller part of the problem or omitted by the auto-tuning runtime. This is more visible in the large (HL) problems which are compute-bound, since in small (HS) problems the communication optimization is still more important. The difference between *PARALiA select(PERF)* and *PARALiA select(EDP_i)* becomes more evident when comparing them with *PARALiA comm_opt*; *PARALiA select(PERF)* offers 1.12× mean performance and 1.3× mean PDP_i improvement while *PARALiA select(EDP_i)* offers 1.06× performance and 1.5× PDP_i. Summing up, both *PARALiA select(PERF)* and *PARALiA select(EDP_i)* benefit from workload selection, offering different insights and balance between performance and energy in the heterogeneous system, outlining the increased importance of additional metrics for such future systems.

5 RELATED WORK

This section provides basic background on BLAS modeling and auto-tuning, its role in BLAS optimization and how it is applied on different architectures. We first focus on research work on CPU BLAS, as it includes the first approaches of autotuning at runtime for performance improvement. We then look at research work on GPU BLAS, which is concerned with computation and communication performance prediction, therefore offers background on modeling. Finally, we discuss the background in hybrid CPU-GPU approaches, which are relevant to the problem of splitting a workload appropriately to utilize different computational resources, encapsulating heterogeneity challenges.

5.1 CPU BLAS Autotuning

Since BLAS is an important part of a plethora of scientific code and solvers, from the early days of computing it played a vital role in scientific code optimization. Dongarra [1] firstly defined the BLAS standard as a set of “black box” routines that should follow a specific input/output layout and should be optimized by vendors and library providers transparently to the user, without any additional performance tuning from the side of the user. This black-box approach required performance engineering effort to reimplement or tune BLAS routines for every new generation of hardware. To offset this, Whaley et al. [11] implemented ATLAS, the most well-known BLAS autotuning framework, which uses their **automated empirical optimization of software (AEOS)** technique. ATLAS explores many possible routine implementations, tuning itself to each new system empirically by testing them and timing them, in order to improve cache utilization, increase parallelism and load balance sub-problems. In a similar notion, Van Zee and Vsan de Geijn [23] defined a set of a few kernels that can be used to optimize all BLAS 2 and BLAS 3 operations using empirical knowledge, and later Low et al. [24] proved that an analytic approach is sufficient for replacing the empirical part with auto-tuning. These techniques are not used directly in our work, but they form the groundwork for most BLAS-related autotuning and model-based optimization.

5.2 GPU BLAS Autotuning

Regarding GPU BLAS implementations, cuBLAS [2] was the first library to provide high performance on a single GPU, expecting, however, the data to be available in the GPU. cuBLAS autotunes the CUDA kernel block sizes internally, based on pre-performed offline empirical testing for each NVIDIA GPU. When the data reside outside the GPU memory, the execution of a BLAS routine requires data transfers as well. Gregg and Hazelwood [25] were the first to highlight the problem of data transfer overheads, arguing against the trend to exclude transfer overheads in the scientific reporting of the performance of GPU applications, and proposed a taxonomy for data transfers and their impact on offload performance. Numerous works model CPU-GPU transfers, using variants of the linear latency-bandwidth model for PCIe transfers [17, 26–28] and machine learning [27] to provide user insights, without, however, specifically targeting BLAS, or integrating them in an autotuner. The inclusion of transfers in GPU performance prediction improved accuracy, but whenever there was communication/computation overlap, simplistic models still failed to predict the actual performance. As highlighted by Hoefler et al. [13], modeling overlap areas is a crucial step in performance modeling. Towards this direction, Gómez-Luna et al. [14] explored the use of CUDA streams for 3-way concurrency, but they consider the stream creation time as the only overlap overhead. Werkhoven et al. [15] enhanced this work by offering multiple performance models for communication/computation overlap for various common offload scenarios (RMA, 2-way, 3-way), introduced stream transfer overlap latency, and provided methods to obtain the optimal number of CUDA streams for a given problem. In a similar notion, Liu et al. [19] offered a mathematical framework for software pipelining on GPUs using non-equal tiles, which focused on partitioning, scheduling, and granularity. All these models offer high accuracy, however, their modeling approach does not capture all problem characteristics present in BLAS. Moreover, they were never used in practice for autotuning, and therefore never faced the practical problems of transfer prediction in real systems, where empirical bandwidths defer from their theoretical counterparts depending on many parameters like the type of src/dest memory (normal, unified, pinned) or the underlying sharing of interconnect resources [29–31]. In our previous work CoCoPeLia [12], which aimed to create an autotuner that would selected good tiling sizes for domain decomposition in single-GPU BLAS, we had to overcome both of these problems to enable performance estimation. To overcome the first problem, we modified these models to account for common BLAS characteristics, like data reuse and bidirectional transfers. To overcome the second problem, we created an automated pipeline which fueled these models with offline empirical tests and integrated these into an end-to-end BLAS framework, which provides state-of-the-art performance for single-GPU BLAS.

5.3 Hybrid CPU-GPU/Heterogeneous Autotuning

Several research works focus on CPU-GPU hybrid BLAS execution, a topic relevant to our work, because of the problem of heterogeneous splitting, namely the splitting of a BLAS problem into subproblems, based on performance estimations for the execution on the CPU and the GPU, integrated in an end-to-end solution. Luk et al. [32] first proposed such an end-to-end solution for model-based CPU-GPU splitting for a subset of BLAS routines. Their solution employs an initial benchmark run, through which their backend BLAS functions obtain linear models for CPU and GPU performance, store them in a database (file) and utilize them in subsequent runs, to split a problem depending on its predicted CPU and GPU performance. Tomov et al. [4] envisioned a heterogeneous LAPACK and made use of BLAS-level parallelism, where the program is represented as a collection of BLAS-based tasks and dependencies. To this end, they use task graphs to represent BLAS task dependencies and use the CPU for small kernels, the execution of which is inefficient on GPUs, and the GPU for everything else, overlapping CPU and GPU work and transfers. Humphrey et al. [33–35] developed CULA, a CUDA framework which enables GPU-CPU simultaneous execution, with each one running predefined routine parts most fit to its paradigm. Bernabé et al. [18] employed CPU-GPU hybridization auto-tuning based on micro-benchmarks run at the time of the library installation, which were used to feed polynomial model coefficients. Finally, Ma et al. [36] explored energy efficiency, splitting the problem during execution until a balance between CPU-GPU execution time is reached, and then scaling frequencies to limit energy consumption. All the above approaches do not take into account transfers and overlap in their modeling/assumptions, favoring the GPU even in cases that the problem is transfer bound if its computational capability is larger than the CPU, and do not target multi-GPU systems. This leaves a wide gap between multi-GPU libraries and hybrid/heterogeneous BLAS research. In our work, PARALiA, we bridge this gap with high-quality subproblem scheduling and communication optimization on multiple GPUs, together with performance awareness and device selection, to ensure resource utilization for any type of heterogeneity.

6 CONCLUSION

In this work, we have presented PARALiA, an end-to-end framework for multi-GPU BLAS execution. Similar to existing multi-GPU BLAS approaches, PARALiA employs problem splitting, subproblem scheduling, and computation-communication overlap to maximize the performance of BLAS routines on multi-GPU setup. Contrary to existing approaches, PARALiA puts emphasis on optimizing the communication, through a generic hardware abstraction, which allows for more accurate performance estimation, offered by an autotuner to the scheduler component within PARALiA. Additionally, PARALiA performs careful device selection, based on a pre-set optimization target, which can be performance or some energy-related metric, avoiding resource waste.

We evaluate PARALiA on a multi-GPU testbed which exposes heterogeneous connections between the devices. Our experiments focus on the performance of GEMM with double-precision, as a representative level-3 BLAS. Our evaluation shows that PARALiA outperforms the state-of-the-art BLASX and XKBLAS multi-GPU BLAS frameworks with a (geo)mean improvement of 1.7× and 2.4× respectively, with significant performance gains for routine executions where the data originally reside on the various GPUs. We additionally show how, with device selection and by setting different optimization targets, PARALiA is able to achieve high performance coupled with better energy efficiency, with a (geo)mean improvement of 2.5× versus BLASX and 3.4× versus XKBLAS. Finally, PARALiA adjusts well to a heterogeneous system with different compute capabilities among the GPUs, offering improved performance and superior energy efficiency over BLASX and XKBLAS.

We conclude that, despite the common conception that level-3 BLAS routines are well-suited for multi-GPU systems, high performance for any problem size and data location can only be achieved by minimizing communication costs. As multi-GPU setups become more heterogeneous, both resource selection and communication optimization have increasing importance for the performance and energy efficiency of BLAS libraries alike. In the future, we aim to extend PARALiA with more sophisticated scheduling techniques, and we will work towards the generalization of the autotuning approach of PARALiA and the extensibility of the PARALiA components for other important compute libraries. Finally, looking at advanced multi-GPU systems, with more balanced bandwidth levels, we aim to improve our rerouting algorithm to balance overlap and address congestion issues.

REFERENCES

[1] Jack Dongarra. 2002. Basic linear algebra subprograms technical (BLAST) forum standard II. *IJHPCA* 16 (05 2002), 1–111. DOI: http://dx.doi.org/10.1177/10943420020160020101

[2] developer.nvidia.com/cublas. ([n. d.]).

[3] E. W. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn. 2007. *SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-blocks*. Computer Science Department, University of Texas at Austin. https://books.google.gr/books?id=ggn-jwEACAAJ

[4] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5 (2010), 232–240. DOI: http://dx.doi.org/10.1016/j.parco.2009.12.005 Parallel Matrix Algorithms and Applications.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2010. Faster, cheaper, better–a hybridization methodology to develop linear algebra software for GPUs. In *GPU Computing Gems*, Wen mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann. https://hal.inria.fr/inria-00547847

[6] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. 2015. Hierarchical DAG scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 156–165. DOI: http://dx.doi.org/10.1109/IPDPS.2015.56

[7] NVIDIA cublasxt. 2015. [online] Available: https://docs.nvidia.com/cuda/cublas/

[8] NVIDIA nvblas. 2015. [online] Available: https://docs.nvidia.com/cuda/nvblas/

[9] Linnan Wang, Wei Wu, Jianxiong Xiao, and Yi Yang. 2015. BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing. *CoRR* abs/1510.05041 (2015). arXiv:1510.05041 http://arxiv.org/abs/1510.05041

[10] T. Gautier and J. V. F. Lima. 2020. XKBlas: A high performance implementation of BLAS-3 kernels on multi-GPU server. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 1–8. DOI: http://dx.doi.org/10.1109/PDP50117.2020.00008

[11] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. DOI: http://dx.doi.org/10.1016/S0167-8191(00)00087-9 New Trends in High Performance Computing.

[12] Petros Anastasiadis, Nikela Papadopoulou, Georgios Goumas, and Nectarios Koziris. 2021. CoCoPeLia: Communication-computation overlap prediction for efficient linear algebra on GPUs. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 36–47. DOI: http://dx.doi.org/10.1109/ISPASS51385.2021.00015

[13] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. 2011. Performance modeling for systematic performance tuning. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[14] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. 2012. Performance models for asynchronous data transfers on consumer graphics processing units. *J. Parallel and Distrib. Comput.* 72, 9 (2012), 1117–1126. DOI: http://dx.doi.org/10.1016/j.jpdc.2011.07.011 Accelerators for High-Performance Computing.

[15] B. v. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal. 2014. Performance models for CPU-GPU data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 11–20.

[16] NVIDIA, Péter Vingelmann, and Frank H. P. Fitzek. 2020. CUDA, release: 10.2.89. (2020). https://developer.nvidia.com/cuda-toolkit

[17] M. Boyer, J. Meng, and K. Kumaran. 2013. Improving GPU performance prediction with data transfer modeling. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum*. 1097–1106.

[18] Gregorio Bernabé, Javier Cuenca, Luis-Pedro García, and Domingo Giménez. 2014. Tuning basic linear algebra routines for hybrid CPU+GPU platforms. *Procedia Computer Science* 29 (2014), 30–39. DOI: http://dx.doi.org/10.1016/j.procs.2014.05.003 2014 International Conference on Computational Science.

[19] Bozhong Liu, Weidong Qiu, Lin Jiang, and Zheng Gong. 2016. Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity. *The International Journal of High Performance Computing Applications* 30, 2 (2016), 169–185. DOI: http://dx.doi.org/10.1177/1094342015585845 arXiv:https://doi.org/10.1177/1094342015585845

[20] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2009. LogGP in theory and practice–an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations. *Simulation Modelling Practice and Theory* 17, 9 (2009), 1511–1521.

[21] VULCAN, HPC cluster at HLRS, [online], Available: https://kb.hlrs.de/platforms/index.php/NEC_Cluster_Hardware_and_Architecture_(vulcan)

[22] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. DOI: http://dx.doi.org/10.1145/2807591.2807644

[23] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. http://doi.acm.org/10.1145/2764454

[24] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Software* 43, 2 (Aug. 2016), 12:1–12:18. http://doi.acm.org/10.1145/2925987

[25] C. Gregg and K. Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. 134–144.

[26] D. Schaa and D. Kaeli. 2009. Exploring the multiple-GPU design space. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12.

[27] Ali Riahi, Abdorreza Savadi, and Mahmoud Naghibzadeh. 2020. Comparison of analytical and ML-based models for predicting CPU-GPU data transfer time. *Computing* (January 2020).

[28] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. 2012. Modeling and predicting performance of high performance computing applications on hardware accelerators. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 1828–1837.

[29] W. Li, G. Jin, X. Cui, and S. See. 2015. An evaluation of unified memory technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1092–1098.

[30] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 1–10.

[31] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 209–218.

[32] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 45–55.

[33] John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini, and Eric J. Kelmelis. 2010. CULA: Hybrid GPU accelerated linear algebra routines. In *Defense + Commercial Sensing*.

[34] Kyle E. Spagnoli, John R. Humphrey, Daniel K. Price, and Eric J. Kelmelis. 2011. Accelerating sparse linear algebra using graphics processing units. In *Defense + Commercial Sensing*.

[35] John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, and Eric J. Kelmelis. 2012. Accelerating CULA linear algebra routines with hybrid GPU and multicore computing. In .

[36] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. 2012. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *2012 41st International Conference on Parallel Processing*. 48–57. DOI: http://dx.doi.org/10.1109/ICPP.2012.31

Received 21 February 2023; revised 13 July 2023; accepted 23 August 2023