National Technical University of Athens

School of Electrical and Computer Engineering

Division of Computer Science

Model-assisted optimization of Linear Algebra routines on multi-GPU computing systems

Ph.D. Thesis

Petros Anastasiadis

Athens, September 2024

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Model-assisted optimization of Linear Algebra routines on multi-GPU computing systems

Ph.D. Thesis

Petros Anastasiadis

Advisors:

Georgios Goumas
Nectarios Koziris
Nikela Papadopoulou

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 09 Σεπτεμβρίου, 2024.

Georgios Goumas
Associate Professor
National Technical University of Athens

Nectarios Koziris
Professor
National Technical University of Athens

Nikela Papadopoulou
Assistant Professor
University of Glasgow

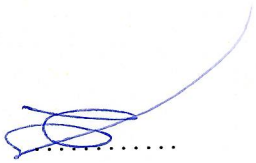
Dionisios N. Pnevmatikatos
Professor
National Technical University of Athens

Nikolaos Papaspyrou
Professor
National Technical University of Athens

Sotirios Xydis
Assistant Professor
National Technical University of Athens

Christos Antonopoulos
Professor
University of Thessaly

Athens, September 2024



Πέτρος Γ. Αναστασιάδης
Διδάκτωρ Ε.Μ.Π.

Περίληψη

Οι πράξεις γραμμικής άλγεβρας εμφανίζονται συχνά σε εφαρμογές υψηλής απόδοσης (HPC), καθιστώντας την απόδοσή τους κρίσιμη για την επίτευξη βέλτιστης κλιμάκωσης. Καθώς πολλές σύγχρονες συστάδες HPC περιλαμβάνουν κόμβους με πολλαπλούς επεξεργαστές γραφικών (GPUs), οι πράξεις BLAS συχνά εκφορτώνονται σε GPUs, καθιστώντας απαραίτητη τη χρήση βελτιστοποιημένων βιβλιοθηκών για τη διασφάλιση της απόδοσης. Ωστόσο, η βελτιστοποίηση των BLAS σε πολλαπλές GPUs εισάγει πολλές προκλήσεις παρόμοιες με εκείνες του κατανεμημένου υπολογισμού, όπως την αποσύνθεση δεδομένων, την δρομολόγηση υποπροβλημάτων και την επικοινωνία μεταξύ GPU με διακριτές μνήμες. Αυτή η πολυπλοκότητα καθιστά την βελτιστοποίηση των BLAS πολύ περίπλοκη, οδηγώντας σε πτώση απόδοσης ή μεμονωμένες λύσεις που λειτουργούν μονο σε μερικά σύστήματα. Για να αντιμετωπίσουμε αυτά τα ζητήματα, προτείνουμε μια προσέγγιση αυτόματης βελτιστοποίησης με τη βοήθεια μοντελοποίησης: εισάγουμε διάφορα μοντέλα απόδοσης για τις BLAS και τα ενσωματώνουμε στο PARALiA, μια ολοκληρωμένη βιβλιοθήκη BLAS. Το PARALiA χρησιμοποιεί μοντέλα επίδοσης για την δυναμική αυτόματη βελτιστοποίηση της εκτέλεσης των BLAS, προσαρμόζοντας κρίσιμες παραμέτρους απόδοσης για κάθε συγκεκριμένο πρόβλημα και σύστημα κατά την εκτέλεση. Αυτή η αυτόματη βελτιστοποίηση συνδυάζεται με έναν δρομολογητή εργασιών, οδηγώντας σε αποδοτική κατανομή δεδομένων και απόδοση πόρων. Το PARALiA παρέχει κορυφαία απόδοση και ενεργειακή αποδοτικότητα και ενσωματώνει την ικανότητα προσαρμογής σε ετερογενή συστήματα και σενάρια μέσω αποφάσεων βασισμένων σε μοντέλα. Τέλος, εστιάζουμε στον πυρήνα GEMM, επεκτείνοντας το PARALiA με έναν προσαρμοσμένο στατικό δρομολογητή που ενσωματώνει νέες βελτιστοποιήσεις στον αλγόριθμο και την επικοινωνία της GEMM βασισμένες σε μοντέλα (PARALiA-GEMMex), ο οποίος παρέχει σημαντικά υψηλότερη απόδοση από τις προηγούμενες βιβλιοθήκες.

Λέξεις Κλειδιά

Γραμμική άλγεβρα, Επεξεργαστές γραφικών, Ρουτίνες BLAS, Πολλαπλασιασμός πινάκων, Μοντελοποίηση, Αυτόματη βελτιστοποίηση, Συστήματα πολυεπεξεργαστών, Βιβλιοθήκες λογισμικού, Δρομολόγηση επικοινωνίας, Βελτιστοποίηση επικάλυψης

Abstract

Dense linear algebra operations appear frequently in high-performance computing (HPC) applications, rendering their performance crucial to achieving optimal scalability. As many modern HPC clusters contain multi-GPU nodes, BLAS operations are frequently offloaded on GPUs, necessitating optimized libraries to ensure good performance. However, optimizing BLAS for multi-GPU introduces numerous challenges similar to distributed computing, like data decomposition, task scheduling, and communication across GPUs with distinct memory spaces. This complexity of multi-GPU makes BLAS optimization very complex, leading to sub-optimal performance or system-specific solutions with reduced portability. To address these issues, we suggest a model-based autotuning approach: we introduce several performance models for BLAS and integrate them into PARALiA, an end-to-end BLAS library. PARALiA uses model-driven insights to dynamically autotune BLAS execution, tailoring performance-critical parameters for each specific problem and system during runtime. This autotuning is coupled with an optimized task scheduler, leading to near-optimal data distribution and performance-aware resource utilization. PARALiA provides state-of-the-art performance and energy efficiency and incorporates the ability to adapt to heterogeneous systems and scenarios via model-based decisions. Finally, we focus on the GEMM kernel, extending PARALiA with a custom static scheduler that integrates model-driven algorithmic, communication, and autotuning optimizations (PARALiA-GEMMex), which delivers significantly superior performance compared to the state-of-the-art.

Keywords

Linear algebra, Graphics processing units (GPUs), BLAS routines, Matrix-matrix multiplication, Modeling, Autotuning, Multi-GPU systems, Software libraries, Communication routing, Overlap optimization

Acknowledgments

First and foremost, I would like to thank my advisors, Georgios Goumas and Nikela Papadopoulou for their continuous guidance, support, and encouragement throughout my PhD journey. Their insights and mentorship have played a crucial role in shaping both my research and personal growth during this process. Without them, I probably wouldn't have a PhD at all, and even if I did, it would have most likely been incomprehensible to the unsuspecting reader.

In addition, a special thanks goes to my colleagues and friends in Cslab, especially Vasiliki, Ioanna, Panagiotis and Aristomenis, for their *camaraderie*, support, patience and considerable resistance to my endless complaining. They made the bad days tolerable, the average days interesting and the good days better.

Similarly, I want to thank my closest friends for their various contributions in my life, including (but not limited to): George(*Kopeli*) for being there and for being himself. Nikos(*Stag*) for always being overwhelmingly positive. George(*Ab*) for always being soothingly negative. Antonis for helping me grow up. Dinos for helping me to not grow up too much. Fanis for the endless constructive criticism. They all could have left me long ago to escape my endless bickering, but they didn't.

I also want to thank my cat, Clara, for the company. She probably would have left me if she could, but she couldn't, so she didn't.

Finally, I owe my deepest thanks to my family for their love and encouragement. To my parents, since their constant support and implausible belief in me have been a constant anchor in a sea of uncertainty. To my sister Elli, since her strength of character and undefeatable determination have been something to look up to. To my *little* sister Eva, for being a good friend. To my partner, [NULL pointer exception]. This journey would not have been possible without you.

Thank you all.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Dense linear algebra operations form the foundational building blocks of many computational algorithms which appear in a plethora of high-performance computing (HPC) applications, including Computational Fluid Dynamics (CFDs), climate modeling, molecular dynamics, image processing, machine learning, and computer vision. Additionally, while these low-level blocks appear in high-level application code that also performs other operations, typically they dominate the execution time ofthe application. Consequently, increasing the performance of dense linear algebra kernels directly impacts the overall effectiveness of HPC applications. This led to the standardization of the Basic Linear Algebra Subprograms (BLAS) [39, 40, 44–49, 90, 101] in the early days of HPC to ease 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 BLAS standard defines a set of "black box" routines that should follow a specific input/output layout and be optimized by vendors and library providers transparently to the user, without requiring additional performance tuning. But, while BLAS libraries using the "black-box" approach ease application development for the domain expert, internally they requires considerable performance engineering effort to achieve high performance. Additionally, even if all BLAS routines are exhaustively optimized for a system, the emergence of new systems poses the practical problem that performance engineers have to revisit all BLAS implementations and tune them again.

The good - CPU BLAS portability through autotuning: A common solution for this problem is to *automate* parts of the system-specific optimization process, commonly referred to as *autotuning*. Autotuning has been established from the early days of CPU BLAS optimization as a standard procedure to increase performance and portability to new systems [15, 29, 60, 61, 96, 107, 140, 158, 160, 163, 164]. While the exact process for BLAS differs per routine, architecture type, and implementation, the concept of autotuning is to automatically adjust and optimize execution towards an optimization target (execution time, energy efficiency, or any other performance metric(s)). Specifically for BLAS, it usually involves finding which *external routine parameters* (problem dimensions, flags, etc) influence performance and adjusting *internal parameters* (block-/tile size, loop unrolling factors, etc) during runtime based on the given external parameters. This requires establishing a *relation* between the values of internal and external parameters, which can be obtained with *benchmarking or modeling*. Benchmarking yields the most accurate results but typically involves a prohibitive search space, while modeling is faster and does not require exhaustive empirical testing, but is less accurate. Additionally, a practical *midway* solution is to use a combination of the two by coupling generic models with fast micro-benchmarks.

Takeaway BLAS is important for HPC applications and must have high performance and portability. This is commonly achieved through autotuning, guided by modeling and micro-benchmarks.

The bad - GPU BLAS offload: The introduction of GPUs in high-performance computing (HPC) clusters changed the landscape of BLAS optimization. The regular parallelism of BLAS routines made them a good fit for GPUs, which led to the development of many GPU-BLAS libraries, the most common being *cuBLAS*, a CUDA-like BLAS library for NVIDIA GPUs [122]. *cuBLAS* offers highly optimized primitive BLAS operations, but unlike the simpler CPU paradigm it requires the input data to reside on the GPU memory. This means that the user must also manage data transfer to and from the GPU before and after execution (henceforth referred to as *offload*), introducing a new performance bottleneck [7, 21, 63, 103, 115, 116, 130, 134, 138]. Additionally, executing problems on data that cannot fit into the GPU memory forces the user to split the initial problem data into smaller chunks and offload them to the GPU in a pipelined manner. This technique introduces extra computation and communication overheads, but also enables the overlap of communication with computation to increase offload performance [21, 63, 65, 159]. These new characteristics of GPU BLAS offload introduce a range of optimization decisions, such as determining whether a routine should be offloaded to a specific GPU, deciding the percentage of the workload that should be executed on the CPU versus the GPU if run in a hybrid manner [5, 16, 41, 79, 80, 108, 110, 145, 148–150], and selecting the appropriate chunk size for splitting the problem [21, 63, 65, 106, 159]. Although previous work has addressed these topics for specific system configurations, the rapid advancement of GPU technology over the past

decade has lead to systems with extremely higher computational capabilities and complex interconnects, rendering these approaches outdated and non-portable. While this problem could be solved with autotuning, it poses a significant challenge due to the gap between GPU kernel performance modeling [13, 35, 78, 81, 89, 109, 111, 113, 114, 127, 131, 136, 138, 165], transfer modeling [6, 21, 25, 34, 52, 63, 72, 73, 75, 76, 83, 88, 113, 115, 117, 134], overlap modeling [21, 63, 65, 106, 159] and actual GPU BLAS implementations that support overlap [4, 54, 55, 57, 68, 92, 124, 125, 157] .

Takeaway Autotuning can considerably benefit GPU BLAS, but is harder to implement due to the added complexity of GPU offload and the gap between GPU implementations and modeling.

And the ugly - Multi-GPU BLAS offload: The success of GPUs in HPC workloads has lead to a widespread adoption of multi-GPU nodes, typically consisting of 4-8 GPUs interconnected with a custom topology. Due to the extreme computational capabilities of these clusters, they are a good fit for the computationally-heavy level-3 BLAS operations. But, optimizing BLAS operations for multi-GPU nodes differs significantly from single-GPU, since it also requires to efficiently distribute and manage *data* and computational *tasks* across multiple *workers* (the GPUs) with distinct memories. Additionally, the existence of the district GPU memories further complicates execution, since a routine's input data can reside on host memory, on GPU memory, or a combination of both. Consequently, multi-GPU optimization introduces new algorithmic concepts to BLAS similar to distributed computing, such as data decomposition, task scheduling, and communication. All these add to the inherent complexity of BLAS and have motivated many libraries to support multi-GPU execution, either by extending previous distributed approaches with GPU support [4, 5, 11, 18, 19, 54, 55, 68, 92, 94, 161] or with specialized libraries specifically designed for this case [9, 57, 124, 125, 157]

Similarly to single-GPU, but even more so due to the added complexity of multi-GPU execution, the prevalent bottlenecks of each BLAS problem vary based on its external parameters (problem dimensions, flags, data placement) and the hardware characteristics (interconnect, host/GPU memory capacity, GPU capabilities). Consequently, multi-GPU also requires optimization decisions based on the specific problem and system that can enhance performance and portability. Common such decisions are: communication routing to better utilize interconnect bandwidth, managing data-caching in GPUs, workload decisions to address imbalance, task scheduling and ordering to minimize I/O dependency blocking, and resource-related decisions to avoid unnecessary GPU usage. Moreover, due to the higher problem complexity of multi-GPU, the performance impact of these decisions is considerably higher than single-GPU. Unfortunately, the complexity of making all these decisions during execution is prohibitive. This forces libraries either to focus on a subset of these, implementing heuristics and fine-tuning them for each new system empirically, or employ generalized solutions, like task-graph optimization

or work-stealing [4, 54, 56–58, 68, 92, 157]. The first solution results in high performance for a subset of the total problems and for certain systems, but with very low portability leading to extreme performance meltdown in other configurations [9]. The second, on the other hand, is more generic and portable to new configurations, but lacks the performance of a specialized solution tuned for certain hardware and problem characteristics [8]. While this problem could also be solved with autotuning, the current state-of-the-art offers no such solutions due to the lack of modeling solutions for multi-GPU coupled with its very high complexity.

Takeaway Autotuning can solve the critical performance and portability issues of multi-GPU BLAS, but has been prohibitively complex to implement with current methods.

1.1 Problem Statement

Summing up, BLAS is very important for HPC applications and should be easily portable and performance-optimal in modern HPC clusters with GPUs. Unfortunately, while *in theory* BLAS routines are well-suited for single- and multi-GPU systems, *in practice* the additional communication, scheduling, and imbalance overheads introduced in these systems limit BLAS performance considerably and result in resource under-utilization. Moreover, the existing BLAS solutions cannot adapt dynamically to the underlying hardware architecture and workload, relying heavily on manual tuning and user input, resulting in low performance robustness and portability. Finally, while modeling and autotuning can considerably enhance performance, robustness and portability, the *hardware* complexity of modern GPU clusters coupled with the *algorithmic* complexity of GPU BLAS execution deem performance analysis and autotuning very hard.

1.1.1 Contributions

The goal of this thesis is to break free of these limitations and achieve portability, near-optimal performance, and efficient resource utilization for single- and multi-GPU BLAS. To this end, we choose a model-based approach where problem and system characteristics are parameterized and used to feed prediction models. Then, we apply these models in practice for BLAS autotuning by engineering an end-to-end library that tailors a variety of performance-effecting parameters to each individual BLAS call, taking into account its routine, problem and system-specific characteristics during runtime and adapting execution accordingly for each scenario. Consequently, the main contributions of our work are:

1. The introduction of new accurate performance models for communication-computation overlap, used for BLAS single-GPU offload performance estimation.

2. A methodology for abstracting interconnect characteristics to model BLAS communication in multi-GPU clusters and the extension of single-GPU performance models to multi-GPU.
3. An automated micro-benchmark and autotuning framework that enables the application of modeling for BLAS autotuning *in practice*.
4. PARALiA, a high-performance and resource-aware end-to-end multi-GPU BLAS library based on performance modeling and autotuning.
5. PARALiA-GEMMx, a state-of-the-art matrix-matrix multiplication implementation for multi-GPU clusters that relies on model-driven decomposition, scheduling, and communication optimization.

1.2 Outline

The remaining five chapters of this thesis are organized as following:

1. Chapter 2 focuses on the single-GPU execution scenario, exploring models and techniques for estimating single-GPU BLAS performance and providing a basis for BLAS autotuning, based on our published paper *CoCoPeLia* [7].
2. Chapter 3 describes the extension of these models to multi-GPU and heterogeneous scenarios, and introduces a model-assisted BLAS library that employs runtime autotuning to achieve high-performance and good resource utilization, based on our published paper *PARALiA* [9].
3. Chapter 4 then applies the aforementioned modeling and autotuning techniques specifically for the optimization of the matrix-matrix multiplication algorithm in multi-GPU clusters, and offers a model-driven matrix-matrix multiplication kernel that considerably outperforms the state-of-the-art approaches.
4. Chapter 5 provides a systematic view of all related work for this thesis, split in 1) BLAS optimization techniques (sec. 5.1) and 2) modeling approaches relevant to BLAS performance estimation (sec. 5.2).
5. Finally, chapter 6 describes the open questions and future work after this thesis and closes with our conclusions.

CHAPTER 2

Near-optimal single-GPU BLAS offload via model-based autotuning

Due to the high complexity of modeling multi-GPU BLAS execution, the first part of this thesis explores the simpler scenario of single-GPU BLAS. We use this scenario as a baseline before we expand to the more complex multi-GPU modeling problem in Chapter 3. Consequently, the targets of this chapter can be surmised in 1) obtaining *accurate performance models* for single-GPU BLAS *performance* and 2) enabling the application of these models in practice for *autotuning* important performance-effecting BLAS parameters at runtime. To this end, we introduce two performance models for GPU BLAS execution that, in addition to computation, also take into account data transfers and communication/computation overlap (Section 2.2) and an automated empirical methodology to instantiate these models on any system. Then, we combine these to an end-to-end GPU BLAS framework to enable automatic tiling size selection (Section 2.3) which demonstrates considerable performance improvement over similar state-of-the-art libraries (Section 2.4).

2.1 Problem formulation

The inherent parallelism of BLAS routines renders them particularly well-suited for GPU execution, hence the existence of numerous GPU-BLAS libraries. Among these, *cuBLAS* is the most

widely used BLAS library, providing highly optimized implementations of fundamental linear algebra operations tailored for NVIDIA GPUs [122]. However, execution with cuBLAS differs from the traditional CPU BLAS paradigm, since the input data must reside within the GPU memory space. This requirement presents an additional layer of complexity for programmers or domain scientists who use BLAS - they have to also account for transferring the data inputs/outputs to the GPU prior/after execution (henceforth *GPU offload*). Depending on the complexity of their scientific code, performing these additional data transfers can range from relatively straightforward, such as when solving a single algebraic equation, to extremely complex, like in the context of iterative solvers involving multiple BLAS routines with I/O dependencies. This diverges from the original design principles of BLAS, which aimed to provide a seamless and efficient computational framework regardless of problem and system characteristics.

As the popularity of GPUs increased rapidly, the problem of data transfers between the GPU and host memory emerged as a common challenge for GPU offloading [7, 21, 63, 103, 115, 116, 130, 134, 138]. In response to this, NVIDIA introduced the concept of unified memory in CUDA 6.0, which allowed the user to seemingly directly access host memory from the GPU, alleviating the need for explicit data transfers. However, despite the increase in programmability through unified memory, critical performance issues still exist, since data transfers still happen under the unified memory abstraction. More specifically, the communication between host and GPU memory is constrained by limited bandwidth, resulting in unavoidable transfer overheads and the unified memory abstraction also introduces additional performance bottlenecks [95,103,116]. This scenario poses a significant challenge for domain scientists who, by design, should not be burdened with these details. Consequently, their code often does not achieve the expected levels of the advertised GPU performance even for embarrassingly parallel problems.

A common approach to mitigate transfer overheads effectively involves the concurrent execution of host-to-device (h2d) and device-to-host (d2h) data transfers alongside computational tasks, a technique supported by OpeCL and CUDA and referred to as *3-way-concurrency* or *communication-computation overlap*. Figure 2.1 shows an example of this method for an arbitrary problem; the problem is split to smaller sub-problems (henceforth *sub-kernels*) with their own input/output dependencies. Then, these can run concurrently with other sub-problems, overlapping their communication and computation in a pipeline way and resulting in reduced *GPU offload* time.

2.1.1 State-of-the-art limitations

Tiling in GPU BLAS libraries: Since all BLAS routines operate on vectors and matrices, *domain decomposition* (e.g. splitting their workload to smaller chunks, creating sub-problems) is a common concept already used for various other BLAS optimizations. Similarly, data tiling and

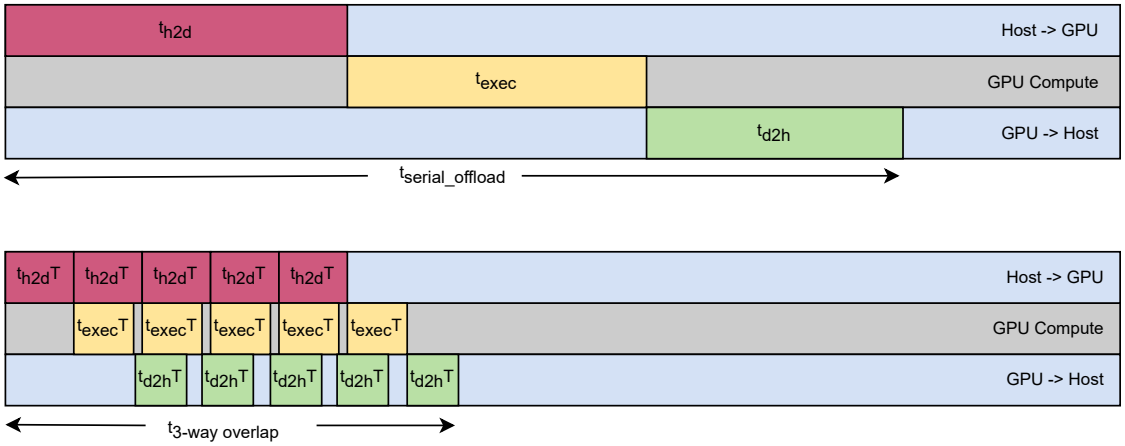
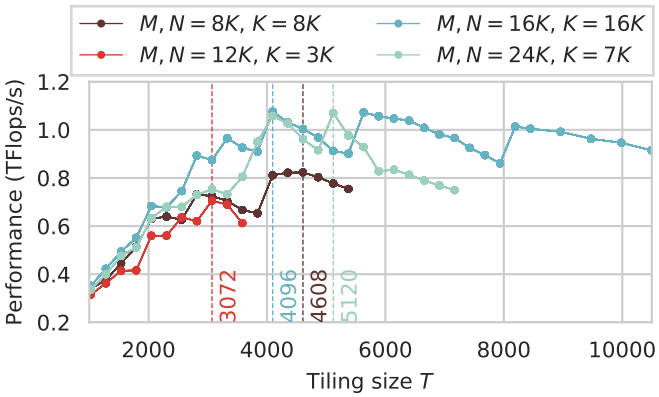


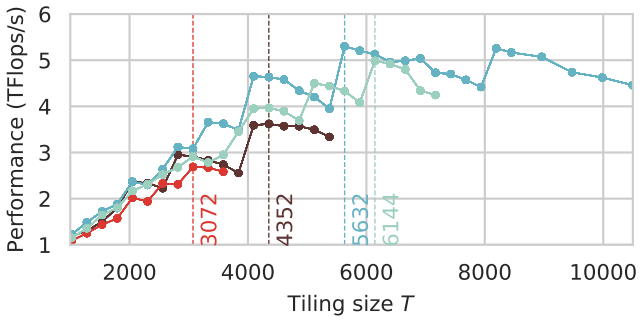
Figure 2.1: An example of offloading a computational problem to a GPU when the input/output data initially reside on the host memory. The default method is *serial offload* (top), where the input data are fetched, then the desired computations are performed in the GPU, and finally the output is sent back to the host. On the other hand, a better method is *3-way overlap* (bottom), where the initial problem dataset into smaller chunks of tiling size T, which are then transferred to the GPU in a pipeline-like way. This allows performing computations for one chunk while concurrently handling the output data transfer from the previous chunks and the input data transfer for the subsequent chunks, considerably decreasing the total time required for GPU execution.

3-way overlap for BLAS GPU offload have been the focus of many research approaches. Multiple BLAS libraries internally use tiling for better cache and memory utilization and to enable task parallelism [11, 56, 61, 68, 157, 161]. To achieve performance gain with 3-way-concurrency, GPU BLAS libraries internally split the initial problem size into tiles (more details are provided in Sections 2.2 and 2.3). In most cases, vectors (in level-1 and level-2 BLAS) are split to 1D chunks of length T and BLAS matrices (in level-2 and level-3 BLAS) to equal squares $T \times T$, where T is the *tiling size*.

Regardless of the internal tiling optimizations of each library, selecting the appropriate tiling size can considerably affect their resulting performance [21, 63, 65, 159]. GPU BLAS libraries follow two different strategies for tiling size selection, trading between programming ease and performance. The first is to expose the tiling size as an additional BLAS routine input parameter, leaving tuning to the user [57, 124, 125]. This directly contradicts the BLAS standard routine layout and limits the effective use of a BLAS routine to a handful of experts who can weigh the trade-offs of tiling size selection correctly. The second and most prevalent technique is to define and use a static tile size, usually defined per routine at compilation time, which provides a good average performance in the tested machines [11, 57, 58, 61, 68, 157, 161]. This is more user-friendly but requires engineering effort and empirical tuning by an expert in each new system, and results



(a) Testbed I - K40



(b) Testbed II - V100

Figure 2.2: `cuBLASXtDgemm` performance on two different testbeds, relative to the tiling size T used for internal 3-way overlap with $T \times T$ tiles, for four different problem sizes (no transpose). The vertical lines outline the tiling size that achieves the best performance for each problem.

in sub-optimal average performance since it disregards the specific communication/computation characteristics of each different problem to tiling size selection [159]. We argue that none of these approaches are generic enough or performance-optimal. As an example, Figure 2.2 illustrates the effect of T on performance for `cuBLASXtDgemm` on two testbeds. As the tiling size decreases, the performance increases due to better overlap, but after reaching one or two maxima, it rapidly degrades. These maxima "break-points" vary greatly across the two testbeds and problem sizes. Regarding the use of a static pre-defined tile, we annotate `dgemm` performance using the static tiling size of $T = 4096$, which offers the *best* average performance for *cuBLASXt* (details in Section 2.4), for the examples in Figure 2.2; it results in up to 9.4% slowdown on testbed I and up to 14.7% slowdown for testbed II. Furthermore, static tiling sizes offer no performance guarantee for future machines with different transfer bandwidth/computation ratios and can result in increased slowdowns in such cases.

Takeaway The impact of tiling size on BLAS performance makes a compelling case for dynamic tiling size selection and autotuning, driven by accurate performance models.

Performance prediction for tiling size selection: In order to achieve dynamic problem-specific tiling size selection, we require a prediction model that estimates the total GPU BLAS offload time as a function of tiling size, in order to evaluate different tiling sizes and find the best-performing one. Such a model does not exist in the literature, since GPU BLAS libraries consider the optimization problem too complex to model [157]. On the other hand, there is some research on performance modeling for GPU communication/computation not specifically for BLAS that is relevant to our case. A significant amount of prior work focuses on modeling the computation time or performance of GPU kernels [13, 35, 78, 81, 89, 109, 111, 113, 114, 127, 131, 136, 138, 165]. However, these models neglect offload time when data transfers are required before, after or during kernel execution. Gregg et al. [63] first highlighted this problem, arguing against the trend of excluding transfer overheads from scientific reporting of GPU application performance, and proposed a taxonomy for data transfers and their impact on offload performance. Numerous later works model CPU-GPU transfers, using variances of the linear latency-bandwidth model for PCIe transfers [21, 115, 134, 138].

When modeling communication and computation separately (e.g. for *serial offload*), including transfers improves prediction accuracy. However, if there is communication/computation overlap, simplistic models fail to predict the actual performance. As an example for the case of Figure 2.1, the total serial offload time $t_{serial\_offload}$ is equal to the sum of the input transfer, compute, and output transfer times, and can therefore be inferred simply by combining GPU compute and transfer models. On the other hand, modeling $t_{3-way\_overlap}$ is more complex since it requires an approach to estimate the *degree* of overlap of communication and computation, and also splitting the problem to smaller chunks also introduces latencies/overheads. To fill the gap of serial offload models, Gómez-Luna et al. [65] were the first to explore 3-way concurrency modeling, but they considered the stream creation time as the only overlap overhead. Later, Werkhoven et al. [159] enhanced their work by offering multiple performance models for communication/computation overlap for various common offload scenarios focusing on communication overlap latency, and provided methods to obtain the optimal number of CUDA streams for a given problem. Their models offered high accuracy for simple 1D problems, however, their modeling approach did not capture all problem characteristics present in BLAS (details in Section 2.2.2). Finally, Liu et al. [106] offered a mathematical framework for software pipelining (another term for communication/computation overlap) on GPUs, using non-equal tiles and focused on partitioning, scheduling and granularity. However, they targeted problems defined by linear functions and with equal input/output bytes, which does not apply to most BLAS routines.

Takeaway Previous communication-computation overlap models are not sufficient for GPU BLAS autotuning since 1) they are more generic, resulting in low accuracy for BLAS and 2) they are analytical, thus requiring engineering effort in order to be applied in practice for tiling size tuning.

2.1.2 Contributions

BLAS GPU libraries can benefit from dynamic *Tiling size* selection, but this requires both complex prediction models for 3-way concurrency overlap applicable to BLAS and a way to integrate these into an actual BLAS library. Consequently, in this chapter we explore single-GPU BLAS offload optimization and target the research gap between BLAS libraries and model-based autotuning that motivated our work CoCoPeLiA [7]. Overall, we make the following contributions:

1. Two new 3-way-concurrency analytical models for BLAS GPU offload time, one targeting level-1 and level-2 BLAS and one for more complex level-3 BLAS routines (Section 2.2).
2. An automated empirical methodology to instantiate these models on a system and three example models for `daxpy`, `dgemm`, and `sgemm` (Section 2.3).
3. The incorporation of the above with a runtime tile scheduler into *CoCoPeLia*, an end-to-end GPU BLAS framework utilizing automatic tiling size selection (Section 2.3), which demonstrates considerable performance improvement over similar state-of-the-art libraries (Section 2.4).

2.2 Modeling GPU BLAS offload

This section covers the core of our single-GPU offload research: the introduction of accurate 3-way concurrency prediction models applicable to BLAS problems. First, it discusses 3-way concurrency and the required adjustments in order to accurately model BLAS, taking into account 1) data location, 2) bidirectional overlap, 3) non-linear kernel execution times and 4) data reuse. It then builds upon a baseline model and presents our proposed adjustments in order to account for each of these characteristics, and concludes to two final models, one proposed for problems which employ little or no data reuse, like level-1 and level-2 BLAS, and one focused for optimized level-3 BLAS routines which fully utilize data reuse.

2.2.1 BLAS routine parameters

The BLAS specification standard defines a list of BLAS routines and their parameters [39, 40, 45, 47, 101]. BLAS usually have a considerable number of parameters to cover a wide range of

Table 2.1: BLAS modeling notation

$D1[, D2[, D3]$	routine problem size dimensions
$dtype$	routine datatype
k	the number of subproblems after tiling
k_{in}	the number of subproblems with input after tiling
T	tiling size
opd	number of input/output data structures
Per data structure:	$i : 0 \rightarrow opd$
get_i	flag to denote if data requires transferring from the host to the GPU
set_i	flag to denote if data requires transferring from the GPU to the host
$S1_i$, $S2_i$	initial dimensions, extracted from the routine problem size dimensions

configurations used within scientific code. The specification defines the formulas for calculating the problem dimensions for all parameter combinations and the exact mathematical operation performed. Some libraries also add extra parameters, usually related to featured optimizations or matrix layouts. For this work, we are only interested in the parameters defined by the standard and for LAPACK layout (column-major) storage of matrices. Throughout this work we use the GEMM routine as an example because it is the most widely used level-3 BLAS routine, and the other level-3 routines consist mostly of internal GEMM operations. The BLAS specification skeleton of GEMM is the following:

$$\overbrace{x}^{\textbf{dtype}} GEMM(\overbrace{TRANSA, TRANSB}^{\text{Flags}}, \overbrace{M, N, K}^{\textbf{Sizes}}, \overbrace{ALPHA}^{\text{Scalar}},$$
$$\underbrace{A,}_{\textbf{Data Ptr}} \underbrace{LDA,}_{\text{Ldim}} \underbrace{B,}_{\textbf{Data Ptr}} \underbrace{LDB}_{\text{Ldim}}, \underbrace{BETA,}_{\text{Scalar}} \underbrace{C,}_{\textbf{Data Ptr}} \underbrace{LDC}_{\text{Ldim}})$$

First, the BLAS standard defines strictly that the routine name must include the routine **datatype**. All other BLAS parameters can be split in 5 categories of interest; 1) flags/options, 2) sizes/dimensions, 3) scalars, 4) data pointers and 5) leading dimensions/increments [131]. From these, we are interested in the routine datatype and size, which effect the communication and computation volume of each problem, and the data pointers, which effect its communication volume and pattern.

Table 2.1 describes this notation which is used throughout this section, split in two categories; routine-specific (e.g. for a single `gemm` problem) and data specific (e.g. A,B,C - the matrices of

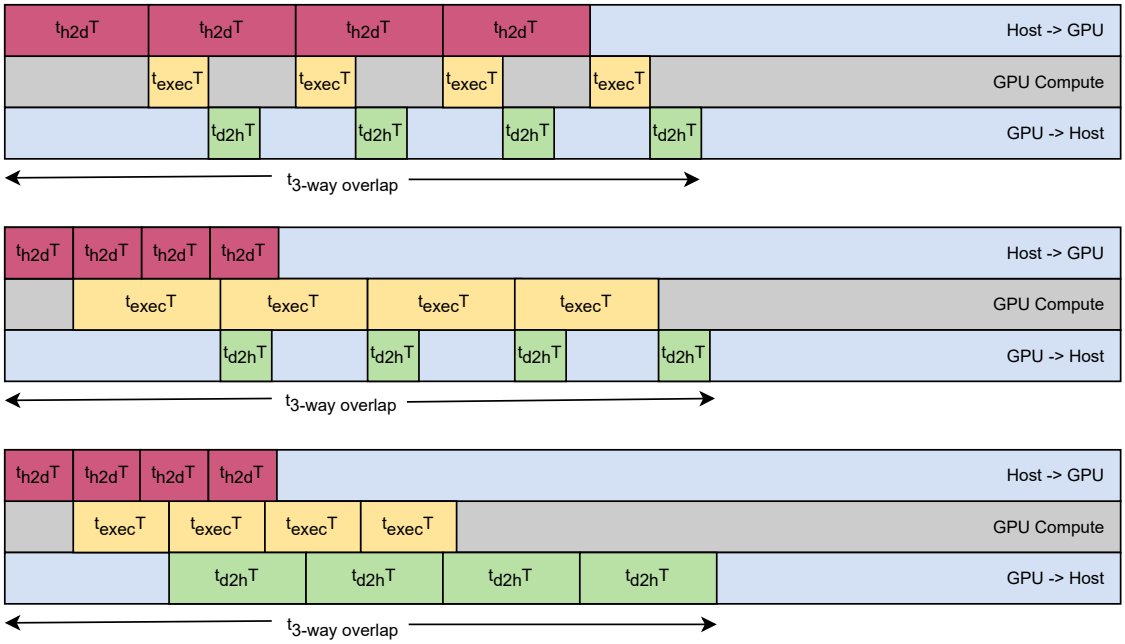


Figure 2.3: The forms a 3-way concurrency pipeline takes depending on the ratio of the h2d/d2h transfers and the execution time of a problem, for $k = 4$. The h2d-bound case (top) is dominated by input transfers, forcing computation and d2h transfers to block waiting for input, while the compute-bound (mid) is dominated by execution and the d2h-bound (bottom) by output d2h transfers.

the `gemm` routine) values. Certain parameters (e.g. opd, $dtype$) are inferred directly from the BLAS standard, others (e.g. $D1$, $D2$, $D3$) are problem-specific, while others are a combination of both (e.g. get_i, set_i, $S1_i$, $S2_i$). We provide details on obtaining or instantiating these parameters in Section 2.3. T is the optimization target parameter, namely the tiling size, and the formulas for k, k_{in} are defined later in this section.

2.2.2 GPU BLAS 3-way concurrency/overlap

3-way concurrency (or overlap) is a term referring to software pipelining, in order to overlap CPU-GPU communication with GPU computation, for any problem that can be split in parts without complex dependencies. Figure 2.3 shows an example pipeline using 3-way concurrency. The problem is split into k smaller parts (henceforth *sub-kernels*) based on a tiling size T, and each sub-kernel's completion requires 1) host-to-device (h2d) transfers for its input, 2) computation on the GPU and 3) device-to-host (d2h) transfers of its output. These 3 steps must be executed serially for each sub-kernel, but can be overlapped with the similar steps of previous and following sub-kernels, since they use different resources.

The most recent work on 3-way overlap modeling [159] distinguishes between three potential categories for any problem, outlined in the figure, and predicts total offload performance by calculating its total (non-overlapped) h2d time, its execution time and its d2h time and comparing them. Then, to account for the beginning and the end of the pipeline, which are non-overlapable, it adds to this time the duration of the first and last transfer, which depend on the selected tiling size T. Consequently, for the example of Figure 2.3, it assumes $t_{h2d} = t^T_{h2d} \times k$, $t_{exec} = t^T_{exec} \times k$ and $t_{d2h} = t^T_{d2h} \times k$, and calculates the final 3-way overlap time $t_{total} = \max(t_{h2d}, t_{exec}, t_{d2h}) + t_{non-over}$, where $t_{non-over}$ depends on which of the three sub-cases the problem belongs. Below we list the shortcomings of this approach that deem it inapplicable for BLAS tiling size prediction in practice.

Non-linear kernel execution times: By design, models that use the max of non-overlapped times to predict total time make the assumption that if a problem is split in k subproblems, and these are executed sequentially on the GPU, the total execution, h2d and d2h time will not change considerably ($t_{h2d} \approx t^T_{h2d} \times k$, $t_{exec} \approx t^T_{exec} \times k$ and $t_{d2h} \approx t^T_{d2h} \times k$). This does not hold for BLAS routines for three reasons. First, BLAS operations have internal dimensions and dependencies, which, in the case of tiling, may require additional reduction operations or lead to a change in the communication/computation ratio of the problem. Second, the performance of level-2/3 BLAS kernels does not depend linearly on their working set [16,131], since the problem shape (e.g. square vs fat-by-thin matrix multiplications) influences performance. Third, if a subproblem becomes too small, the GPU is underutilized and performance drops. Consequently, the previous models are good for predicting performance for small k decompositions, where sub-problems have similar properties with the original problem, but their accuracy degrades for larger k which introduce latencies that are unaccounted for.

Arbitrary data locations: While the input/compute/output volumes of BLAS routines as a function of their problem size is predefined, translating all input/output volume as transfers assumes that *all data is initially resident on the memory of the host CPU* (henceforth *full-offload*). This assumption does not hold if a kernel is executed iteratively, because some of the data may remain updated on the GPU between iterations, which is a very common scenario for BLAS [57]. Additionally, modern BLAS routines also allow mixed data configurations, where some of the input data reside on the CPU and the rest are already available on the GPU. Since previous overlap models [65, 106, 159] are analytical, the amount of communication volume (h2d/d2h transfer bytes) is a model input that must be defined by hand. This deems their application very hard to arbitrary BLAS input configurations, since the programmer must define all the potential data location combinations and their corresponding communication volumes, creating different models for each case. Instead, we want a model that also encompasses the location information to apply it to any BLAS problem without loss of accuracy.

Bidirectional overlap: 3-way concurrency goes beyond simple communication/computation overlap, considering also bidirectional host-device overlap (h2d with d2h transfers). While modern GPUs have virtually separate copy engines for h2d and d2h, both engines utilize the same communication medium and therefore simultaneous usage imposes a slowdown [106, 159]. This slowdown is asymmetric; usually the d2h transfers are more heavily affected, but the extent of this effect depends on the underlying interconnect and is therefore system-specific [130]. While this has been discussed in previous work, its hardware-specific effect on performance is hard to predict [159], so it has not been integrated in any overlap models. Consequently, overlap models have lower accuracy due to underestimating h2d and d2h transfer time, when these occur simultaneously.

Data reuse: Data reuse refers to the case when in a tiled problem, part of the data required for a subkernel's execution is also needed by subsequent subkernels. In level-3 BLAS, it can be present in all three dimensions of the problem. Unfortunately, while data-reuse is very important for performance, it is very hard to model accurately. This is because the reuse pattern and percentage are not effected only by the aforementioned problem and system parameters, but also by the max GPU memory and even by each specific implementation of the routine in question. For this reason state of the art BLAS 3 GPU libraries deploy efficient load balancing and execution runtime tile management, and not static prediction mechanisms [57, 157].

2.2.3 BLAS 3-way concurrency modeling

As shown in Figure 2.3, in order to enable 3-way concurrency for a problem it is split to k equal parts (sub-kernels). To calculate these parts, we split all problem dimensions, and use an equal tiling size T across them. This applies to level-1 BLAS, as well as level-2 and -3 BLAS square tiling, with the latter being the typical approach in BLAS GPU libraries [124, 157]. Thus, k is given by:

$$k = \frac{D1}{T}[\times\frac{D2}{T}][\times\frac{D3}{T}]$$

where $D2$ ($D2$ and $D3$) applies to level-2 (level-3) BLAS.

Then, modeling 3-way concurrency requires knowledge of the time required for each of the overlapped parts (*h2d*, *execution* and *d2h*). To that end, we consider that the total time t_{total} is a function of k and the individual times for h2d transfer t^T_{h2d}, kernel execution t^T_{exec}, and d2h transfer t^T_{d2h}, for each sub-kernel generated for tiling size T. Since we want to also account for latencies/overheads from splitting the problem to sub-problems, we do not calculate t^T_{h2d}, t^T_{exec} and t^T_{d2h} from the transfer and execution times of the original problem (e.g. $t^T_{h2d} \neq \frac{t_{h2d}}{k}$, $t^T_{exec} \neq \frac{t_{exec}}{k}$ and $t^T_{d2h} \neq \frac{t_{d2h}}{k}$). Instead, we consider the following sub-models that contribute to

the total offload time:

$$t^T_{h2d} = f_1(system, dtype, T[, T])$$
$$t^T_{d2h} = f_2(system, dtype, T[, T])$$
$$t^T_{exec} = f_3(routine, dtype, T[, T[, T]])$$

where t^T_{h2d}, t^T_{d2h} are the system-wide transfer times for a single tile of size T (if it is a vector) or $T \times T$ (if it is a matrix), and t^T_{exec} is the BLAS routine-specific execution time for a kernel where $D1[= D2[= D3]] = T$. These times are empirically collected in the CoCoPeLia framework (more in Section 2.3).

We assume that in the 3-way-concurrency scenario, each subkernel execution on the GPU is overlapped with 1) the subsequent subkernel input and 2) the previous subkernel output, in a pipelined manner [65,159]. Under the assumption that all data initially reside on the CPU and are both input and output data, the 3-way-concurrency execution time for a BLAS routine is then:

$$t^{baseline}_{total} = max(t^T_{exec}, opd \cdot t^T_{h2d}, opd \cdot t^T_{d2h}) \times (k-1) + opd \cdot t^T_{h2d} + t^T_{exec} + opd \cdot t^T_{d2h} \quad (2.1)$$

Where the max part of the equation estimates the overlap-able part and the remaining equation the start/end of the pipeline that cannot be overlapped. As an example for Figure 2.3 (where $opd = 1$), the max term corresponds to the part dominating the pipeline time in each case (t^T_{h2d} (top), t^T_{exec} (mid), t^T_{d2h} (bottom)) and the remaining part to a single sub-kernel's t^T_{h2d}, t^T_{exec} and t^T_{d2h} that are not overlapped due to dependencies.

2.2.3.1 Data Location Modeling

In practice, Equation 2.1 overestimates transfers to and from the GPU; by including the opd multiplier, we assume that all data is both input and output data and therefore must be transferred, and that all data is initially on the CPU. To avoid this, we define the get_i, set_i flags, which determine which of the opd tiles require to be fetched to the GPU or returned to the host. These values are extracted from the BLAS routine and the data pointers for each input/output vector/matrix during runtime. Then, we define t^T_{in} and t^T_{out} as the time required to transfer all tiles for which $get_i = 1$ and $set_i = 1$, respectively, as follows:

$$t^T_{in} = \sum_{i=0}^{opd} get_i \cdot t^T_{h2d} \quad \text{and} \quad t^T_{out} = \sum_{i=0}^{opd} set_i \cdot t^T_{d2h}$$

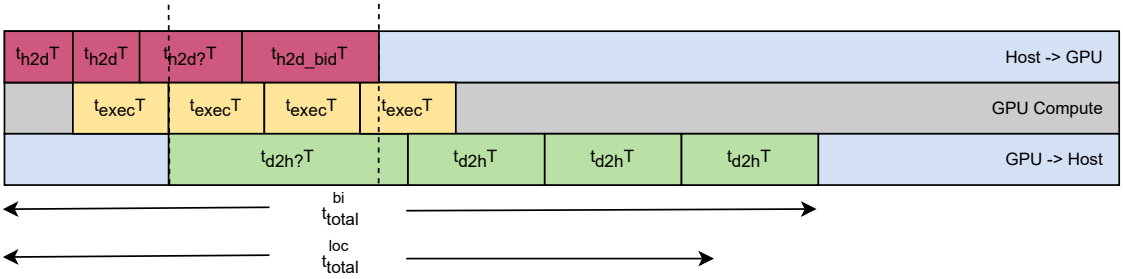


Figure 2.4: The effect of bidirectional slowdown for the d2h-bound problem of Figure 2.3. When performing h2d and d2h transfers simultaneously in the pipeline (dotted lines), both t^T_{h2d} and t^T_{d2h} increase, but due to the partial overlap of some transfers (denoted as $t^T_{\{h2d,d2h\}?}$), predicting the exact time for this area is not straightforward. This results in a total time underestimation when using the t^{loc}_{total} model for h2d/d2h-bound problems.

Following the notion of Equation 2.1, the location-aware execution time of a BLAS problem using 3-way concurrency is:

$$t^{loc}_{total} = max(t^T_{exec}, t^T_{in}, t^T_{out}) \times (k-1) + t^T_{in} + t^T_{exec} + t^T_{out} \tag{2.2}$$

2.2.3.2 Bidirectional Slowdown Modeling

Figure 2.4 shows the d2h-bound example of Figure 2.3 in a more realistic scenario where simultaneous h2d and d2h transfers impose a slowdown on both sides. To account for this phenomenon in our models, we first need an estimation for the bidirectional transfer time $t^T_{\{h2d,d2h\}\_bid}$ of a h2d or d2h transfer, when the opposite link is also in use. To this end, we define the slowdown factors $sl_{h2d\_bid}, sl_{d2h\_bid}$ for each direction as scaling factors applied to transfer time, in the case the opposite direction is also in use for the whole duration of the transfer. We estimate the slowdown factors empirically in the CoCoPeLia framework (Section 2.3). $t^T_{\{h2d,d2h\}\_bid}$ then become:

$$t^T_{\{h2d,d2h\}\_bid} = sl_{\{h2d,d2h\}\_bid} \cdot t^T_{\{h2d,d2h\}}$$

However, full bidirectional overlap only applies in practice if $t^T_{h2d} = t^T_{d2h}$. In a common case, two simultaneous opposite transfers have different duration, and the total overlap time t^T_{over} is split in two parts; 1) the part during which actual overlap occurs and 2) the single-way transfer of the remaining partially-complete transfer:

$$t^T_{over} = \begin{cases} t^T_{d2h\_bid} + \frac{t^T_{h2d\_bid} - t^T_{d2h\_bid}}{sl_{h2d\_bid}}, & \text{if } t^T_{h2d\_bid} \geq t^T_{d2h\_bid} \\ t^T_{h2d\_bid} + \frac{t^T_{d2h\_bid} - t^T_{h2d\_bid}}{sl_{d2h\_bid}}, & \text{otherwise} \end{cases} \tag{2.3}$$

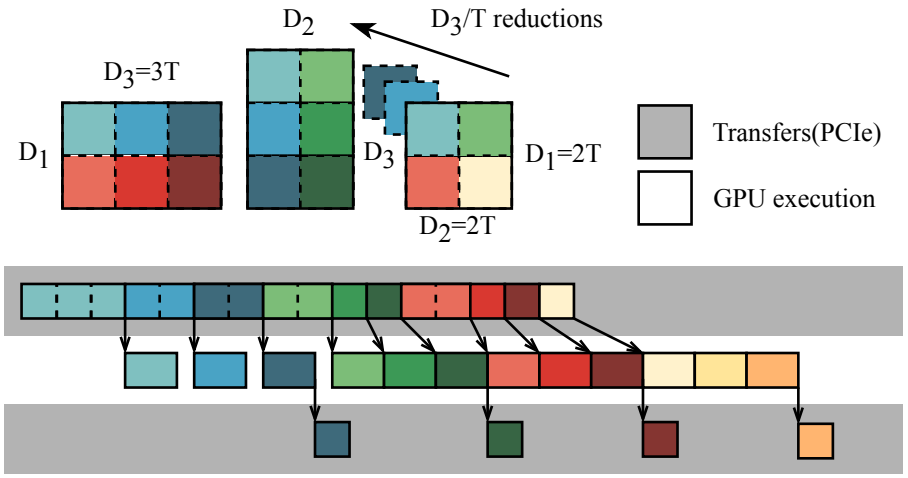


Figure 2.5: An example of the 3-way concurrency pipeline for a `gemm` implementation which iterates through the tiles of the M, N, K dimensions, for a problem with $t^T_{h2d} < t^T_{exec} < 2 \cdot t^T_{h2d}$. Data reuse results in roughly two areas; one where the problem is h2d bound, and one where its execution bound, a scenario previous approaches cannot account for.

The fraction in the equation corresponds to the time required to transfer the remaining part of the longer transfer. Equation 2.2 therefore evolves to account for bidirectional overlap as follows:

$$t^{bi}_{total} = max(t^T_{exec}, t^T_{over}) \times (k-1) + t^T_{in} + t^T_{exec} + t^T_{out} \tag{2.4}$$

2.2.3.3 Data Reuse Modeling

All previous models are not accurate for optimized level-3 BLAS problems, as they do not account for data reuse. Reuse exists in both level-2 BLAS (vector reuse) and level-3 BLAS (matrix reuse), but is mostly relevant to level-3 BLAS performance. Figure 2.5 shows an example of a level-3 BLAS routine with data reuse. Initially, the problem is transfer-bound. Then, h2d transfers decrease due to data reuse, and the problem becomes execution-bound. The example refers to a specific t^T_{h2d}, t^T_{exec} ratio, and the amount of reuse and this ratio can significantly alter performance. We construct a generic model, for the ideal reuse case, namely full reuse, where all available tile reuse potential is utilized.

Given the tiling size T, we can compute how many tiles an initial matrix i of dimensions $S1_i$, $S2_i$ is split into, as follows:

$$tiles_i = \frac{S1_i}{T} \cdot \frac{S2_i}{T}, i : 0 \rightarrow opd$$

In level-3 BLAS, we opt to account for the transfer of tiles only once, assuming that they then become available for all subsequent subkernels that use them. In reality, this creates four

subkernel categories; 1) subkernels that require three input tiles 2) subkernels that require two input tiles, 3) subkernels requiring a single tile and 4) subkernels which have all input available by previous subkernel transfers. Previous models and Equation 2.1, 2.2, 2.4 do not distinguish between these categories; the first two assume that the data volume is equally distributed among kernels because of their top-down approach, and the later assumes all subkernels belong in the first category. In reality the first category contains only the first subkernel, since afterwards at least one of its tiles will be used by subsequent invocations, and the transfers for this subkernel are already accounted for separately because they can't be overlapped. Additionally, the number of subkernels belonging in the second and the third category are very hard to split, since they depend on the specific tile distribution algorithm. We therefore avoid splitting them and categorize them together in order not to complicate the model. Consequently, we compute the number of subkernels among the k total subkernels that require one or two tile transfers, as follows:

$$k_{in} = \sum_{i=0}^{opd} (get_i \cdot tiles_i - 1)$$

For a more optimized implementation, the larger percentage of k_{in} collapses to single tile transfers. We follow this assumption for our final 3-way-concurrency offload time model with reuse, which is given by the following model:

$$t^{re}_{total} = max(t^T_{h2d}, t^T_{exec}) \cdot k_{in} + t^T_{exec} \cdot (k - k_{in}) + t^T_{in} + t^T_{out} \tag{2.5}$$

2.2.4 Model application per BLAS level

Due to the difference in input types and communication/computation ratios, different BLAS levels have different model requirements. For the rest of this work, we apply the introduced 3-way concurency models to different BLAS levels as follows:

Level-1 BLAS routines perform vector-vector operations and their working set is $D1 = N$. These transfer-bound routines have no working set overlaps and are therefore modeled effectively by Equation 2.4.

Level-2 BLAS routines perform matrix-vector operations and have two problem dimensions $D1, D2$ where $D1$ is the output vector length and $D2$ is the remaining dimension of the multiplied matrix. While there is a minor working set overlap among sub-kernels for the vector, it is relatively small ($D1$) compared to the matrix dimensions ($D1 \times D2$), and therefore Equation 2.4 is still sufficient for modeling them.

Level-3 BLAS routines perform matrix-matrix operations and have 3 problem dimensions $D1, D2$ and $D3$ where $D1, D2$ are the output matrix dimensions and $D3$ is the internal matrix multiplication dimension. Splitting $D3$ results in $\frac{D3}{T}$ reductions of size $D1 \cdot D2$, which increase the

problem computations, but benefit tiling and transfers. Although Equation 2.4 can provide an estimate for the offload time of many applications, optimized implementations that employ tiling and data reuse lead to higher performance. Since our focus is tiling size selection for state of the art performance, we devise Equation 2.5 to account for data reuse in level-3 BLAS.

2.3 Runtime framework integration

After presenting 3-way concurrency prediction models for BLAS routines on GPUs, we now turn our attention on how to utilize them in practice. Transitioning from analytical modeling of 3-way concurrency to the integration of tile autotuning into a practical, end-to-end implementation involves numerous challenges. The first is that BLAS parameters only become available at routine invocation, which directly forces a runtime-autotuning approach. This in turn requires that the autotuning process should be fairly lightweight, to avoid autotuning overheads pouring into BLAS end-to-end performance. Then, our 3-way concurrency models also require some sub-models and/or empirical measurements that differ per system and routine. Finally, all the autotuning process must be integrated with state-of-the-art scheduling and execution mechanisms, to avoid implementation-based latency from effecting total performance.

Taking all these into account, we present the **CoCoPeLia** framework, which handles all the necessary steps including the automatic instantiation of the model for a specific machine and the development of a proof-of-concept library that utilizes the model itself at runtime. Figure 2.6 shows the complete CoCoPeLia framework. At the heart of the framework lies the Tile selection runtime which employs the prediction models described in detail in Section 2.2. The deployment module feeds the runtime with the proper transfer and execution sub-models (predictors for t^T_{h2d}, t^T_{d2h} and t^T_{exec}, sl_{h2d_bid}, sl_{d2h_bid} – details in Section 2.3.1) while the library implements an optimized subset of the BLAS prototype on top of basic GPU BLAS kernels (details in Section 2.3.3). During application execution, when a BLAS routine with a specific set of parameters is invoked for the first time, the CoCoPeLia model is consulted in order to pick the best tiling size.

2.3.1 Deployment: Empirical initialization of model coefficients

To instantiate the models of Section 2.2, we first model the *transfer time* of the target system, with a semi-empirical approach. We perform a set of micro-benchmarks offline and use them to fit the coefficients of basic linear models for transfer time. We use the well-accepted latency/bandwidth model [16, 21, 65, 106, 159], which estimates transfer time as a function of *bytes*. In our case, the latency/bandwidth model for host-to-device transfers takes the form:

$$t^T_{h2d} = t_i + t_b \cdot (\overbrace{T[\cdot T] \cdot sizeof(dtype)}^{\textbf{bytes}})$$

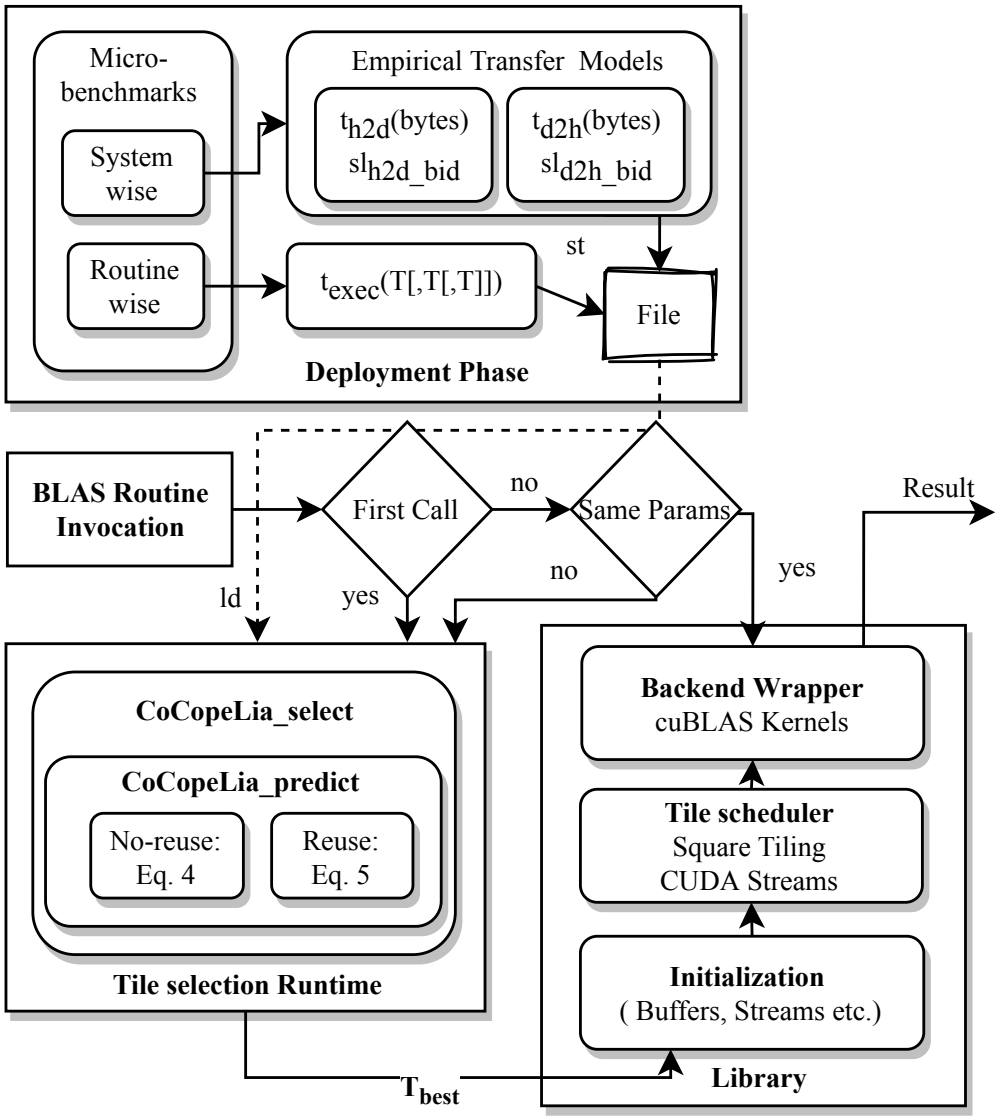


Figure 2.6: The CoCoPeLia framework pipeline. During the offline deployment phase the framework performs micro-benchmarks. Then, when a BLAS routine is invoked with some problem parameters for the first time, the tile selection runtime uses them in conjugation with the values obtained during deployment to predict the best tiling size T for this problem. Finally the library is invoked to perform the operation for the given T_{best} and produce the routine result. In case the routine has been called with the same problem parameters before, all unnecessary steps are skipped and the previous tiling scheme and T_{best} is reused.

As discussed, we assume that bidirectional overlap imposes a constant slowdown (sl) to transfer, and therefore the bidirectional transfer time can be estimated by:

$$t_{h2d\_bid} = sl_{h2d} \times t_{h2d}$$

Table 2.2: Transfer sub-models for the two testbeds.

System						
	t_i	$1/t_b$	RSE	$1/t_b$ bid.	RSE bid.	sl
Testbed I (Nvidia Tesla K40)						
$h2d$	$2.4e^{-6}$	$3.15e^{9}$	$1.1e^{-6}$	$2.94e^{9}$	$2.7e^{-6}$	1.07
$d2h$	$2.2e^{-6}$	$3.29e^{9}$	$2.1e^{-6}$	$2.84e^{9}$	$3.4e^{-6}$	1.16
Testbed II (Nvidia Tesla V100)						
$h2d$	$2.5e^{-6}$	$12.18e^{9}$	$1.7e^{-6}$	$9.59e^{9}$	$3.4e^{-6}$	1.27
$d2h$	$2.5e^{-6}$	$12.98e^{9}$	$2.8e^{-6}$	$9.21e^{9}$	$4.2e^{-6}$	1.41

Similarly, t^{T}_{d2h}, t_{d2h_bid} are modeled with the same equations. Therefore, the system-wise transfer parameters required for prediction are t_i, t_b and sl for h2d and d2h (six in total). To fit these coefficients, we conduct a set of micro-benchmarks, subset of those proposed by Pearson [130]. For all transfer experiments, we use the `cublas{Set/Get}MatrixAsync` routines for h2d, d2h transfers respectively, with pinned host memory, as required by these asynchronous calls. We obtain t_i empirically as the average latency of multiple single-byte transfers. For t_b, we run benchmarks for square transfers with $dtype = double$, for $D1 = D2 = 256 \xrightarrow{\text{step=256}} max\_device\_memory/2$, and follow the same approach for sl, but couple the entire transfer with a concurrent transfer towards the opposite direction. We use least square regressions on the 64 samples to compute t_b, both in the case of uni-directional and in the case of bi-directional transfers, excluding t_i from the transfer time during the regression (assuming zero intercept), in the manner of [75], and then estimate sl. We ensure the statistical robustness of the empirical values by collecting repetitive measurements, until the 95% confidence interval of the mean falls within 5% of the reported mean value, for all micro-benchmarks. The micro-benchmarks for transfer times are lightweight (requiring less than 10 minutes and less than 3 minutes on Testbed I and II, respectively), and only need to be run once on every new system CoCoPeLia is deployed. They can either be performed automatically when a BLAS GPU library is installed in a system, or by hand when a domain scientist plans to port his application which uses said library to a new system.

Table 2.2 contains the obtained values for the two testbeds used in this work, described later in Section 2.4. The displayed $1/t_b$ is equal to each system's PCIe bandwidth for each direction. Testbed II has almost $3\times$ higher bandwidth than testbed I, but also has much larger bidirectional slowdowns sl for both directions, indicating that overlapping h2d and d2h transfers is not going to be as effective. The least square regression coefficient p-values for t_b and sl_{bid} are $< 2.2e-16$, and the RSEs are between 1 and $4.2e^{-6}$, which is comparable to t_i, which means the model is not be very accurate in describing transfer latency. This might pose a problem in modeling very

small transfers, but since the T size maxima reside far above those areas, this is acceptable for our approach.

Second, we estimate routine *GPU execution time*. We are only interested in the time of fine-grained chunks of specific small tiling sizes, therefore we measure the execution time for a set of tiling sizes T for each routine, store them and perform value lookups at runtime, for usage in our models. The usage of empirical estimates is favored by the tiled execution and the non-linear execution time assumption in CoCoPeLia, since micro-benchmarks for these chunks are much more lightweight than an approach that would require the full problem's execution time, as in [159]. For example, empirically estimating the execution time for a `gemm` problem of size $M = N = K = 32K$ using $T = 2048$ would require $t_{exex\_routine}(M, N, K)$ for [159], while for our case $t_{exex\_routine}(T, T, T)$, which requires 4096 times less computations, would be sufficient. To measure the GPU BLAS execution time, we use cuBLAS, but given that kernel execution is wrapped and all libraries follow the BLAS standard, this benchmarking method is applicable to any BLAS GPU library with minimal adjustments.

We choose three representative routines; `axpy` for 1D splitting and `gemm` for single and double precision for square tiling. For `daxpy`, we run 256 benchmarks for $D1 = N = 2^{18} \xrightarrow{\text{step}=2^{18}} 2^{26}$. For `dgemm` and `sgemm`, we use 64 benchmarks with square dimensions $D1 = D2 = D3 = 256 \xrightarrow{\text{step}=256} 16384$ for value lookup of tiled sub-kernels. Therefore, the sub-model t^T_{exec} can only predict time for these 380 and 64 tile sizes (via direct value lookup) for `daxpy` and `gemm` respectively. We repeat the micro-benchmarks, until the 95% confidence interval of the mean falls within 5% of the reported mean value. The required time for the benchmark execution is less than 6 minutes for each routine on Testbed I, and less than 2 minutes on Testbed II.

CoCoPeLia automates the micro-benchmark execution on any new system without modifications. Upper limits for the required benchmarks are extracted based on the available GPU memory. Additionally, CoCoPeLia can be easily extended for any BLAS routine by modifying the existing micro-benchmark template scripts with the new routine and its parameters.

2.3.2 Tile selection runtime: Tiling size autotuning

The CoCoPeLia runtime includes two functions for tile selection. The function `CoCoPeLia_predict` combines the empirical values obtained at deployment with the problem-specific parameters used in the BLAS routine invocation listed in Table 2.1, to provide the execution time of a BLAS routine, as a function of the tiling size T, using Equation 2.4 and 2.5 for cases without and with data reuse. The problem dimensions $D1[, D2[, D3]]$ are inferred from the BLAS dimensions M, N, K, and $S1_i, S2_i$ are then calculated based on the above. Finally, get_i, set_i are obtained by querying the pointers of the respective $i-th$ data structure (e.g. matrix, vector) using `cudaPointerGetAttributes`. The function `CoCoPeLia_select` is used to

provide the best tiling size T for a specific problem, using the `CoCoPeLia_predict` routine to find T_{best}, which minimizes the total offload time, by iterating through all sizes T obtained at deployment for the target routine. The function can be extended to include different optimization criteria (e.g. GPU utilization, memory etc.). We have measured model initialization to take 2-3 ms and prediction time to be negligible (less than 100 μs).

The extension of the CoCoPeLia Tile selection runtime with additional BLAS routines, besides the micro-benchmarks explained in 2.3.1, requires the following modifications: i) the extension of a skeleton for a `CoCoPeLia_{routine}_init` function, that matches the routine's parameters to the struct with the model parameters of table 2.1, and ii) the selection of a `CoCoPeLia_predict_{ModelName}` function for Tile prediction of this routine. The extension of CoCoPeLia with new prediction models is possible by defining a new `CoCoPeLia_predict_{ModelName}`. However, if any additional parameters are required, the struct of table 2.1 must be also modified accordingly.

2.3.3 Library: Task orchestration

While selecting an appropriate tiling size T should suffice for a 3-way concurrency optimized library to achieve near-optimal performance, existing libraries do not optimize level-1 and level-2 BLAS routines, while *cuBLASXt* and *BLASX* often result in less performance than what Equation 2.5 hints. To validate the accuracy of the proposed data-reuse model and fill this performance gap, as a part of CoCoPeLia, we implement an optimized end-to-end library for a subset of the BLAS prototype on top of state-of-the art primitive libraries.

We use cuBLAS as the GPU execution and data transfer backend, utilizing `cublas{Dtype}{Routine}` and `cublas{Set/Get}MatrixAsync` routines respectively. For 3-way concurrency, we use CUDA streams, utilizing one stream per operation (h2d transfer, d2h transfer, kernel execution). The tile splitting, address matching and distribution (tile scheduler) are implemented based on the square tiling approach (as implied in Eq 2.5), also used by [124, 157]. After calculating these, the tile scheduler hands over all underlying transfers and execution to the aforementioned cuBLAS calls. Additionally, we enable GPU buffer and CUDA stream reuse after the first routine call, to avoid allocation/de-allocation overheads, as proposed by *BLASX* [157] to emulate an iterative use-case scenario. Finally, CoCoPeLia routines support either passing the tiling size T as an extra BLAS parameter, similar to *cuBLASXt*, for validation reasons, or using `CoCoPeLia_select` internally to predict T_{best} during invocation. CoCoPeLia routines also take advantage of model reuse in the second case; they initialize the corresponding model only the first time a user makes a call to CoCoPeLia with a set of parameters (routine, problem size, flags, etc) and use the preobtained T_{best} in subsequent calls. The tile scheduler is generalized per BLAS-level. To add a new BLAS routine that utilizes the tile scheduler requires the creation

Table 2.3: Testbed characteristics

	Testbed I	**Testbed II**
CPU	Intel Core i7-4820K	Intel Xeon Silver 4114
	3.7GHz	2.2GHz
GPU	NVIDIA Tesla K40	NVIDIA Tesla V100
	FP Peak 4.3 TFlop/s	FP Peak 14 TFlop/s
	DP Peak 1.4 TFlop/s	DP Peak 7 TFlop/s
PCIe	Gen2 x8	Gen3 x16
Compiler (host)	g++ 4.7.2	g++ 7.5.0
Compiler flags	-O3, -lm, -std=gnu99	-O3, -lm, -std=gnu99
CUDA	9.2	9.2
Compiler flags (GPU)	-O3, -arch=sm_35	-O3, -arch=sm_75

of a routine wrapper, since the specifics of each BLAS operation differ, but transfer/ execution overlap is then achieved by the tile scheduler without modifications. The underlying backend functions are wrapped and can also be modified, as long as an overlap mechanism similar to CUDA streams is available.

2.4 Experimental evaluation

In this section, we evaluate the CoCoPeLia framework for three example kernels; `daxpy`, `sgemm`, and `dgemm`. First, we present our experimental setup, and describe the micro-benchmark and validation sets we used. Then, we validate the proposed 3-way concurrency time prediction model error and the ability of CoCoPeLia to select near-optimal tile sizes. Finally, we present the end-to-end performance achieved with CoCoPeLia using our own 3-way concurrency implementation coupled with automatic tile selection, and compare it with the state of the art.

2.4.1 Experimental setup

We perform experiments for the validation of our models and the evaluation of CoCoPeLia on two different testbeds, the details of which are presented in Table 2.3, along with the information on code compilation. For time measurements we use `clock_gettime`, with device synchronization (`cudaDeviceSynchronize()`) also included; both timer and synchronization overhead were less than 1% of the benchmarked times. We perform 100 executions for each benchmark, after a warmup run, not accounted for, and we report the average time for all models, unless otherwise noted. 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 Async CUDA calls and the cashes/buffers are not flushed between runs.

2.4.2 Validation sets

To validate different initial memory locations and problem sizes, we select four large problem sizes ($N = \{8, 64, 128, 256\} \cdot 2^{20}$) for `daxpy`, for all $2^2 - 1 = 3$ location combinations (15 problems). Similarly, for `sgemm` and `dgemm` we want to validate different locations, problem sizes and shapes. We use four square problem sizes $M = N = K = \{4, 8, 12, 16\} \cdot 2^{10}$, for all $2^3 - 1 = 7$ location combinations (28 problems) to validate the location-problem size, and 3 problem sizes with $M \cdot N \cdot K = \{4, 8, 12, 16\} \cdot 2^{10 \cdot 3}$ for 3 fat-by-thin ratios $M = N = K \cdot \frac{r^3}{8}, r \in [3, 4, 5]$ and 3 thin-by-fat ratios $M = N = K \cdot \frac{8}{r^3}, r \in [3, 4, 5]$ for the scenario of all data initially residing on the CPU (24 problems). We exclude the scenario where all data is located on the GPU, since there is no overlap. All selected problem sizes can fit in the device memory; we do not consider larger problem sizes since that would require a considerably more sophisticated implementation of overlap with memory constraints, which is outside the scope of this work. For each problem size, we measure the execution time t of 1) the CoCoPeLia wrapper and 2) *cuBLASXt*, for all tile sizes $T = 1024 \xrightarrow{\text{step=256}} 16384$ for which $T \leq \frac{min(D1[,D2,D3])}{1.5}$.

2.4.3 Time prediction validation

We first focus on validating the prediction ability of our bidirectional transfer overlap-aware model of Equation 2.4, hereafter referred to as *BTS-Model*, and our data reuse-aware model of Equation 2.5, hereafter referred to as *DR-Model*, used in $t_{over} =$ `CoCoPeLia_predict` of Figure 2.6. We examine their error over measured execution time and compare their predictive power against the analytical CUDA stream overlap model with two copy engines, proposed in [159], hereafter referred to as *CSO-Model*. We evaluate the percentage (relative) error $e_{\%} = 100 \cdot (t_{predicted} - t_{measured})/t_{measured}$. We highlight that both models include empirical parts, which impose second order errors. Nonetheless, the comparison between different models is fair, as we rely on the same micro-benchmarks to collect the empirical values. In our observations bellow we take into account the bias this approach imposes on negative errors.

We first validate the prediction accuracy of the *BTS-Model*, which is suitable for problems without data reuse between subkernels, using the level-1 BLAS `daxpy`, which does not reuse data, and *cuBLASXt* `sgemm` and `dgemm`, which do not sufficiently utilize data reuse to minimize transfers [57, 157]. Figure 2.7 shows the relative error distribution for `daxpy`, `sgemm` and `dgemm`, for both testbeds, in the form of violinplots. First, we note that, on both testbeds, the *BTS-Model* achieves very high prediction accuracy for `daxpy` with median errors between 1 to 2%, while the *CSO-Model* underpredicts execution time with median errors between -3% to -7%. This is attributed to the *CSO-Model* not accurately modeling the actual bidirectional overlap, and is more evident on Testbed-II, where the slowdown is larger on both directions. Second, for both `sgemm` and `dgemm`, the prediction error is higher. The *CSO-Model* again significantly

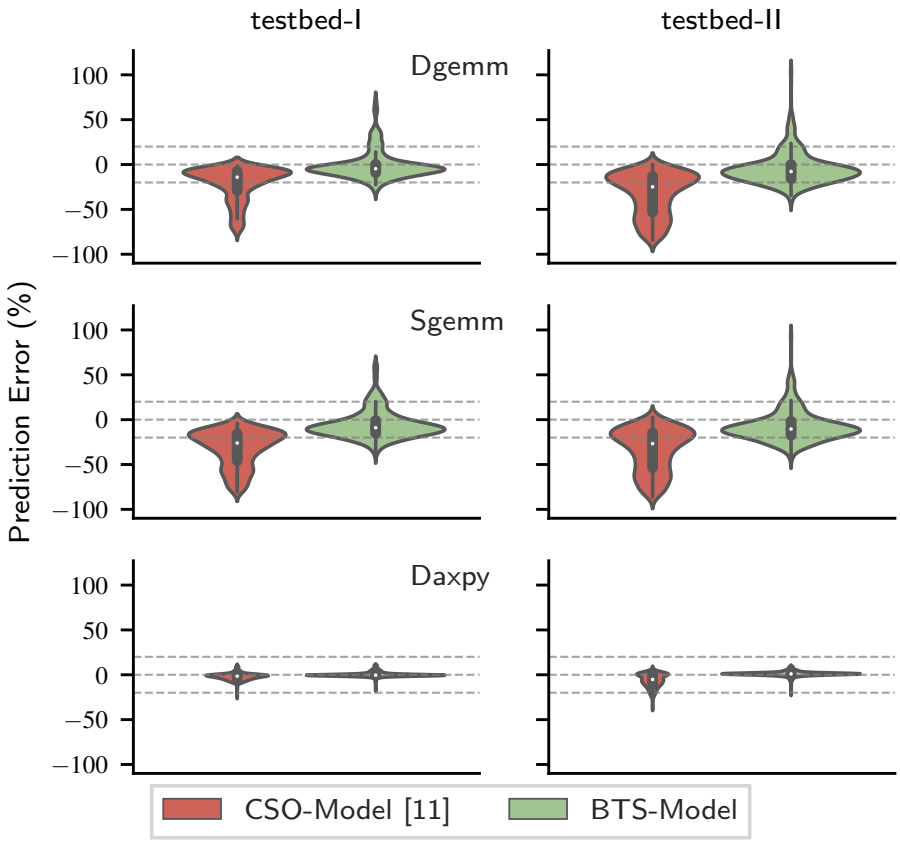


Figure 2.7: Error distribution of the *CSO-Model* and the *BTS-Model* for `daxpy` and `cublasXt_{D,S}gemm` without data reuse on testbeds I and II.

underestimates the execution time in almost all cases, with median errors between -20% to -34%. On the other hand, the *BTS-Model* demonstrates smaller median errors between -10% to -15% and a better error distribution with no bias towards underprediction.

We then validate the prediction accuracy of the *DR-Model*, using our aforementioned implementations for `sgemm` and `dgemm`, in the CoCoPeLia library, which have near-optimal data reuse on single-GPU scenarios, when the problem fits the GPU memory. Figure 2.8 demonstrates the relative error distribution on both testbeds. The *CSO-Model* underestimates execution times, similarly to the *cuBLASXt* case in Figure 2.7, however with fewer underestimations with errors ranging from -20% to -60% and a lower median error of -7% to -15%. Again, our *DR-Model* is significantly more accurate, with median errors ranging from 2% to -5%, and a few high positive errors (overestimations). It is interesting to note that both models exhibit higher errors for `sgemm`, where the memory footprint is half than the equivalent of `dgemm`, and smaller problems are more prone to second order errors from the empirical value acquisition. Additionally, our *DR-Model* is more accurate for Testbed I, than Testbed II. This is due to spikes in performance

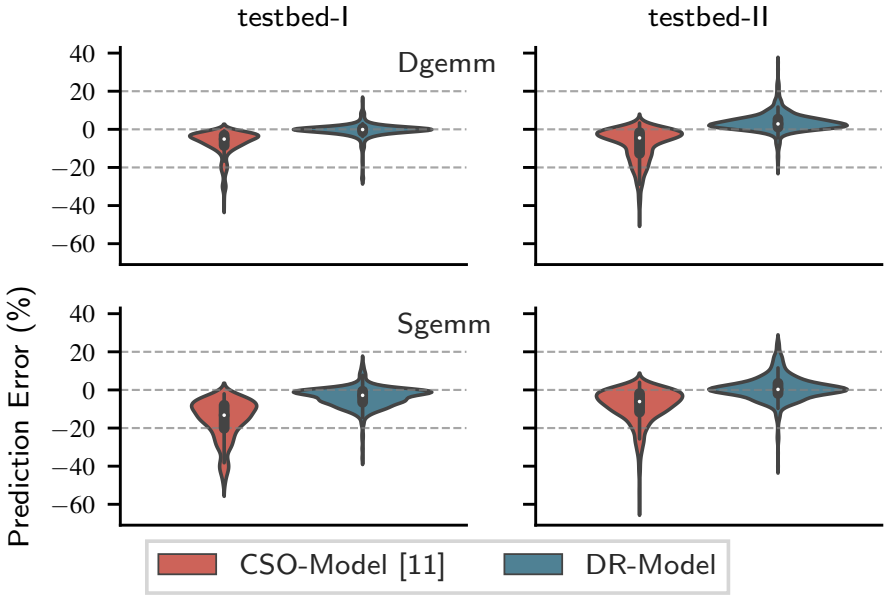


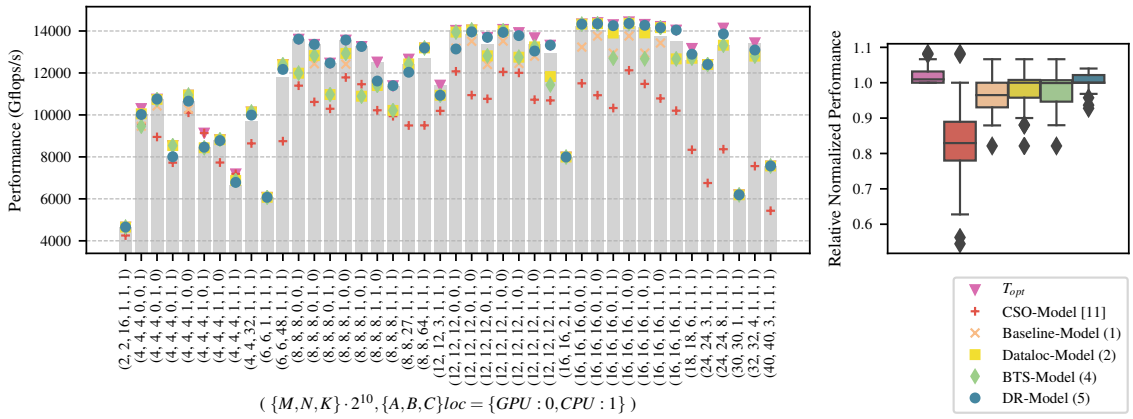
Figure 2.8: Error distribution of the *CSO-Model* and the *DR-Model* for our CoCoPeLia wrapper BLAS implementation of `sgemm` and `dgemm` on testbeds I and II.

on the NVIDIA Tesla V100 GPU of Testbed II for cublas_{D,S}gemm, which are not present in the NVIDIA Tesla K40 of Testbed I. We attribute these to the more complex GPU architecture of the former.

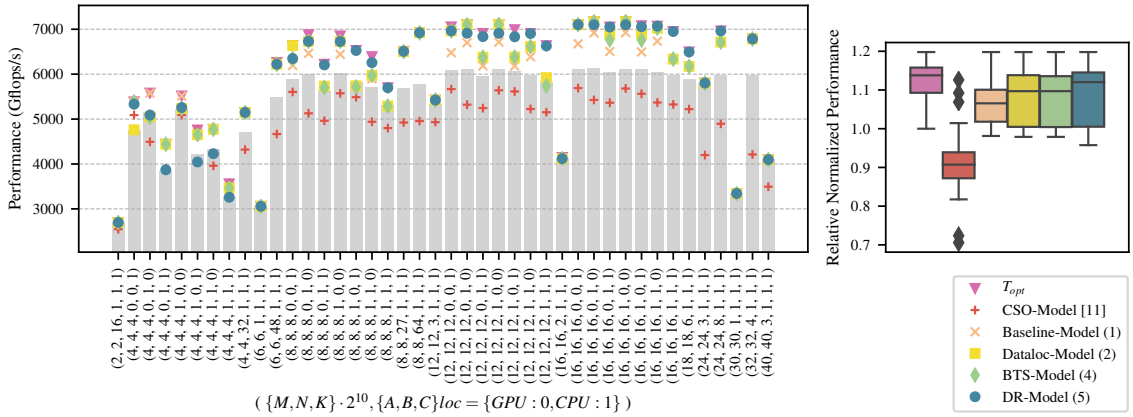
2.4.4 Validation of tiling size selection

Subsequently, we validate the CoCoPeLia tiling size selection ability when used in practice. The target of the CoCoPeLia framework is to predict the tiling size that leads to near-optimal performance. We hence consider the following scenario: for all validation cases in Section 2.4.2, we explore the performance achieved by the prediction of each model, and how this compares with a good baseline tiling size and the maximum achievable performance, using the optimal tiling size T_{opt}.

Figure 2.9 shows the results of this comparison for `dgemm` and `sgemm` on Testbed II. The selected baseline tiling size is $T = 2048$. First, the CSO-Model mispredicts the optimal tiling sizes in both cases leading to performance degradation compared to the baseline. This happens mostly because the CSO-Model does not take into account the non-linearity of execution time, which results in favoring small tiles with limited performance. Additionally, it is evident that the baseline is enough to provide near-optimal performance for Figure 2.9a, where even T_{opt} provides a median performance improvement of 1%, and a maximum of 10%. The CoCoPeLia models provide performance close to the baseline, with the DR-Model surpassing its performance, but less than



(a) Sgemm



(b) Dgemm

Figure 2.9: Evaluation of Tile selection ability for Sgemm (a) and Dgemm (b) on testbed II. The baseline performance (gray bars) is acquired using a static tiling size $T = 2048$, also used by BLASx. We compare this against the experimentally achieved performance using the optimal tiling size for each problem, $T = T_{opt}$, the performance achieved using the tiling size predicted with the `CSO-Model` [159] and c) the performance achieved using $T = T_{best}$ returned by $CoCoPeLia\_select$ using Equations 2.1, 2.2, 2.4, 2.5 respectively.

1% (which is close to the T_{opt} median). On the other hand, in the case of Figure 2.9b, T_{opt} is able to provide improvements of a median of 13.5% and up to 20%. In this case, the incremental improvement of each CoCoPeLia model is more evident; the *Baseline-Model* (Equation 2.1) provides a median speedup of 7%, the *Dataloc-Model* (Equation 2.2) and *BTS-Model* (Equation 2.4) both provide median improvement of 10%, and the DR-Model (Equation 2.5) provides 12% improved performance, which is very close to the T_{opt} median. We note that bidirectional slowdown, considered by the *BTS-Model*, does not significantly affect the performance of `gemm`, which requires

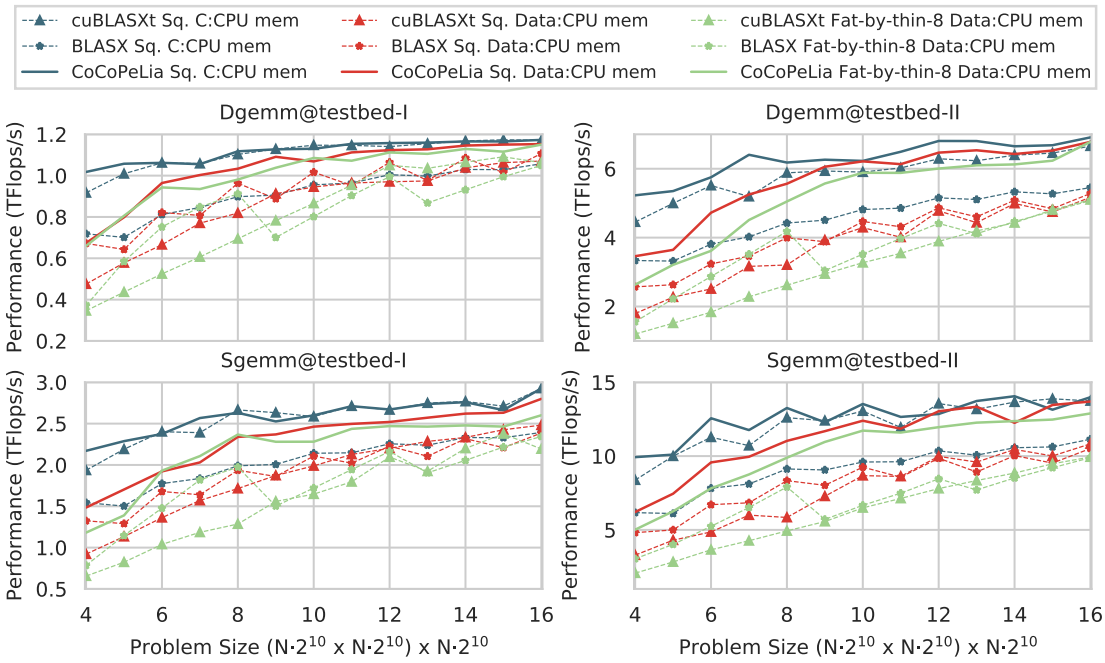


Figure 2.10: `dgemm` and `sgemm` performance evaluation for various problem sizes on testbeds I,II. We use three scenarios with different transfer-to-computation ratios: 1) $M = N = K$ with A, B on the GPU and C requiring update from the CPU (blue), 2) $M = N = K$ with A, B, C on the CPU (red) and 3) $N = M = \frac{K}{8}$ with A, B, C on the CPU (green).

fewer d2h than h2d transfers. Its impact is more evident in level-1 BLAS functions with similar transfers to and from device memory.

2.4.5 Performance evaluation

To evaluate the end-to-end performance of the runtime scheme proposed in Figure 2.6, we extend the validation set of Section 2.4.2. For `daxpy`, we select 11 large problem sizes $N = (1 \xrightarrow{\text{step}=N*2} 1024) \cdot 2^{20}$ for all three location combinations. For `sgemm` and `dgemm`, we select 25 square problem sizes $M = N = K = (4 \xrightarrow{\text{step}=0.5} 16) \cdot 2^{10}$ for all seven location combinations and for all thin/fat ratios of the aforementioned validation set. Overall, we evaluate 33 problems for `daxpy` and 325 for `dgemm` and `sgemm`.

We compare the performance of CoCoPeLia `sgemm` and `dgemm` against *cuBLASXt* and *BLASX*. These are both multi-GPU libraries, but *cuBLASXt* is the state of practice and *BLASX* offers the most performance for transfer-bound cases, deeming them the most relevant comparison targets for single GPU 3-way concurrency. We compare the performance of CoCoPeLia `daxpy` against a unified memory implementation with prefetching. Our *BLASX* results use the

Table 2.4: (geo)mean percentile CoCoPeLia performance improvement over state of the art GPU BLAS libraries.

System	Testbed I		Testbed II	
Offload Scenario	Full	Partial	Full	Partial
$daxpy$	21.5%	9.4%	19.9%	9.1%
$dgemm$	16.2%	5.8%	32.2%	15.6%
$sgemm$	20.6%	5.7%	33.3%	15.7%

library default - static - tiling size $T = 2048$. For *cuBLASXt*, which accepts the tiling size as an input parameter, we test 10 different tiling sizes and choose the best for each problem. We note that this nearly-exhaustive tiling size selection gives a performance advantage to *cuBLASXt* over *BLASX*.

Figure 2.10 visualizes the performance of the three libraries for `dgemm` and `sgemm` on our two testbeds, for three scenarios of problem sizes and data locations. We first note that *BLASX* outperforms *cuBLASXt* in fat-by-thin matrices, while *cuBLASXt* shows better performance in the low transfer cases, where only the C matrix resides on the CPU. Second, CoCoPeLia outperforms both *BLASX* and *cuBLASXt* in all three scenarios. For the low-transfer scenario (blue), its performance is on par with *cuBLASXt*, but it considerably outperforms the other two libraries for the full offload scenario (red) and the transfer-heavy fat-by-thin matrix multiplication (green). Third, CoCoPeLia provides better relative performance on testbed II, which has a lower bandwidth/FLOP ratio and therefore transfers are a bigger bottleneck.

In Table 2.4 we summarize the mean percentile performance improvement of CoCoPeLia over the best among the two other libraries for each problem size, calculated using the geometric mean of the fraction of their times, respectively. We separate full and partial offload cases for reference with relevant literature, where full offload refers to all data residing on the CPU, and partial offload to some of the data residing on the GPU. The results are similar to the outlined cases in Figure 2.10; CoCoPeLia outperforms the other libraries by 16-33% in the full offload case and 5-15% in the partial offload case, indicating that it is able to improve cuBLAS performance without architecture-specific tuning or bias towards specific data shapes.

CHAPTER 3

Extending model-based autotuning for multi-GPU and heterogeneous systems

This chapter describes the extension of our single-GPU modeling and autotuning approach to multi-GPU and heterogeneous systems. We first briefly overview the differences between single- and multi-GPU systems, and provide a brief background on the difficulty of multi-GPU BLAS optimization (Section 3.1.2) and the resulting shortcomings of previous multi-GPU BLAS libraries (Section 3.1.3). Then, we describe the extension of *CoCoPeLiA* into *PARALiA* (Section 3.2), an end-to-end multi-GPU BLAS framework that combines modeling and autotuning to enable portable communication optimization and device selection. Finally, we evaluate *PARALiA's* performance and energy efficiency and compare it with previous SoTA libraries (Section 3.3).

3.1 Problem formulation

Multi-GPU libraries allow input data to reside on host memory, GPU memory, or a combination of both and internally manage all data distribution and computation on multiple devices. Our work focuses on level-3 BLAS routines similar to most existing multi-GPU BLAS libraries [11,54,57,61, 68,124,125,157,161]. 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 first show the low performance and energy efficiency of current multi-GPU BLAS libraries in modern systems, which motivates us to explore multi-GPU BLAS optimization. Then, we present the performance

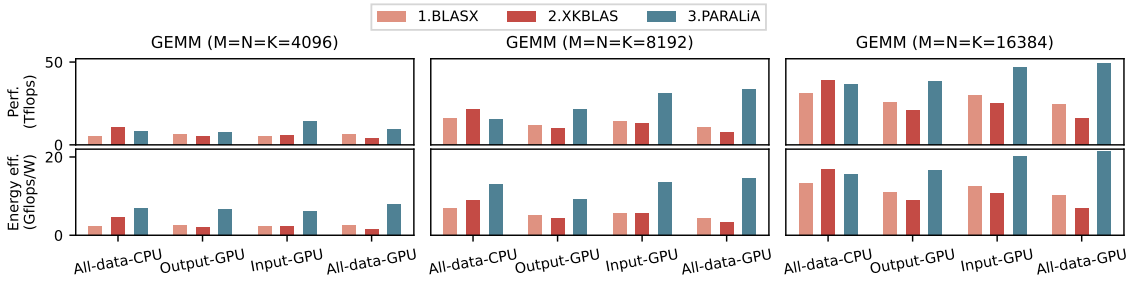


Figure 3.1: The GEMM performance (top) and energy efficiency (bottom, using the power-delay product) of the state-of-the-art multi-GPU BLAS libraries BLASX [157] and XKBLAS [57] and PARALiA [9] (our work), in a multi-GPU cluster with 8 NVIDIA-V100 GPUs, for three problem sizes and four different data 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.

bottlenecks of multi-GPU Level-3 BLAS and analyze the limitations of current approaches in mitigating them.We also discuss the absence of consideration for resource utilization in existing multi-GPU BLAS libraries that derives from these limitations and the relevant efficiency and heterogeneity challenges. Finally, we define the problem formulation for this chapter and list our corresponding contributions.

3.1.1 Motivation

In theory, level-3 BLAS operations are particularly favorable for multi-GPU execution since they are highly parallelizable and are usually characterized by high operational intensity. *In practice*, multi-GPU execution introduces a variety of new bottlenecks that can severely impair performance even for the most convenient problems. For example, Figure 3.1 shows the performance of state-of-the-art multi-GPU libraries for matrix-matrix multiplication (henceforth *gemm*), the most common level-3 BLAS kernel, 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 placements, where some 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. Finally, it is very important to note that, unlike single-GPU, where performance optimization targets a 1.1x-1.3x performance gap between the SoTA and the system peak, in

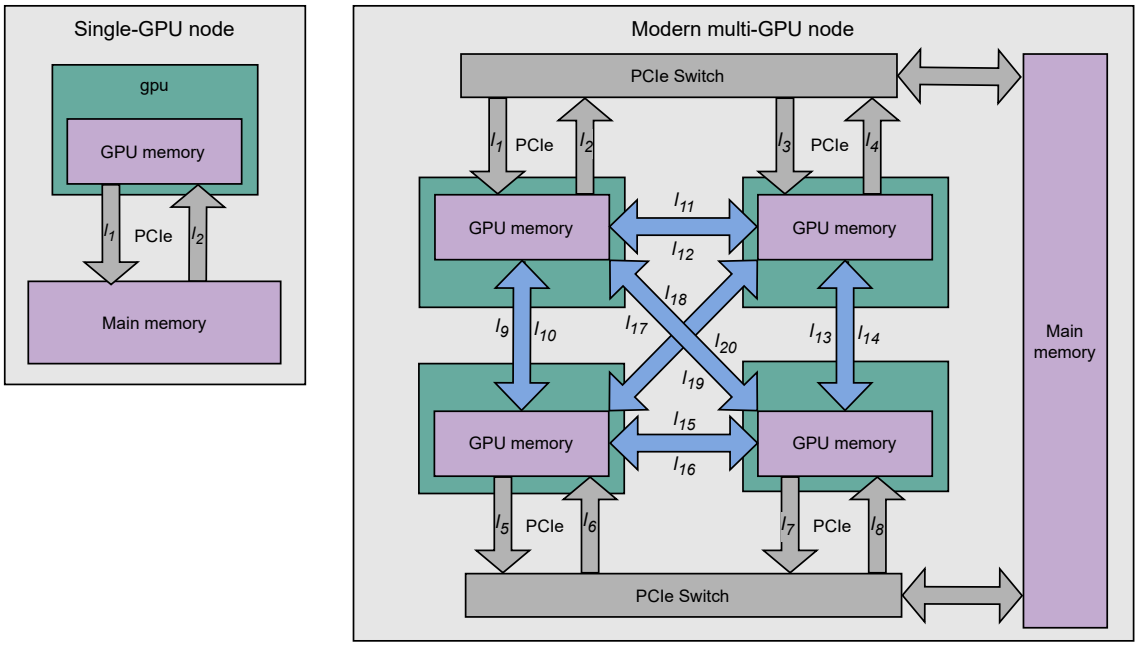


Figure 3.2: An example single-GPU cluster (left) versus a multi-GPU cluster with 4 GPUs (right), showing the large difference in their hardware characteristics. The single-GPU cluster has two communication channels (henceforth *links*) for h2d and d2h communication while the multi-GPU one has 20 connections with potentially different bandwidths and partial resource sharing.

multi-GPU the margin is much larger, with previous libraries operating at 10-30% (3.3x-10x performance gap) in problematic configurations. *Ideally*, a multi-GPU BLAS library should provide robust performance regardless of data placement and avoid under-utilizing hardware to conserve energy whenever possible. An example of this behavior can be seen in Figure 3.1 for our work *PARALiA* [9], introduced later in this chapter. PARALiA offers performance robustness regardless of data configuration and can adapt to each problem, using fewer devices if possible to achieve similar performance coupled with higher energy efficiency.

Takeaway Multi-GPU BLAS libraries suffer from severe **performance** degradation in all but the most common *full-offload* case, and lack mechanisms for **energy efficient** execution.

3.1.2 From single- to multi-GPU clusters

To understand the multi-GPU BLAS optimization problem, we must analyze the key architectural features that influence the performance, and thus, library design, in multi-GPU setups: the increasingly complex underlying interconnect and the distinct GPU memories. Figure 3.2 illustrates the difference between the architecture of a single-GPU cluster vs a simple 4-GPU cluster. As discussed in Chapter 2, a single-GPU interconnect can be characterized by the h2d ($l2$) and

d2h ($l1$) channels, their throughput and their slowdowns for simultaneous usage $sl_{\{h2d,d2h\}\_bid}$. On the other hand, in the multi-GPU example, there are 20 different channels, some sharing the PCIe (*l1-l8*) and some featuring faster GPU-GPU connections (*l9-l20*) with potentially complex relations due to resource sharing, which makes both multi-GPU programming and modeling much more complex. On top of this, unlike the single-GPU case which involves a static number of channels (h2d,d2h) and compute workers (1 GPU), the architecture of multi-GPU systems varies per cluster; it can feature different numbers of GPU workers and different interconnection patterns, changing the optimization problem considerably and increasing modeling difficulty.

Takeaway The architectural complexity of multi-GPU nodes deems BLAS optimization and modeling considerably more complex than single-GPU.

3.1.3 Background : Offloading BLAS in Multi-GPU clusters

In addition to interconnect complexity, the GPUs in multi-GPU systems must also act as parallel workers, which introduces distributed and parallel programming concepts in multi-GPU BLAS execution. As a result, unlike single GPU setups where algorithmic optimizations mainly target the internals of the BLAS kernels, multi-GPU also requires communication and scheduling optimization. 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*, c) *avoidance*, and d) *routing* and e) *load-balancing* between GPUs.

3.1.3.1 Level-3 BLAS decomposition and distribution

As Level-3 BLAS routines operate on matrices, their decomposition involves partitioning matrices into smaller submatrices ($T \times T$ 2D tiles), which are then distributed among the available GPUs that act as parallel workers, similarly to multi-core or multi-node execution [29]. This decomposition must be carefully managed to ensure balanced workloads across GPUs, minimizing the risk of some GPUs idling while others are overloaded. Additionally, the decomposition and distribution schemes also determine the required amount of communication between workers.

While there is a large amount of research on BLAS decomposition in general that spans back decades [1, 28, 38, 51, 94, 142, 151], the limited number of workers in multi-GPU systems leads to different optimization priorities compared to distributed computing. Since expanding single-GPU modeling and autotuning techniques to multi-GPU environments is already challenging, in this chapter we use the well-established 2D block-cyclic decomposition and distribution method employed by state-of-the-art multi-GPU BLAS libraries [57, 157]. We go into more detail regarding selecting a more suitable decomposition in chapter 4.

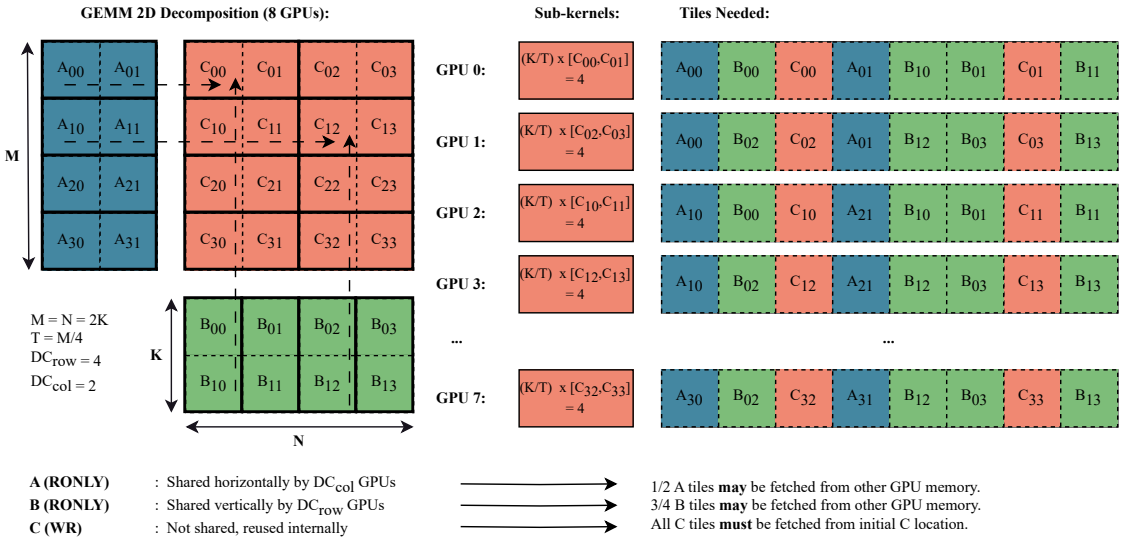


Figure 3.3: An example of GEMM (M = N = 2K) 2D decomposition to sub-problems and data tiles (tiling size T = M/2). The 8 participating devices are distributed in a 2D grid of (DC_{row}, DC_{col}) = (4, 2) to encourage horizontal and vertical device-to-device (d2d) data movement between same row/column 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.

Figure 3.3 shows an example of the 2D block-cyclic distribution used for Level-3 BLAS matrix-matrix multiplication (GEMM). The available devices are organized in a virtual 2D grid, which is as square as possible - for example, a 2x2 grid for 4 devices, a 4x2 or 2x4 grid for 8 devices, a 3x3 grid for 9 devices etc. Then, all problem matrices are split into equal square $T \times T$ tiles, and the C output matrix tiles are mapped on top of the device grid, defining which sub-problems will be executed on each device. This results in a list of A, B and C tile dependencies that must be fetched before execution on each GPU, which defines the *communication pattern* for the problem execution. The advantage of 2D block-cyclic GEMM decomposition over other options is that it results in a favorable communication pattern for the read-only (henceforth *RONLY*) tiles of matrices A and B. This is the basis of multi-GPU BLAS *communication optimization*, as communication is the main bottleneck in multi-GPU BLAS performance on modern systems [7, 57, 157]. While different libraries have used different approaches for improving communication performance, the optimization targets can be roughly classified into communication *overlap*, *avoidance*, and *routing*.

3.1.3.2 Communication overlap

Overlap refers to the concurrent execution of computation and communication, as well as the concurrent execution of multiple communication tasks. While we provide a detailed examination of both types of overlap for single-GPU in Chapter 2, overlap complexity increases significantly in multi-GPU systems. Regarding communication overlap, in addition to the overlap of host-to-device (h2d) and device-to-host (d2h) transfers for each GPU, we must also consider the overlapping of GPU-GPU communication (d2d). Additionally, computation overlap can happen between all GPUs in the system acting as workers. Consequently, the communication-computation overlap pipeline width in multi-GPU setups depends on the number of GPUs within the node, rendering both modeling and overlap optimization considerably more complex. For instance, as illustrated in Figure 6.1, the relatively straightforward 3-way overlap pipeline of a single GPU (h2d and d2h transfers, along with computation on one GPU) becomes much more complex in a multi-GPU system, where overlap involves $(gpu_{num}+1)^2-(gpu_{num}+1)$ communication channels and gpu_{num} GPUs potentially engaged in computation. Nevertheless, while overlap affects total multi-GPU performance to some degree, communication avoidance and routing are more important since their absence can lead to severe bottlenecks. We, therefore, utilize overlap similarly to previous work [7, 57, 157] but do not focus on its model-based optimization, which is left for Chapter 4.

3.1.3.3 Communication avoidance

In a multi-GPU environment, where each GPU has a distinct memory, the problem data must be transferred between devices. While technologies such as Remote Memory Access (RMA) help mitigate the impact of these transfers, a common approach to further reduce redundant communication is data caching or buffering in GPU memory, which also enables data reuse across subsequent subkernels, resulting in *avoiding communication*. In the case of multi-GPU level-3 BLAS, communication avoidance arises from reusing the RONLY input matrix tiles (the tiles of the A and B matrices for the GEMM example of Figure 3.3) that are created by the decomposition process. While minimizing communication is vital for performance optimization, it is relatively straightforward, and communication reduction techniques are well-established in previous research for each decomposition. Consequently, in this chapter, we employ the pre-established data caching strategies for communication reduction of the 2D block-cyclic distribution used by SoTA libraries [7, 57, 157].

Takeaway Multi-GPU BLAS decomposition, communication overlap, and avoidance affect performance but are fairly straightforward and sufficiently optimized in previous work.

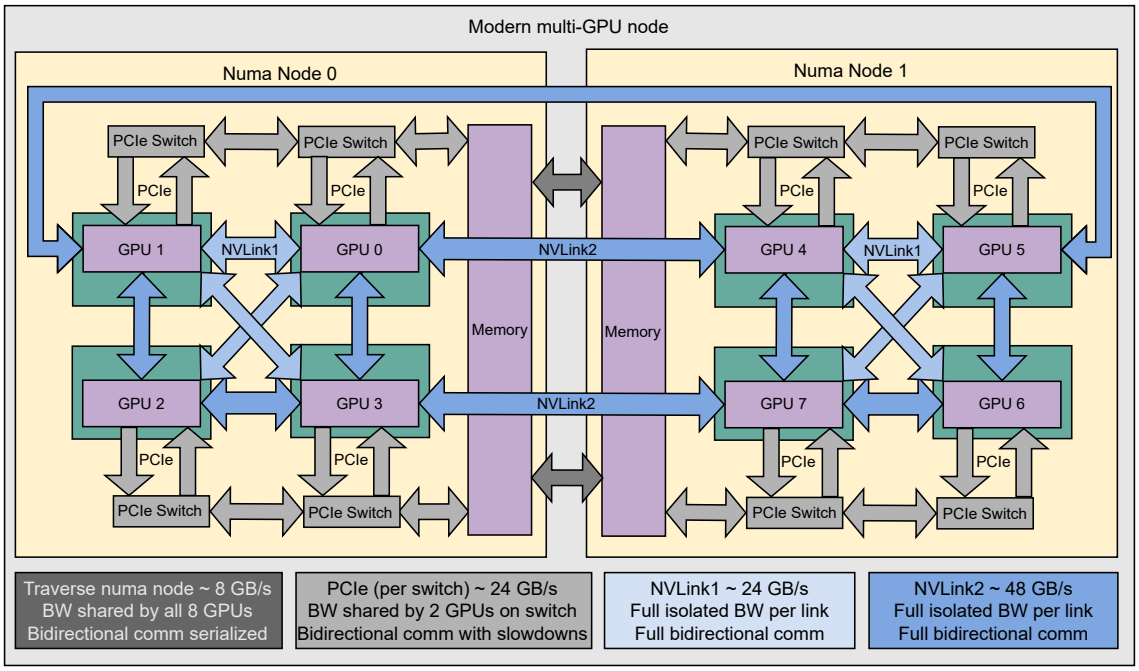


Figure 3.4: A *clx-ai* node of HLRS' HPC cluster Vulcan [69] that features 8 NVIDIA Tesla V100 GPUs and a mixed interconnect with various bandwidth levels and resource-sharing properties. The interconnect utilizes a mix of NVlink-1 ($\approx$ 24 GB/s) and NVlink-2 ($\approx$ 48 GB/s) for inter-GPU connectivity, with the GPUs not being fully connected via NVlink. For CPU-GPU communication it uses PCiE ($\approx$ 12 GB/s), and CPUs share PCIe bandwidth in sets of two (e.g. GPU 0-1, 2-3 etc). Finally, each node has 2 numa nodes connected with a $\approx$ 8 GB/s link shared between all GPUs.

3.1.3.4 Communication routing

Routing optimization refers to selecting the *fastest* route to move some data to a destination memory ($dest_{mem}$) whenever a *routing decision* is involved, e.g. the data are available in multiple potential source memory locations (src_{mem}). In the context of 2D block-cyclic BLAS decomposition, routing decisions appear for RONLY tiles only, since these are required in multiple GPU memories during execution. Consequently, two components are necessary to enable routing optimizations: 1) a cache policy for GPU memory buffers that encodes which data tiles are available where and ensures their Read/Write dependencies are respected, and 2) a mechanism to distinguish the interconnect bandwidth levels in order to select the 'closest' src_{mem} location when a RONLY tile is available in multiple buffers. The first issue has been thoughtfully explored in previous work and can be resolved with a relatively straightforward software optimization, where cache-like metadata are added to the CPU/GPU buffers, which coupled with a MESI-like protocol enables sharing RONLY blocks between GPUs and respecting WR block dependencies [157].

The second issue on the other hand is much more complex, since it requires an estimation of all interconnect bandwidths and their relations/interactions when used simultaneously. In simpler multi-GPU systems like the one in Figure 3.2, this usually boils down to distinguishing between the high-bandwidth peer-to-peer transfers between GPUs (*d2d*) and the lower bandwidth CPU-GPU transfers (*h2d/d2h*). In these simple cases, latency-bandwidth models (similar to those described in Chapter 2) coupled with some additional logic for PCIe switch sharing are sufficient for representing the system interconnect hierarchy. Alternatively, another similar option is to use a system-specific hierarchical representation/abstraction (like BLASx [157] and XKBLAS [57, 58]), which assumes a certain system layout with predefined bandwidth levels.

Routing complexity of modern clusters and SoTA limitations: While both aforementioned approaches work relatively well for simpler systems, they do not apply to more recent complex systems that feature heterogeneous interconnects with complex sharing relations. An example of such a system is depicted in Figure 3.4, which shows the *clx-ai* testbed used later in our experiments (more in Section 3.3). In *clx-ai*, each link has different properties and behavior when used simultaneously or bi-directionally, making modeling and routing very complex for arbitrary problems. In this case, latency-bandwidth modeling or simple hardware abstractions can only estimate the system bandwidths in a *rest state*, e.g. when the interconnect is not utilized, or for *predetermined communication patterns*, where the load is known before execution. Finally, on top of *modeling complexity*, the *performance impact* of routing also increases, since *bad routing* can cause severe bottlenecks due to the extreme differences in bandwidth levels. For example, performing one inter-numa-node transfer per GPU simultaneously has ≈ 48 times lower bandwidth than using 8 simultaneous NVLink2 transfers in *clx-ai*.

In the case of multi-GPU BLAS, each problem's load and communication depend on the decomposition, which in turn depends on the input routine parameters, namely the *problem dimensions* and the *initial matrix locations*, which only become available during invocation. The prevalent technique to overcome this ambiguity is to *assume* some values for these parameters offline using commonly used values and optimize the resulting communication pattern. The usual assumption, which derives from the most common use case of BLAS, is to use square problem dimensions and assume all data are initially on the CPU (*full-offload*). Consequently, as shown in Figure 3.1, SoTA libraries underperform for all other data placements, even though they should theoretically have higher performance due to being less communication-bound (since the data are *closer* to the GPUs). We demystify the cause for this counter-intuitive behavior in Figure 3.5, which shows the communication pattern, number of transfers and average achieved bandwidth for BLASX, XKBLAS and our proposed runtime, PARALiA. BLASX suffers from excessive h2d/d2h transfers to the CPU, which could be completely avoided in the scenario where all data are initially on the GPUs [57]. Additionally, BLASX is bandwidth-agnostic, with transfers passing through a variety of bandwidth levels, since its hierarchical abstraction is only capable of

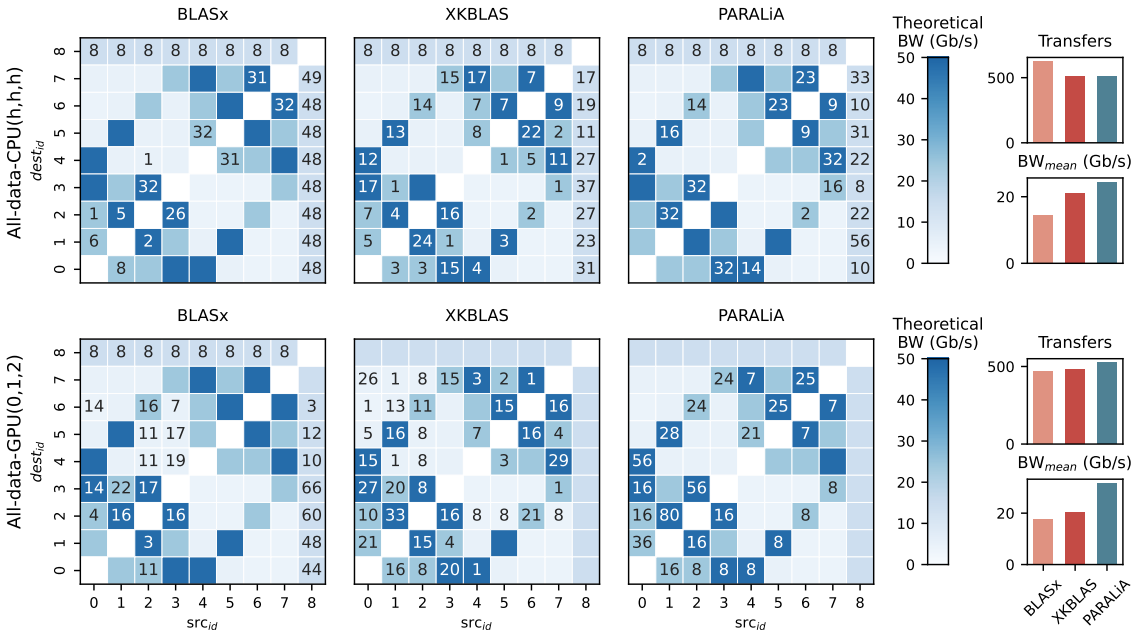


Figure 3.5: The communication pattern of BLASX, XKBLAS and PARALiA for a GEMM execution (M = N = K = 16384, T = 2048) in the testbed of Figure 3.4, for two data placements: the full-offload case (all data at host memory initially) and a case where the A, B and C matrices are initially populating the 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.

recognizing the difference between *h2d/d2h* and *d2d*, and is not sufficient for modern interconnects. XKBLAS, on the other hand, 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 assumption that the communication pattern will be similar to the full-offload case, where all data must first be fetched from the host ($id = 8$ in the heatmaps), which has the same bandwidth for all h2d/d2h connections and therefore favors a balanced distribution. This does not represent the second scenario well, since 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.

Takeaway Communication routing is the prevalent bottleneck for multi-GPU BLAS because current libraries do not account for the heterogeneity in the underlying interconnect.

3.1.3.5 Load balancing

As mentioned previously, in multi-GPU BLAS execution GPUs function as parallel workers, each responsible for executing tasks (sub-kernels) generated by the decomposition of the larger problem. To maximize parallelism and fully exploit the computational capabilities of all GPUs involved, BLAS libraries must effectively load balance these tasks. This process is relatively straightforward in balanced scenarios, such as full-offloading or in systems with uniform data distribution and homogeneous interconnects, and is usually performed with task graph optimizations [56,57] or work stealing [57,157]. However, the challenge becomes significantly more complex in heterogeneous scenarios, where differences in GPU capabilities, the interconnect layout, or data configurations must be carefully managed to avoid imbalance. In such cases, a library should dynamically adapt to the varying conditions, ensuring that the computational and communication load is evenly distributed.

Heterogeneity and energy efficiency SoTA limitations: Although work stealing and task graphs may partially address the load imbalance coming from small performance differences between GPUs, they are also performance-agnostic; they match *a list of tasks* (sub-kernels) to a certain *list of resources* (devices), without the option to use less resources depending on the problem. This is an issue because communication-bound problems, which are limited by the interconnect bandwidth rather than the number of devices, do not benefit from utilizing additional GPUs if the interconnect throughput is the primary constraint, and can even result in *less performance* when using *more devices*. Unfortunately, all current multi-GPU libraries lack a mechanism to adjust the number of utilized devices dynamically, and always default to using all devices or expect the user to specify otherwise. This monolithic design does not lead to efficient execution since it cannot adapt resource allocation to BLAS problem characteristics and is incompatible with future heterogeneous systems.

Takeaway Current multi-GPU BLAS load-balancing mechanisms fall short in heterogeneous configurations and cannot adapt to problems that cannot utilize all devices well.

3.1.4 Contributions

The main shortcomings of current multi-GPU BLAS libraries in modern systems can be surmised in three points. First, they often **assume homogeneous characteristics for the interconnect**, which may not hold true across all systems, leading to sub-optimal performance when

these assumptions are violated. Second, these libraries typically optimize communication routing based on **specific problem data characteristics**, commonly focusing on the full-offload scenario, thereby limiting their flexibility and efficiency for other workloads. Finally, the **absence of performance modeling** in these libraries prevents them from dynamically adjusting execution strategies to suit varying problems, resulting in potential inefficiencies and performance losses. In this chapter, we propose a model-based approach to address these issues, 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. Overall, we make the following contributions:

1. We extend single-GPU transfer modeling for arbitrary multi-GPU clusters, with a system abstraction that encodes system characteristics and adjusts communication routing during runtime in order to better fit to different problem layouts (Section 3.2.2).

2. We explore performance-aware workload distribution and device selection for multi-GPU BLAS (Section 3.2.1), using performance modeling with a variety of target metrics (Section 3.2.3) fueled by empirical micro-benchmarks (Section 3.2.4).

3. We extend the CoCoPELiA 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 1.7X performance and 2.5X energy efficiency improvement over state-of-the-art libraries. (Section 3.3).

3.2 PARALiA: BLAS autotuning in arbitrary multi-GPU systems

In this section we present *PARALiA*, an extension of CoCoPeLiA for multi-GPU systems that addresses the shortcomings of previous multi-GPU libraries by coupling modeling with autotuning. We begin with a high-level overview of the modified framework and its basic components, and then describe each component and its role in the optimization pipeline in more detail. Figure 3.6 shows the the high-level design of the PARALiA framework. PARALiA is activated when user code 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 (sec 3.2.5) that is responsible for preparing the framework environment for execution, a scheduler (sec 3.2.6) that is responsible for managing input/output data and invoking backend BLAS kernels, and an autotuner (sec 3.2.1) that receives system and problem parameters from a database (sec 3.2.4), a hardware abstraction (LinkMap, sec 3.2.2) 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.

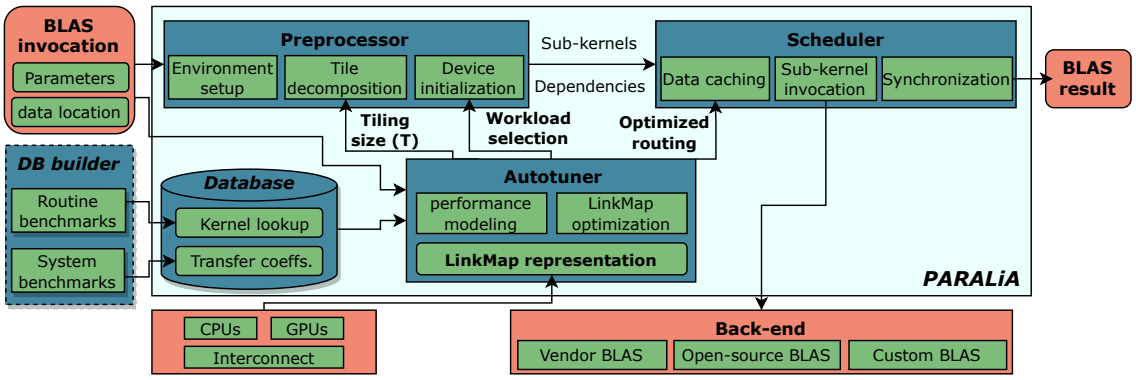


Figure 3.6: An overview of the PARALiA framework and its main components.

3.2.1 The autotuner algorithm

The autotuner, an extension of CoCoPeLiA's *tile selection runtime*, 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 3.7. 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 sec 3.2.3), 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 sec 3.2.2) Finally, the best *workload distribution* is selected by using some metric-related

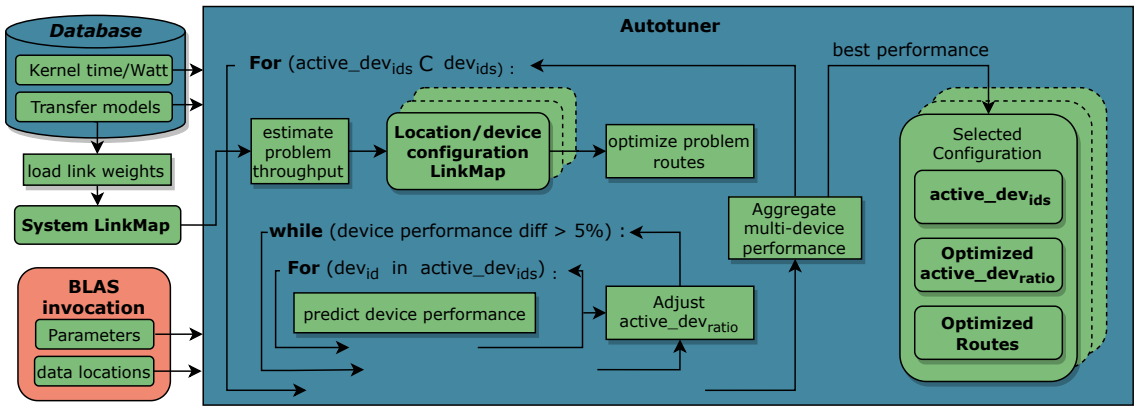


Figure 3.7: An overview of the PARALiA autotuner and its prediction pipeline.

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 3.6), but this process is disconnected from (a), (b) and (c) and performed based on CoCoPeLiA [7] due to its small impact in multi-GPU performance.

3.2.2 Abstracting interconnect heterogeneity: 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.

Hardware abstraction: To model any potential system, 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 3.1. 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.

Table 3.1: `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.

Runtime routing optimization: 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. This process has three basic phases, implemented as the LinkMap functions of Table 3.1. 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 (Eq. 3.6) to the bandwidth 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 apply a simplified *shortest path* algorithm to this graph, similar to the Floyd–Warshall algorithm but with a maximum number of *hops* (intermediate locations). More specifically, we want to reroute transfers that would pass through links with low bandwidth to *series of links* of higher bandwidth. For example using a maximum of 3 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 $0 \rightarrow 1$ can be optimized to $0 \rightarrow 2 \rightarrow 1$, since it is 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, 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. In practice, this re-routing method unlocks even more potential for BLAS communication optimization, but since some of the additional concepts required for this are not compatible with current library design, we leave it for Chapter 4.

Takeaway The LinkMap abstracts *system characteristics* offline, and combines them with BLAS *problem characteristics* during runtime to enable communication routing optimizations.

3.2.3 Performance estimation for workload selection

As explained in Sec. 3.2.1, 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 [159], instead of using more advanced overlap models [7, 65, 159]. 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 3.2 summarizes the modeling notation used in this work.

First, we combine the *full-overlap* upper bound [159] with the PARALiA database to get a routine-specific, full-overlap prediction for *each* device's total performance:

$$pred\_t_{base}(dev_{id}) = max(t_{exec}(dev_{id}, dims), \\ t_{h2d}(dev_{id}, \sum_{i}^{is_R} bytes(i)), t_{d2h}(dev_{id}, \sum_{j}^{is_W} bytes(j))) \tag{3.1}$$

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

Table 3.2: 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.

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:

$$t_{link}(dest_{id}, src_{id}, bytes) = link_{lat}(dev_{id}, src_{id}) + \frac{bytes}{link_{bw-shared}(dest_{id}, src_{id})} \tag{3.2}$$

by combining each link's latency and bandwidth using the well-accepted latency/bandwidth model [7, 16, 21, 65, 106, 159]. Then, we assume the best-case scenario, where all input matrices/vectors are distributed equally between the dev_{num} devices by combining eq. 3.1 with eq. 3.2

to generalize for any initial data placement:

$$pred\_t_{over}(...) = max(t_{exec}(...), \sum_{i}^{is_R} t_{link}(dev_{id}, data_{locs}(i), \frac{bytes(i)}{dev_{num}}), \sum_{j=0}^{is_W} t_{link}(data_{locs}(j), dev_{id}, \frac{bytes(j)}{dev_{num}})) \quad (3.3)$$

Equation 3.3 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 the sum of their corresponding $bytes$ is $RONLY\_sum\_bytes$. 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:

$$t_{extra}(dev_{id}) = extra\_transfer\_bytes \cdot \frac{dev_{num}}{\sum_{idx=0}^{dev_{num}} link_{bw-shared}[dev_{id}][idx]} \quad (3.4)$$

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 Eq. 3.4 to Eq. 3.3:

$$pred_t(dev_{id}, ...) = max(t_{exec}(...), t_{extra}(dev_{id}) + \sum_{i}^{is_R} t_{link}(...), \sum_{j}^{is_W} t_{link}(...)) \quad (3.5)$$

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.

Takeaway The extended PARALiA models predict the total execution time of *any level-3 BLAS problem* for an *arbitrary multi-device system* as a function of the utilized devices, enabling problem-specific device selection.

Performance metrics: In adidtion to time, 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.

3.2.4 Database

Similarly to CoCoPeLiA, 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 *Deployment Phase* component in CoCoPeLia [7], 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 [7], 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 [71]. 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:

$$sl_{link1-link2} = (t_{link1-link2_{over}})/t_{link1} \tag{3.6}$$

Since the method of obtaining the sl is empirical, we assume a maximum slowdown $sl_{link1-link2}$ of 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.

3.2.5 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 [57, 124, 157, 161], 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 [57, 157] that rely on dynamic load-balancing, PARALiA follows a static approach since load-balancing is based on effective performance estimation performed before scheduling.

3.2.6 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 [157]. 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 [57,157]. 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.

3.3 Experimental 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 [124], BLASX [157] and XKBLAS [57], 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). 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.

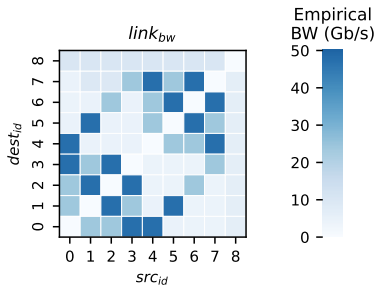
3.3.1 Experimental setup

For the performance evaluation we use a single testbed: the "clx-ai" nodes of HLRS' HPC cluster `Vulcan` [69]. System details are presented in Table 3.3, along with the interconnect bandwidths stored in the `LinkMap` for the 9 devices (8 GPUs + CPU). 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 (`cudaDevice Synchronize()`) also included; both timer and synchronization overhead were less than 1% for all benchmarks. We

Table 3.3: **CLX-AI** 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 x16	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

Figure 3.8: **CLX-AI** interconnect Linkmap.



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.

3.3.2 Evaluation Dataset

3.3.2.1 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 sec. 2.1, 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 [157], 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.

3.3.2.2 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 3.9) 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 [61] and PARSEC [161] that were designed taking older GPU architectures into account, as they are outperformed by both BLASX and XKBLAS.

3.3.3 Comparison with state-of-the-art

3.3.3.1 Performance

Figure 3.9 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,

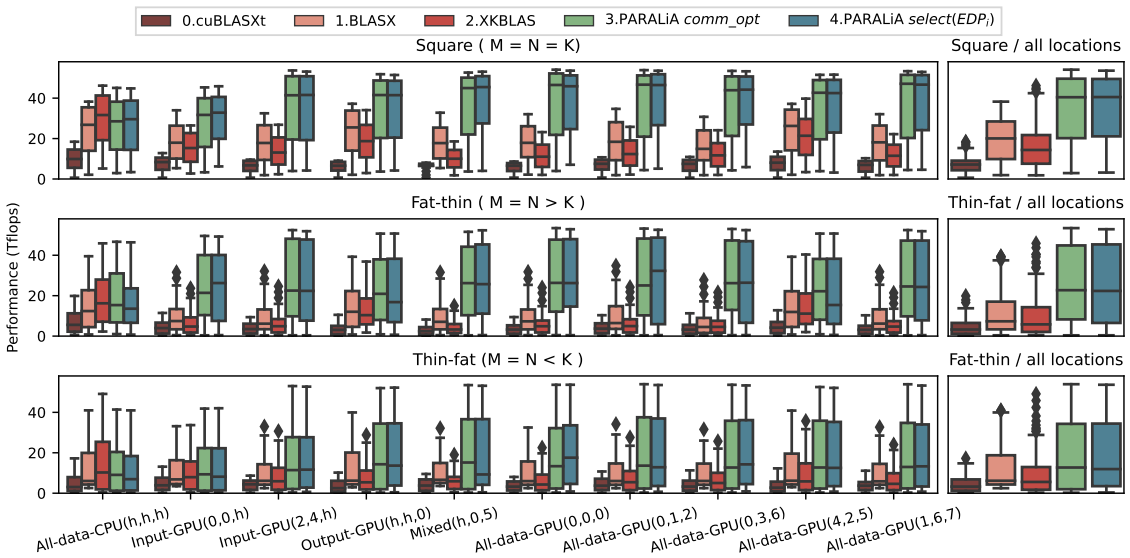


Figure 3.9: Performance of `dgemm` with cuBLASXT, BLASX, XKBLAS, and two variants of PARALiA (one always utilizing all GPUs and one selecting the workload distribution to maximize the inverse energy-delay product - EDP_i), on the dataset described in subsection 3.3.2. Each row corresponds to a different 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.

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 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

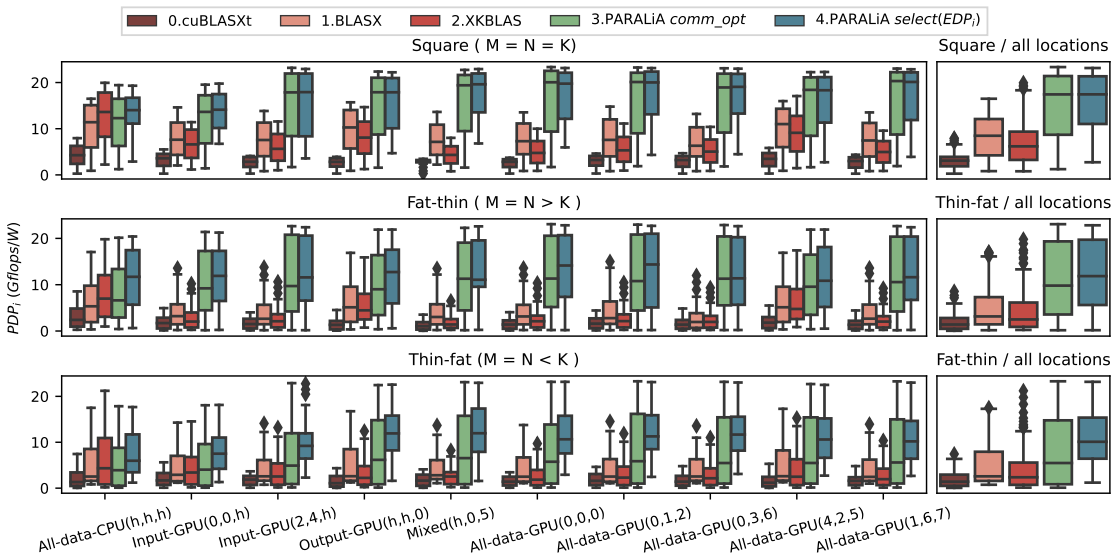


Figure 3.10: Energy efficiency of `dgemm` (Gflops/W) for all problem configurations presented in Figure 3.9. cuBLASXt, BLASX, XKBLAS and PARALiA *comm_opt* have PDP_is relative to their performance (since they all utilize all 8 system GPUs), resulting in a much better PDP_i for PARALiA due to its higher 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.

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 3.9 illustrates that PARALiA outperforms previous approaches in terms of performance (details in sec. 3.3.3.3) in a complete, diverge dataset, containing various transfer- and compute- bound cases, due to its better communication optimization scheme.

3.3.3.2 Energy efficiency

Figure 3.10 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 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.

3.3.3.3 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 3.9 and 3.10 such problems are overshadowed by the compute-bound portion of the total dataset, we include Table 3.4 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 ζαμπονιαhow *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 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

Table 3.4: A summary of the performance of `dgemm` for the whole dataset for each implementation, using the $\frac{mean(Gflop)}{mean(metric)}$ [70]. 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.

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

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 (1.5X 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.

3.3.4 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 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 3.5 and Figure 3.11.

Figure 3.11 contains the aggregated performance (left) and energy efficiency-PDP_i (right) box-plots 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 3 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.

Figure 3.11: DGEMM performance (Tflops) and energy efficiency (Gflops/W) for half of the problem configurations presented in Figure 3.9. 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.

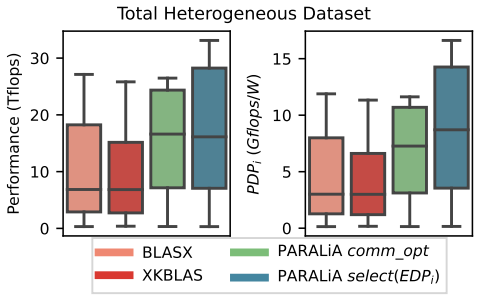


Table 3.5: A table summarizing the GEMM performance for the whole dataset for each implementation, using the $\frac{mean(Gflop)}{mean(metric)}$ [70] for half of the small (HS), large (HL) and total (HT) problems of Table 3.4 ran on a heterogeneous emulated system. PARALiA outperforms all multi-GPU scheduler-based approaches both in performance and energy efficiency, further boosted by a better workload selection.

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

Table 3.5 shows the achieved mean performance in a similar layout with subsection 3.3.3 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 3.11 we use Table 3.5 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 form 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 1.12X mean performance and 1.3X mean PDP_i improvement while *PARALiA select(EDP_i)*

offers 1.06X performance and 1.5X 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.

CHAPTER 4

A communication-aware multi-GPU matrix multiplication library

This chapter describes the culmination of our thesis: the merging of the model-based autotuning of previous chapters with a highly-optimized schedule to provide a state-of-the-art library for the most widely used BLAS kernel, general matrix multiplication (GEMM). First, we analyze the motivation for selecting this final part of our thesis, which involves the shortcomings of previous approaches in providing peak performance for matrix multiplication and the considerable unrealized potential of the model-based knowledge PARALiA provides (Section 4.1). Then, we describe our end-to-end matrix multiplication library design inspired by distributed GEMM but bolstered with knowledge-driven communication optimizations (Section 4.2). Finally, we evaluate the scalability, performance, and robustness of our approach and compare it with state-of-the-art libraries (Section 4.3).

4.1 Problem formulation

In Chapter 2, we introduced a generalized modeling approach for BLAS operations in general within a single-GPU context, laying the basis for enhancing performance through dynamic runtime autotuning. Building on this, in Chapter 3 we focused on level-3 BLAS operations, which are more common in multi-GPU clusters due to their higher arithmetic complexity. Focusing on level-3 BLAS instead of modeling a broader range of BLAS operations allowed us to address the

increased modeling complexity of multi-GPU environments without making excessive generalizations. On the same note, in this final chapter, we want to delve even deeper by concentrating on a single kernel: the general matrix multiplication (GEMM).

Why use GEMM?: *In theory*, focusing on a single kernel narrows the applicability of our approach from PARALiA's level-3 BLAS target. *In practice*, level-3 BLAS kernels are either matrix-multiplications with different data input types/layouts or perform mainly GEMM operations internally [157]. This makes GEMM the most fundamental operation in high-performance computing (HPC) and machine learning (ML) applications, and deems its optimization equally important for the optimization of level-3 BLAS in general. Consequently, our objective is to conduct an exhaustive optimization of GEMM, addressing the kernel-specific challenges and fine-tune our optimization strategies beyond the more generalized approaches of previous chapters.

4.1.1 Background: Optimizing GEMM for Multi-GPU systems

In general, there are two paths to execute GEMM on a multi-GPU compute node, each with its own advantages and disadvantages.

The theoretical approach: The first path is to utilize existing libraries for distributed GEMM (or BLAS in general) that have been extended with GPU support [11, 18, 54, 68, 94, 123, 161]. Distributed GEMM libraries are characterized by a very solid *theoretical foundation*, since distributed GEMM optimization is a problem with decades of previous work. Consequently, they utilize complex decomposition and scheduling algorithms and advanced communication reduction techniques [1, 23, 28, 38, 51, 94, 142, 151] (more details in Chapter 5). The state-of-the-art multi-GPU distributed libraries are SLATE [54], PARSEC [68] and cuBLASMp [123]. SLATE implements the SUMMA [151] algorithm for GPUs, distributing problem data on their memories with a PBLAS user-friendly C++ data layout. PARSEC offers a similar approach, based on task graph optimization [68, 161], which can support larger problems than SLATE with better scalability. cuBLASMp is the most recent NVIDIA implementation for multi-node multi-GPU GEMM for pre-distributed PBLAS data layout, and offers state-of-the-art performance for data pre-distributed to GPUs. The main problem of these approaches is that their algorithms and optimizations are designed for *scalability* on hundreds (or thousands) of workers and and *very large problem sizes*, not towards single-node performance. Consequently, they result in low performance, especially for smaller more communication-bound problems.

The practical approach: The second multi-GPU GEMM path is the path of single-node optimization, with specialized libraries designed specifically for *performance* [9, 57, 124, 125, 157]. In contrast to large-scale distributed systems, where research on GEMM focuses on communication-optimal algorithms and complex process decomposition grids, multi-GPU nodes feature a fairly

small number of GPU devices. Consequently, these libraries focus more on the *implementation*, using lower-level scheduling, overlap, and communication optimization techniques available within one node as described in Chapter 2.1. Due to the proximity of these optimizations to the system architecture and the general complexity of multi-GPU optimization, these libraries usually focus on the most performance-effecting subset of these optimizations *for their development systems* and for *certain problem configurations*. This results in robust performance for these systems and problems, but leaves room for improvements for the rest that might favor different optimizations [9]. The most evident example of this is the state-of-the-art single-node multi-GPU library XKBLAS [57]. As we showed in Chapter 2.1, XKBLAS [57] reduces communication volume and employs a lightweight DAG-based scheduling to balance GPU work, but employs a simplistic heuristic-based mechanism for routing and applies naive communication-computation overlap and communication balancing techniques. As a result, XKBLAS performs well for regular problems in regular systems, but suffers from severe performance degradation in nodes with irregular interconnects and non-square problem shapes.

Understanding the gap: To bridge the gap between the theoretical and practical approaches, one must understand what it entails within the context of one node. Distributed GEMM libraries are communication and/or memory optimal, but they are not designed specifically for multi-GPU and are therefore *oblivious* of the GPU hardware. Consequently, they are more robust to changes in hardware or problem characteristics but lose some performance. On the other hand, single-node multi-GPU GEMM libraries are *closer* to the GPU hardware, prioritizing lower-level techniques, but do not incorporate the sophisticated theoretical foundations and advanced communication reduction strategies of distributed GEMM solutions. Therefore, practical approaches offer higher performance when fine-tuned for specific scenarios/hardware, but suffer severe performance degradation when system or problem parameters change [9].

Takeaway Despite the advancements in distributed and single-node GEMM multi-GPU optimization, a significant gap remains in addressing the intersection of these two domains. Consequently, libraries either offer *high performance* for certain problems/systems, or *robustness* but with a trade-off in performance.

4.1.2 Background: PARALiA limitations on covering this gap

The middle way: modeling and autotuning: The original motivation for CoCoPeLia and PARALiA is that modeling offers a *midway* solution that balances the theoretical and practical approaches. Our approach towards this *midway* was to abstract system and problem characteristics by integrating analytical performance models, which are initialized empirically in each system. Then, an autotuner is integrated into an existing library, enabling dynamic, adaptable

optimizations during runtime for some tunable parameters. This enables more generalized yet efficient performance across diverse systems and problem configurations, overcoming some limitations inherent in purely theoretical or practical strategies.

Current PARALiA limitations: While PARALiA represents a significant advancement in multi-GPU BLAS optimization, it still is a *midway* between performance and robustness/portability. For instance, in the case of GEMM, it struggles to fully exploit the potential of modern multi-GPU nodes, losing 5-15% performance compared to XKBLAS in favorable scenarios such as full offload. The first shortcoming of PARALiA lies in the communication optimization provided by LinkMap. LinkMap operates re-actively, determining routing decisions at runtime without a comprehensive view of the data flow. This approach, while adaptive, misses the opportunity for proactive optimization based on a known decomposition. Additionally, the LinkMap relies solely on bandwidth considerations for routing decisions, neglecting the *current load* on communication links when a decision is taken. Furthermore, the intermediate hops in rerouted transfers that the LinkMap optimization algorithm suggests increase the total number of transfers. This bandwidth-extra communication trade-off is usually better for performance, but it still deviates from the performance peak of a communication- and bandwidth-optimal solution. Finally, the most performance-effecting problems of PARALiA, derive from the scheduling, decomposition, and caching techniques adapted from previous work and combined with autotuning. While these methods have been enhanced by performance modeling, they remain limited by their design complexity and lack of full integration between modeling and implementation.

The best of both worlds for GEMM?: Using runtime task schedulers has been the standard for multi-GPU BLAS libraries, since they must support arbitrary BLAS problems with different communication, computation and concurrency needs. However, when focusing specifically on GEMM, the execution is influenced solely by the *problem dimensions, data placement*, and the *decomposition* strategy. Given that the problem characteristics become available when the routine is invoked and decomposition occurs before execution, all communication/computation requirements and dependencies can be determined before execution. Consequently, *routing, overlap, and scheduling decisions* can be made *proactively* and based on the actual communication, not a model-based estimation. Additionally, these decisions can be made considering the execution process as a whole rather than making individual decisions *reactively* during runtime, opening up the possibility for more complex optimizations. Unfortunately, the external autotuner of PARALiA as described in Chapter 3 is insufficient for this purpose, because it operates independently from the preprocessing and scheduling components. Instead, achieving the best of both worlds requires an end-to-end approach that incorporates model-driven knowledge in the decomposition, routing, and scheduling algorithms by design.

Takeaway Modeling and autotuning can offer a *midway* solution between high performance and robustness, but there is still considerable room for improvement when applied on GEMM.

4.1.3 Contributions

In this chapter, we focus on an optimized multi-GPU GEMM library that achieves both near-optimal performance and robustness. Our approach is based on a static schedule that is calculated ahead of execution, whenever a GEMM routine is called with a new set of parameters. Each static schedule created for a given problem is reusable by subsequent routine calls, zeroing out scheduling overheads for all but the first call. Knowledge of the input parameters gives us a complete view of the routine's communication pattern and scheduling characteristics, which we exploit to simultaneously minimize communication volume, maximize interconnect utilization, optimize overlap, and minimize imbalance and GPU idle time. We note that state-of-the-art multi-GPU libraries for linear algebra [9,57,58,157] already implement some of the optimizations introduced in this chapter, like *caching* (Section 4.2.3), *overlap* (Section 4.2.2) and *BW-based routing* (Section 4.2.4). Our work refines these optimizations, employing a simpler caching scheme, achieving full (rather than partial) overlap of all computation/communication streams, and reducing routing overhead through static scheduling. Additionally, our work introduces novel optimizations including *ETA-based routing* (Section 4.2.4.2), *RONLY-fetch batching* (Section 4.2.5), *lazy WR-tile fetching* (Section 4.2.6) and *ETA-based sub-kernel ordering* (Section 4.2.7).

In summary, we make the following contributions:

1. Starting from a baseline implementation of multi-GPU GEMM, we introduce a set of communication, algorithmic, and auto-tuning optimizations, inspired by both GPU and distributed scheduling, leading to a near-optimal multi-GPU GEMM that combines the best of both worlds.

2. We provide an open-source multi-GPU GEMM library[1] that simplifies efficient multi-GPU computation. Our library is compliant with the BLAS standard, uses the LAPACK data layout, and can handle any combination of CPU- and GPU-resident matrices. This flexibility allows both for easy drop-in replacement of existing GEMM routines, and integration with libraries that already distribute data across multiple GPUs.

3. We evaluate our optimized GEMM on a multi-GPU NVIDIA HGX system with 8 NVIDIA A100 GPUs interconnected with NVLink3 and NVSwitch2, for single and double precision, using a variety of GEMM problem sizes, shapes and matrix memory placements. Our results show that our implementation offers 1.37× and 1.29× higher performance over state-of-the-art libraries on average, for single and double precision respectively.

[1]Available at *https://github.com/p-anastas/PARALiA-GEMMex*

4.2 Implementation

In this section, we describe the design of a static schedule inspired by distributed and multi-GPU approaches, that optimize GEMM for multi-GPU. From the available BLAS or PBLAS standard input layouts for GEMM, we follow the BLAS standard [39], with the input matrices stored in LAPACK layout, residing either on host or GPU memory [9, 57, 124, 157]. A GEMM routine following the BLAS definition performs the operation:

$$C_{out} = a \cdot A \times B + b \cdot C_{in} \tag{4.1}$$

Assuming that C_{out} is stored in-place in the C_{in} buffer, the operation requires three matrices, $A(M \times K)$, $B(K \times N)$ and $C(M \times N)$, with A and B being read-only (henceforth denoted as *RONLY*) and C being read and written (henceforth denoted as *WR*) internally.

4.2.1 Hierarchical decomposition

The decomposition of matrices A, B, and C determines the *subproblems* to be executed on each GPU and also determines the communication pattern. In decomposing the problem, we opt to avoid inter-GPU sharing of the C matrix, since it leads to additional inter-GPU synchronization and extra communication for multi-GPU execution [9, 57, 157] . We select a 2D decomposition similar to the SUMMA [151] algorithm, as a simple and practical solution, upon which we apply a number of optimizations.

Figure 4.1 depicts our proposed *hierarchical decomposition* based on the blocking version of the SUMMA distributed algorithm [151]. The first decomposition level is based on the number of GPUs, which are the processing elements, viewed as a 2D grid of $r \times c$ GPUs. We decompose C matrix into 2D-blocks of $M_r \times N_c$, and assign each block individually to a single GPU. We then decompose matrices A and B into blocks of rows of size $M_r \times K$ and blocks of columns of size $K \times N_c$, respectively. The row-blocks of matrix A are assigned to c GPUs and the column-blocks of matrix B to r GPUs.

At a second level, the blocks of A, B, and C on each GPU are further decomposed into 2D blocks, or else *tiles*, of size $T \times T$, applying padding where required. This results in a 3D grid of $\frac{M}{T} \times \frac{N}{T} \times \frac{K}{T}$ square GEMM subproblems, each requiring different input and output tiles. To obtain the GEMM routine output C_{out}, we then need to compute the tile-based outer-product [151]:

$$C_{i,j} = b \cdot C_{i,j} + \sum_{k=0}^{\frac{K}{T}} a \cdot A_{i,k} \times B_{k,j} \tag{4.2}$$

for $i = 0 \rightarrow \frac{M}{T}$ and $j = 0 \rightarrow \frac{N}{T}$.

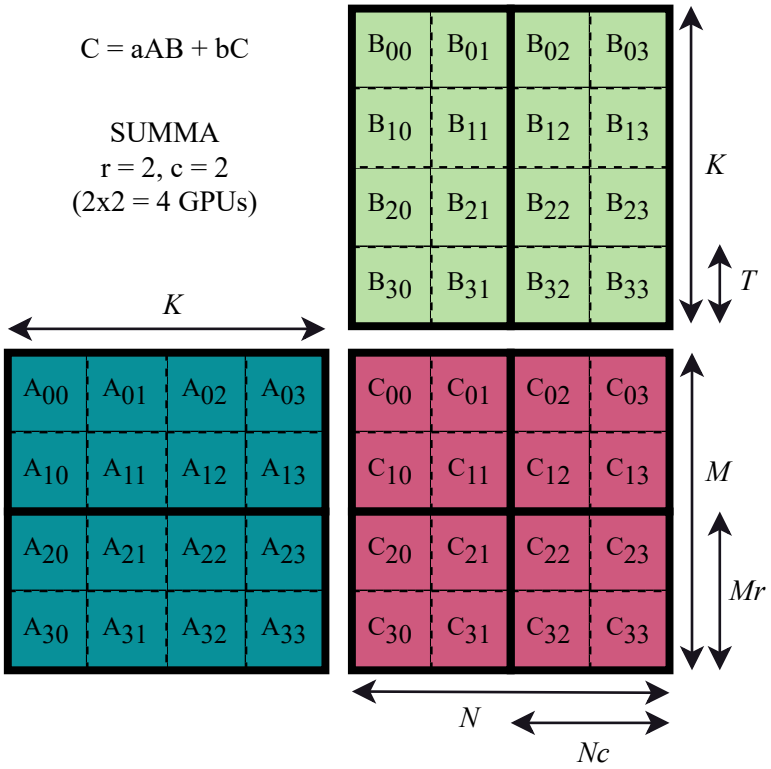


Figure 4.1: An example of our GEMM 2-level hierarchical decomposition for a square problem (M, N, K) based on the SUMMA blocking algorithm [151]. The first level depends on the number of workers (here: 4 *GPUs*) split to a 2D grid $(r, c) = (2, 2)$ which decomposes (M, N) to (M_r, N_c) chunks, leaving K untouched. Then, the second level is splitting (M, N, K) to 2D square (T, T) *blocks*, which creates square GEMM sub-problems and enables communication/-computation overlap.

The value of T 1) should avoid excessive padding, 2) must be small enough to create a sufficient number of sub-problems for overlap and 3) must be large enough to avoid kernel, transfer, and scheduling latencies [7, 9, 63, 65, 159]. To ensure a balance between the three, we minimize a composite cost function, based on three heuristics. The first heuristic tries to avoid padding by penalizing any remainder when dividing M, N, K to tiles:

$$C_{padding}(T) = \sum_{i \in \{M,N,K\}, i \nmid 0} \frac{i}{T}$$

The second heuristic concerns the ability to overlap and penalizes the problem percentage that cannot be overlapped, estimated as the inverse of the overlap pipeline length, equal to the number

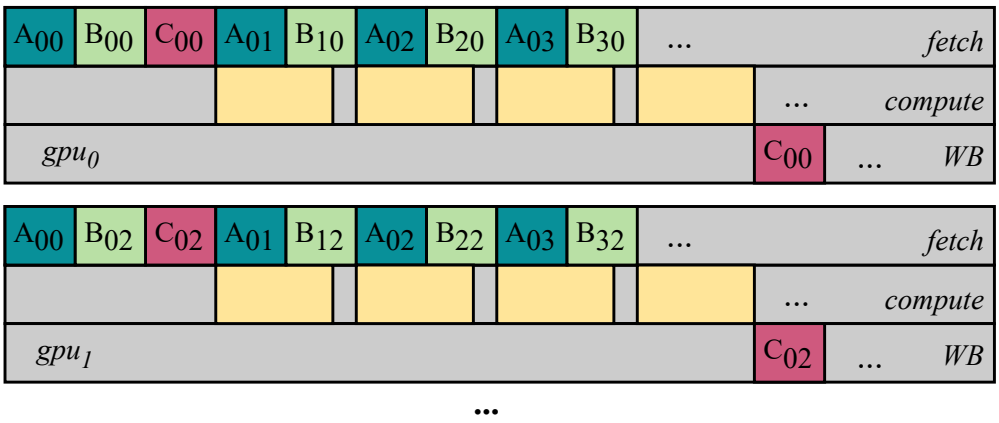


Figure 4.2: The process of executing the tasks of the first four sub-problems as seen in Figure 4.1 on two GPUs in parallel. Tasks of different types (*fetch*, *compute*, *WB*) are placed on different streams and overlapped in a pipelined manner for each GPU, using different streams for intra-GPU parallelism.

of sub-problems per GPU:

$$C_{overlap}(T) = \frac{gpu\_num}{\frac{M}{T} \times \frac{N}{T} \times \frac{K}{T}}$$

The last heuristic for latency uses a minimum pre-selected tile size T_{min} (default = 2048), large enough to avoid high latency, and penalizes smaller tiles proportionally:

$$C_{latency}(T) = \frac{T_{min} \times 0.2}{T}$$

We apply these heuristics before second-level decomposition by selecting the tile size T that minimizes $C_{total}(T) = C_{padding}(T) + C_{overlap}(T) + C_{latency}(T)$, for all $T = 128 \rightarrow min(M_r, N_c, K)$.

4.2.2 Communication/computation overlap

The end-to-end offloading of a GEMM subproblem that results from decomposition to a GPU requires fetching the input dependencies, i.e. the required input tiles A_T, B_T, and C_T, computing the kernel and potentially writing back the result C_T to the location of the initial matrix. We therefore consider each decomposed subproblem as a five-task process: tasks (1)-(3) of fetching the input dependencies, $fetch^{dst}_{src}(A_T)$, $fetch^{dst}_{src}(B_T)$, $fetch^{dst}_{src}(C_T)$, task (4) of computing the kernel $compute$, and task (5) of writing back the output $WB^{src}_{dst}(C_T)$. The subproblems and their corresponding tasks enable intra- and inter-GPU parallelism, as the different tasks can be scheduled on different streams.

Figure 4.2 depicts a simplified example of scheduling the first four sub-problems of the aforementioned decomposition of Figure 4.1 on gpu_0 and gpu_1, respectively. We asssume that matrices A, B, and C initially reside on the same location ($A_{loc} = B_{loc} = C_{loc}$). Placing the different types of tasks ($fetch$, $compute$, and WB) on different streams enables overlap in a pipelined manner. This reduces the total time for a subproblem from $t_{total} = t_{fetch(A)} + t_{fetch(B)} + t_{fetch(C)} + t_{compute} + t_{WB(C)}$ to approximately $t_{total} \approx max((t_{fetch(A)} + t_{fetch(B)} + t_{fetch(C)}), t_{compute}, t_{WB}(C))$ [159]. In our implementation, overlap works similarly, but additionally, we enqueue the data transfer tasks (*fetch*, *WB*) on $(gpu\_num + 1)^2$ different CUDA streams, enabling simultaneous bidirectional point-to-point communication between all devices plus the host memory. Consequently, if A_{loc}, B_{loc} and C_{loc} are discrete locations/device memories, the *fetch* tasks are also overlapped, resulting in $t_{total} \approx max(t_{fetch(A)}, t_{fetch(B)}, t_{fetch(C)}, t_{compute}, t_{WB(C)})$. For computation/communication overlap, we schedule the *compute* tasks, performed using cuBLAS [122] kernels, on a configurable number of CUDA streams per GPU (the default is 8). Finally, task dependencies between streams are defined and respected with CUDA events [126] similarly to previous approaches [9, 57, 157].

4.2.3 Data caching / Communication avoidance

To avoid excessive communication, most multi-GPU BLAS implementations [9, 57, 157] *cache* the tiles that are *fetched* to a GPU memory for a specific subproblem, to be reused by subsequent subproblems. Caching is necessary for problem sizes that do not fit in the GPU memory, and important for performance as it reduces the communication volume, but its management adds some overhead. To avoid this, we refrain from a complex caching mechanism, based on the fact that, in recent systems, the GPU memory sizes are large enough to hold the necessary data and GEMM becomes compute-bound long before memory capacity becomes an issue. Instead, we use a buffer per GPU, denoted as $SoftBuf[i]$, with $i = 0 \rightarrow gpu\_num$. This buffer is allocated whenever a GEMM routine is called for the first time, tailored to the memory requirements of the decomposition of Figure 4.1 for that specific problem. The buffer stores the necessary tiles throughout the entire routine lifetime. If a subsequent GEMM routine has a larger memory requirement, the buffer is automatically resized.

4.2.3.1 Offloading problems exceeding memory capacity

It is common for modern multi-GPU nodes to feature large host memories (in the order of TBs) that far exceed the capacity of GPU memory (in the order of tens of GBs). For the specific case of large problems, where A, B, and C all reside in host memory and the $SoftBuf$ memory requirements exceed the GPU memory capacity, we employ an additional level of decomposition

on the host side, before applying the hierarchical decomposition of Section 4.2.1. This decomposition is automatically triggered upon routine invocation, when the problem size would result in any $SoftBuf[i]$ larger than a preset percentage (default = 80%) of the available GPU memory on gpu_i. We decompose the original problem dimensions M_L, N_L, K_L in a 3D manner into square tiles of size T_L, resulting in a 3D grid of GEMM subproblems of size $M \times N \times K$, with $M = N = K = T_L$. We select the maximum T_L that satisfies the memory requirement for the $SoftBuf$ buffer. We then schedule each subproblem sequentially onto the GPUs, with each subproblem utilizing all the optimizations described in this work. We note that this limits the communication-computation ratio and consequently, the total performance for the (M_L, N_L, K_L) problem to that of the (T_L, T_L, T_L) problem.

4.2.4 Communication routing

The main communication volume in multi-GPU GEMM results from the read-only (RONLY) tiles A_T, B_T of matrices A and B, which must be fetched to multiple GPUs, where the relevant subproblems are to be executed. During execution, the first fetch of each RONLY tile, e.g. A_T, to a gpu_i requires a transfer from its original matrix location in_loc, e.g. $fetch_{in\_loc}^{gpu_i}(A_T)$, that stores A_T to $SoftBuf[gpu_i]$. On the other hand, subsequent tile fetches to any other device gpu_j can either fetch a copy of the tile from in_loc or from gpu_i, which requires a *communication routing* decision.

4.2.4.1 Bandwidth-based routing

A common approach to perform communication routing is to use the bandwidth between the different memory locations, based on the interconnect topology [9, 58, 157]. The selection of the optimal route is based on the available bandwidth. For the example given above, where A_T may reside on both in_loc and gpu_i, the decision boils down to comparing $BW_{in\_loc}^{gpu_i}$ and $BW_{gpu_i}^{gpu_j}$ and selecting the route with the highest bandwidth. We estimate the bandwidth of the different $(gpu\_num + 1)^2$ routes/streams empirically with micro-benchmarks, as in [9].

4.2.4.2 Accounting for interconnect load

While bandwidth-based routing can be effective because it increases the average bandwidth utilization, the goal of communication routing is to minimize the estimated time of arrival (ETA) of a tile, i.e. the end-to-end time to fetch the input data to the target GPU, so that the *compute* task starts as early as possible. Bandwidth-based routing fails to take into account the pre-existing load over a *link*, which may delay the time of arrival of this tile. An example is depicted in Figure 4.3, where A_{00} needs to be fetched to gpu_0. Bandwidth-based routing selects to fetch the tile

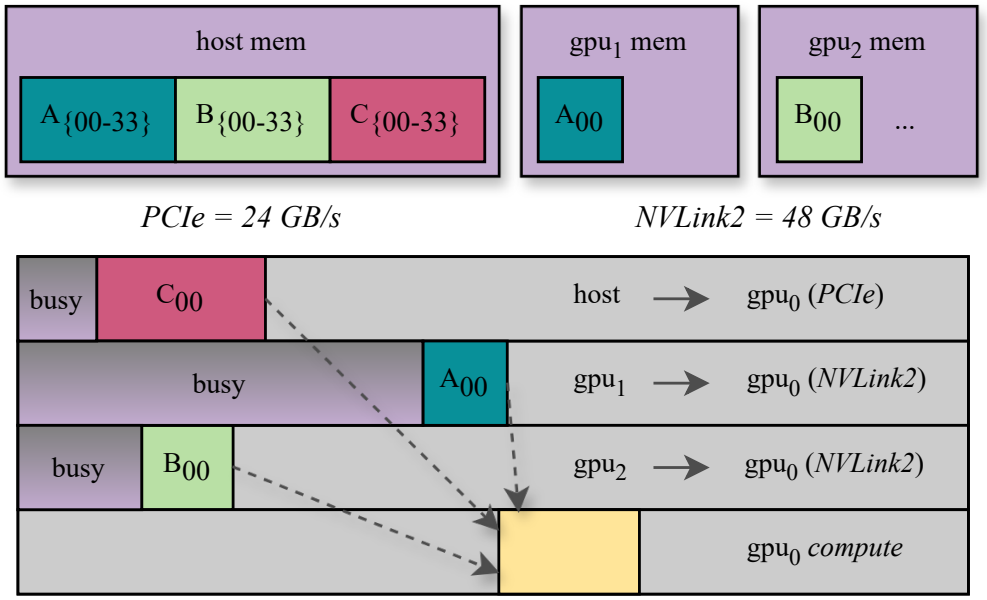


Figure 4.3: An example of bandwidth-based routing misprediction that results in increased GPU idle time. When the sub-problem with input dependencies (A_{00}, B_{00}, C_{00}) is scheduled in gpu_0, A_{00} and B_{00} are already available in gpu_1 and gpu_2 from previous tile fetches. Since BW-based routing is unaware of interconnect load, it copies A_{00} from gpu_1 and B_{00} from gpu_2 since these transfers utilize higher P2P GPU bandwidth. In the case of A_{00}, this results in gpu_0 compute blocking longer than if A_{00} was fetched from the host instead, due to a high pre-existing load in $gpu_1 \rightarrow gpu_0$.

from gpu_1, since $BW_{gpu_1}^{gpu_0} > BW_{host}^{gpu_0}$, without accounting for the high load on that stream from previous copies, resulting in the delayed arrival of A_{00} to gpu_0, and rendering gpu_0 idle.

To avoid this effect, we optimize communication routing by considering the interconnect load when selecting the route for a tile transfer. To achieve this, we define a 2D matrix for the link availability (henceforth denoted as *LAM*), which stores the estimated time each $src \rightarrow dst$ communication stream will be available (e.g. without remaining load). We additionally define the *ETA vector* for each decomposed tile, which stores the estimated time that a valid copy of this tile will arrive at each memory location. Initially, all LAM fields are set to zero. All ETA vector fields are set to inf, except for the initial data location of the tile, which is set to zero ($ETA[in\_loc] = 0$). During scheduling, whenever the scheduler needs to take a routing decision, to fetch a tile of $size$ bytes to dst, the scheduler combines the LAM and ETA vectors with the estimated fetch cost $t_i^{dst} = \frac{size}{BW_i^{dst}}$, searching for the source i with the lowest ETA_{min}, where:

$$ETA_{min} = \min_{i=0}^{gpu\_num+1} (\max(ETA[i], LAM[dest][i]) + t_i^{dst})$$

Link availability matrix

src / dest	G0	G1	G2	G3	H
G0	-	40	15	...	10
G1	...	-	...	...	...
G2	...	...	-	...	...
G3	...	...	...	-	...
H	...	...	...	...	-

Tile ETA vectors

A_{00}	
G0	?
G1	30
G2	inf
G3	inf
H	0

B_{00}	
G0	?
G1	inf
G2	15
G3	inf
H	0

Figure 4.4: An example state of LAM and ETA vectors for two tiles A_{00} and B_{00}. The tile ETA vectors show when these tiles should be available to GPUs 1 and 2, respectively (initial on host memory - ETA[h] = 0), while the LAM stores an estimation for the interconnect load up to that scheduling state.

Then, the LAM and tile vector are updated for the new transfer to $LAM[dest][i] = ETA[dest] = ETA_{min}$. The aforementioned LAM update also happens for C tile fetches and writebacks to take into account their interconnect load as well but without routing optimization. Figure 4.4 shows the LAM for the example in Figure 4.3 ahead of routing the transfers, with two example ETA vector states for tiles A_{00} and B_{00}. Following our ETA-based routing algorithm, C_{00} will be fetched from host memory (since it is only available there), B_{00} will be fetched from gpu_2 since $\max(LAM[0][2], ETA\_B_{00}[2]) + t_2^0 < \max(LAM[0][h], ETA\_B_{00}[h]) + t_h^0$, and A_{00} will be fetched from the host since $\max(LAM[0][h], ETA\_A_{00}[h]) + t_h^0 < \max(LAM[0][1], ETA\_A_{00}[1]) + t_1^0$. ETA-based routing provides the optimal decision based on past and current knowledge (e.g. load and bandwidth) for the fetch of each RONLY tile.

4.2.5 Optimizing RONLY tile transfers with batching

The standard approach for communication routing in previous work [9, 57, 58, 157] is to dynamically optimize the route of a *fetch* task individually, when the task is scheduled. On the other hand, our static schedule which is constructed ahead of execution allows us to batch

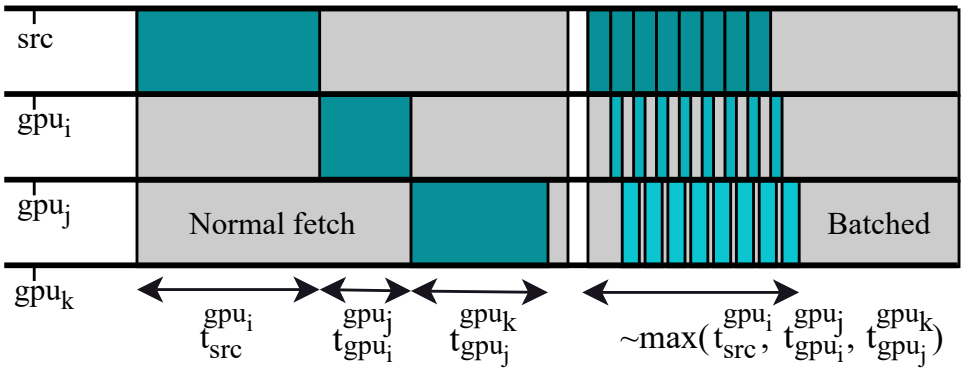


Figure 4.5: An example of performing three fetches of the same data to three GPUs either one by one (left) or with a simultaneous broadcast-like batched fetch (right) that uses $p = 8$ sub-transfers to overlap the process in a pipelined manner. Batching all fetch operations together results in the same fetch cost for gpu_i, but considerably decreases the fetch costs for gpu_j and gpu_k.

tile transfers to different GPUs, with a broadcast-like *fetch* operation to multiple GPUs, e.g. $fetch_{in\_loc}^{gpu_i,gpu_j,\cdots}(T)$. Figure 4.5 shows an example of how a batched fetch works; we split the transfer of the same tile to three locations into p (default = 8) smaller transfers and overlap them internally in a pipelined manner. This decreases the total cost of $fetch_{src}^{gpu_i,gpu_j,gpu_k}(T)$ from $t_{fetch} = t_{src}^{gpu_i} + t_{gpu_i}^{gpu_j} + t_{gpu_j}^{gpu_k}$ to $t_{fetch} \approx \max(t_{src}^{gpu_i}, t_{gpu_i}^{gpu_j}, t_{gpu_j}^{gpu_k})$, considerably decreasing the fetch cost for all but the first transfer destination (gpu_i). This optimization has a greater impact as the number of GPUs increases due to more data-sharing between GPUs. For example, for a 4-GPU system the tiles of matrix A are shared between 2 GPUs and require $fetch_{src}^{gpu_i,gpu_j}(A_T)$ operations, but on an 8-GPU system they are shared by 4 GPUs and require $fetch_{src}^{gpu_i,gpu_j,gpu_k,gpu_l}(A_T)$ operations. We apply this to all RONLY tiles of A and B.

4.2.5.1 Batched-fetch routing

In addition to the decreased fetch cost, batching RONLY-tile fetches is beneficial to communication routing, as it opens up additional route options for each transfer. In batched fetches, the pipeline order is not important (e.g. $fetch_{src}^{gpu_i,gpu_j}(T) \approx fetch_{src}^{gpu_j,gpu_i}(T)$), since the resulting fetch costs are balanced for all destinations. We therefore apply the LAM ETA-based routing of Section 4.2.4.2 in the following way: when a batched-fetch operation $fetch_{src}^{gpu_i,gpu_j,\cdots}(T)$ is scheduled for routing, we examine the individual steps of the composed operation, (e.g. $fetch_{src}^{gpu_i}$, $fetch_{gpu_i}^{gpu_j}(T), \ldots$) and apply the LAM ETA-estimation algorithm iteratively, for all possible order combinations, selecting the order $gpus\_best\_order$ that results in the minimum ETA. Then, we update all intermediate LAM links and all tile ETA destinations based on the ETA_{min} of the selected order and schedule the batched-fetch transfer ($fetch_{src}^{\{gpus\_best\_order\}}$).

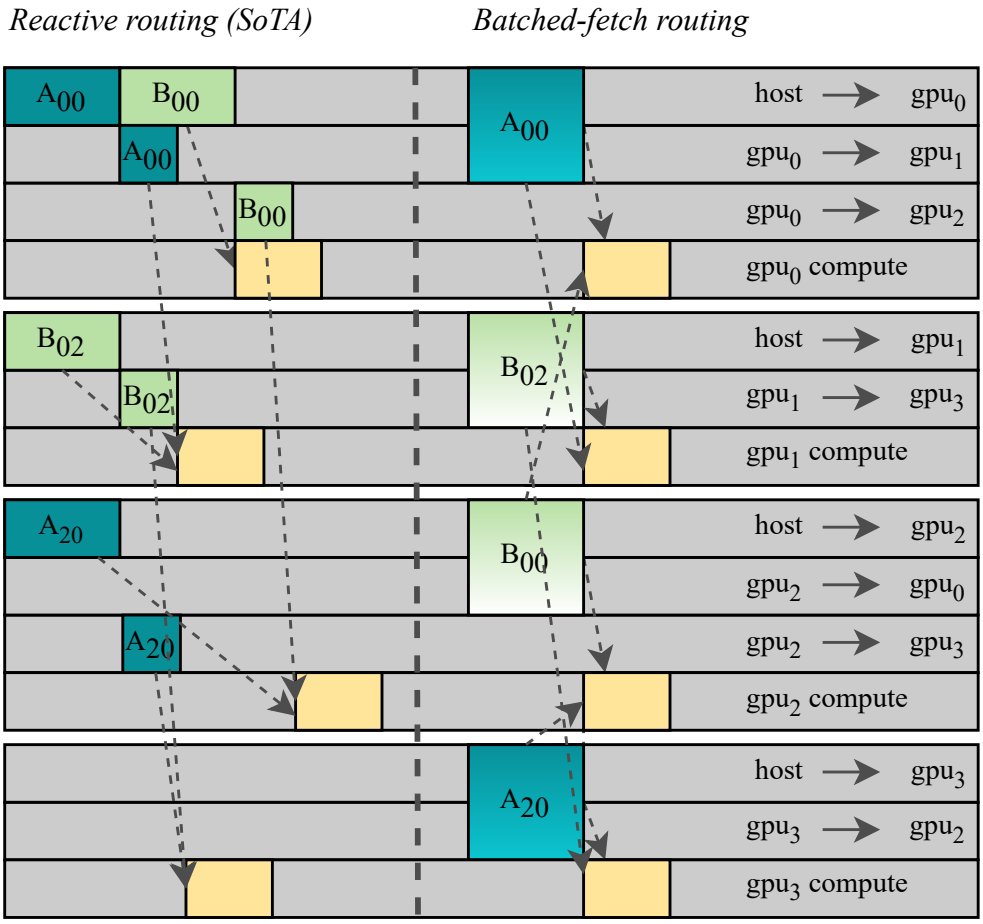


Figure 4.6: An example of routing the first GEMM sub-problem on each GPU using a) reactive routing (left) employed by the SoTA and b) ETA-based proactive routing combined with RONLY batched-fetch (right) used in this work. Reactive routing optimizes the effective bandwidth by using faster links whenever possible, but results in imbalanced interconnect usage, streams becoming idle, being blocked by transfer dependencies, and varied GPU *compute* start times. On the other hand, our approach balances interconnect usage, mitigates idle streams by internally pipelining transfers, and results in a simultaneous start for *compute* in all GPUs.

To show the importance of this optimization in multi-GPU GEMM routing, Figure 4.6 compares the reactive routing employed by previous work against our ETA-based proactive routing + RONLY-tile batched-fetch for the first 4 scheduled sub-problems (one on each GPU), excluding C tile fetches from the pipeline (more on this in Section 4.2.6). Our approach significantly reduces GPU idle time (45% lower on average, 60% best case), overlaps all communication streams internally, avoids blocking due to transfer dependencies, and provides a perfectly load-balanced execution in all GPUs.

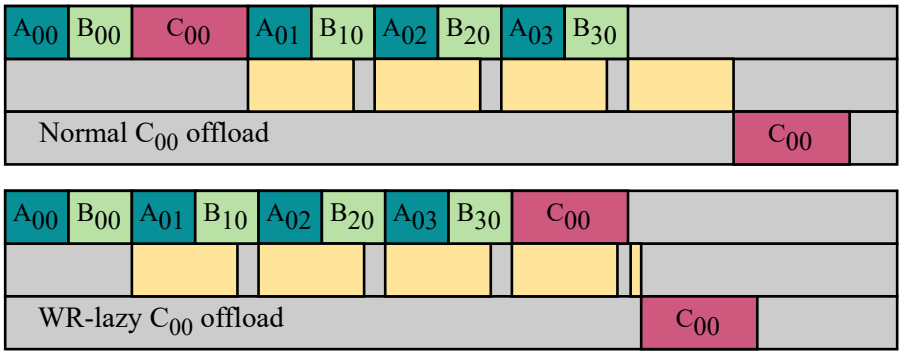


Figure 4.7: Scheduling dependencies and computing four sub-problems on C_{00}, with a normal offload (top) or by using the WR-lazy fetch approach (bottom). WR-lazy fetching reduces GPU idle time, removing C_{00} from the input dependencies of the first sub-problem on the cost of a lightweight extra computation before writing back the result.

4.2.6 Optimizing WR tile transfers with lazy fetching

Our routing and batching optimization discussed in the previous subsections targets the RONLY tiles, which are fetched in multiple GPUs. On the other hand, the WR tiles of the C matrix are exclusive to each GPU (see Figure 4.1) and constitute unmitigable fetch volume, which in the worst-case (nothing cached) scenario for square matrices can reach 1/3 of the total fetch volume. As the batched-fetch optimization does not apply in this case, we opt to limit GPU idle time by *delaying* the transfer of the C tiles, to achieve better computation-communication overlap. To achieve this, we decompose the original GEMM operation of Equation 4.1 into:

$$C' = a \cdot A \times B\ (GEMM),\ C_{out} = b \cdot C_{in} + C'\ (AXPY)$$

The original GEMM operation is decomposed to 1) a GEMM operation without an input matrix C_{in} (equivalent to a GEMM with $b = 0$) and 2) a lightweight addition operation that accumulates the result of the first to the output matrix C_{out} right before writeback (equivalent to an AXPY operation with $alpha = b$). In this way, we decouple the computation-heavy part of the GEMM kernel ($a \cdot A \times B$) from the C_{in} input dependency. Figure 4.7 shows an example of how this changes GEMM offloading for C_{00}. The dependencies of this first subproblem (A_{00}, B_{00}, C_{00}) change to (A_{00}, B_{00}), resulting in lower GPU idle time, as we lazily fetch(C_{00}) at the end of the GEMM computation and update it with an AXPY operation before writing it back. This optimization is beneficial to problems where the C matrix initially shares the location of either A or B. In all other cases, it does not improve the performance, as the input tiles A_T, B_T, C_T use different streams when fetched, and their transfers are therefore overlapped. Moreover, to enable this optimization, an additional buffer memory of $sizeof(C)/gpu\_num$ is required per GPU, as

Algorithm 1: The static schedule algorithm

```
   Data: GEMM params (A, B, C, M, N, K, A_loc, B_loc, C_loc)
 1 if (new params) then
 2     SP[num_sp] ← decompose2D(T = T_min, gpu_num, params)
 3     if (A_loc == C_loc OR B_loc == C_loc) then
 4         SP.adjust_SPs_WRLAZY()
 5     SoftBuf ←assertMemRequirements(SPs)
 6     LAM = {0}, sched_sp = 0
 7     Runtime task queue RTQ = []
 8     while (sched_sp < num_sp) do
 9         for (gpu_i in gpu_num) do
10             currSP = select_SP(gpu_i, SP, LAM)
11             tasklist = split_to_tasks(currSP)
12             for (task in tasklist) do
13                 if (task is fetch) then
14                     task.optimize_routing(LAM)
15                     LAM.update_load(task)
16                 RTQ.append(task)
17             sched_sp ← sched_sp + 1
18 for (task in RTQ) do  task.fire() ;
19 sync_GPUs()
```

both C'_T and $C_{T,in}$ need to be stored before computing and writing back $C_{T,out}$. We therefore selectively apply this optimization only in cases where $A_{loc} == C_{loc}$ OR $B_{loc} == C_{loc}$.

4.2.7 The static schedule

Finally, we provide an end-to-end GEMM implementation that combines the described optimizations to an algorithm as shown in Algorithm 1. The first part of the algorithm (lines 1-17) runs every time a GEMM routine is invoked with a new set of $params$ (input, problem size, matrix locations), calculating an optimized runtime task queue RTQ for that problem. First, the GEMM operation is decomposed to sub-problems (in line 2), which are in turn assigned to devices and adjusted for the WR-lazy algorithm if this is beneficial for the problem $params$ (in line 3). Then, an iterative part runs (in lines 8-17), selecting sub-problems in devices with a round-robin order until all sub-problems have been scheduled. The sub-problem order per GPU is defined by a cost function $select\_SP$ (in line 10) that returns the *optimal* sub-problem based on the current schedule state. After a sub-problem is selected, it is split into tasks (in line 11) as described in Section 4.2.2. The fetch tasks are optimized (in lines 14-15) as described in Sections 4.2.3,4.2.4, and 4.2.5 and each task is enqueued in RTQ (line 16). After this part completes, the RTQ for this set of $params$ is stored internally and reused for all subsequent problems using the same

$params$ during a program's lifetime. The second part of the algorithm (lines 18-19) simply iterates over the RTQ and fires all tasks in their corresponding streams and GPUs.

4.2.7.1 Optimizing sub-problem scheduling order

It is well accepted that the order with which sub-problems are scheduled to GPUs is important because it affects 1) communication routing and 2) idle GPU time [9, 57, 161]. A common technique to optimize the sub-problem order is to prioritize the sub-problems with the minimum fetch operations [57]. We use a similar technique for $select\_SP$, but instead of favoring the minimum fetch operations, we use *ETA estimation* for these fetches, by leveraging the LAM, in combination with the tile dependencies of each sub-problem, as described in 4.2.4.2. This method is load-aware and results in the minimum amount of idle time for *compute* tasks, since it prioritizes the ones whose fetch dependencies are expected to be satisfied earlier.

4.3 Evaluation

In this section, we evaluate our GEMM routine implementation performance and compare it with the state-of-the-art. First, we use a common square GEMM dataset to compare the performance of our implementation against existing libraries and analyze the performance contribution of each optimization introduced in this work. We then expand the dataset with non-square matrices and a variety of data placements to show the performance robustness of our approach. Finally, we discuss our decision to exclude memory constraints from our implementation design and describe how we extend our implementation to run such cases without sacrificing performance.

4.3.1 Experimental Setup

4.3.1.1 Testbed

For the performance evaluation, we use an NVIDIA HGX system, which is part of the acceleration nodes of the Karolina HPC cluster [85], and is described in Table 4.1. Each node consists of 8 NVIDIA A100 GPUs connected with an advanced inter-GPU grid based on NVLink3.0 and NVSwitch2.0, that enables simultaneous bidirectional point-to-point communication between all GPUs with an aggregate bandwidth of 4.8 TB/s (600 Gb/s bidirectional per GPU). The GPUs are connected to the host memory via PCiE with an average bandwidth of 96 Gb/s (12 GB/s per GPU) for all CPU-GPU communication. The GPU clock frequency of the A100 GPUs is tuned for higher energy efficiency, resulting in 12% less peak performance (17.2 vs 19.5 TFlops per A100 GPU).

Table 4.1: The NVIDIA HGX testbed characteristics.

Karolina GPU	CPU	GPU
Computation:	2 x AMD Zen 3, 7763 CPU 128 cores @ 2.45 GHz	8 X NVIDIA A100 FP peak 17.2* TFlop/s DP peak 17.2* TFlop/s
Memory:	1TB DDR4	40 GB HBM2 1.56 TB/s
Interconnect:	PCIe Gen4 x16	NVLink3 / NVSwitch2
Compiler:	g++ 11.2.0	CUDA 12.2
Opt. flags:	-O3	-O3, -arch=sm_80

4.3.1.2 Benchmark methodology

We use the following methodology for all experiments for our implementation and all compared libraries. We perform 10 warm-up runs followed by 100 timed iterations for each GEMM problem size and report the median time/performance of these runs. For time measurements we use `clock_gettime` with device synchronization (`cudaDeviceSynchronize()`) after each iteration (e.g. no inter-loop overlap of GEMM calls). Each matrix is initially stored in a single memory location, to maintain compatibility with the BLAS API standard and to be in line with previous multi-GPU BLAS libraries which also use this data layout. For host memory matrices we use interleaved memory across NUMA nodes, to achieve a balanced CPU-GPU bandwidth between GPUs. We initialize all matrices with random values before execution, and then pin them to memory, to enable asynchronous CUDA calls. GPU caches/buffers are allocated once and reused by subsequent iterations. We flush these buffers after each iteration. Finally, all benchmarked libraries use the same cuBLAS-11 single-GPU `cuBLAS{Dtype}GEMM` routines at their backend.

4.3.1.3 Dataset

For performance evaluation, we use two datasets: a *regular* dataset with square problems, as also reported in related work [57,68,157], and an *irregular* dataset which extends the regular one with mixed initial locations for the matrices, and extra fat-thin and thin-fat problems that divert from the usual GEMM communication/computation ratio [9]. For the *regular* dataset, we select 12 problem sizes ($M_{sq} = N_{sq} = K_{sq} = (5120 \xrightarrow{\text{step=1024}} 16384)$) that are communication-bound on our testbed, based on their operational intensity, and 7 *large* problem sizes ($M_{sq} = N_{sq} = K_{sq} = (20480 \xrightarrow{\text{step=2048}} 32768)$) that are expected to be computation-bound. We run the selected problem sizes with two configurations. In the first configuration, all matrices initially reside on the CPU memory (h, h, h), therefore we expect the major bottleneck to be the PCIe bandwidth. In the second configuration, all matrices initially reside in the memory of gpu_0

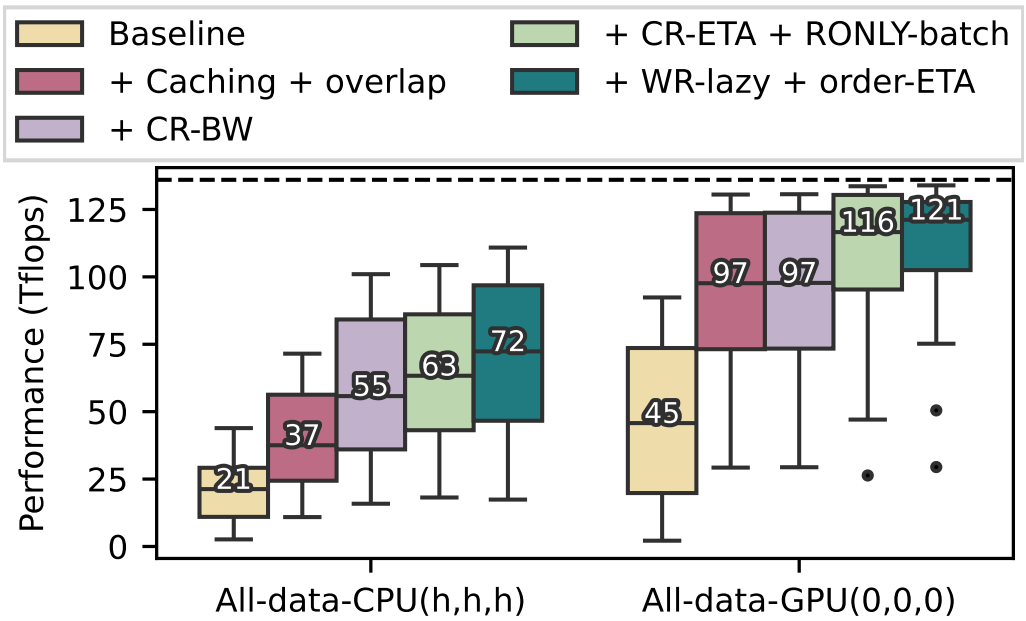


Figure 4.8: The performance of each optimization described in Section. 4.2 for FP64 square GEMM (M=N=K) using all 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line), for the regular dataset of Section 4.3.1. The two clusters correspond to different configurations. Optimizations listed on the legend are applied incrementally left-to-right (yellow = Baseline, blue = all optimizations enabled). Each optimization mitigates a different bottleneck of multi-GPU GEMM, resulting in increased performance in both cases regardless of the differences in communication pattern, overlap, and load balance because of the initial placement.

$(0, 0, 0)$, therefore transfers can directly use the NVLink. The *irregular* dataset is described in Section 4.3.5.

4.3.2 Evaluation of performance optimizations

We first evaluate the optimizations described in Section 4.2 incrementally, to assess how each contributes to total performance. Figure 4.8 shows the performance of FP64 GEMM using 8 GPUS for the *regular* dataset, using our implementation with our optimizations applied incrementally. We note that the optimizations that are designed to work together (1) caching and overlap, 2) ETA-based communication routing and RONLY-batch fetches, and 3) WR-lazy fetches with ETA-based ordering) are also evaluated in pairs. As the baseline, we use a naive implementation of SUMMA decomposition to GPUs (Section 4.2.1) without any optimizations, and the speedup of each bar is calculated with respect to the bar at its left, assessing the impact of 'stacking' an optimization. First, minimizing communication volume with caching, together with overlapping communication with computation offers a performance improvement of almost $2\times$. Adding

BW-based communication routing further improves performance by 1.33x for the (h,h,h) case by favoring faster GPU-GPU transfers, but has no effect when data are already in gpu_0, since all GPU-GPU connections have equal bandwidth. Swapping the routing to ETA-based routing, paired with batching fetches of RONLY tiles improves performance for both data configurations by 1.18x on average, by improving routing, increasing overlap and reducing GPU idle time. Finally, WR-lazy fetching, together with an improved ETA-based order selection for firing subproblems results in an additional 1.1x speedup due to reduced GPU idle time.

4.3.3 Comparison with state-of-the-art

Next, we compare our implementation against the state-of-the-art multi-GPU libraries that attain the highest GEMM performance, by performing weak scaling experiments for the regular dataset using the full node (8 GPUs) of Table 4.1. In particular, we evaluate XKBLAS [57] and PARALiA [9] and exclude previous approaches that they outperform [68, 157]. We also evaluate cuBLASXt [124], as the state-of-practice library, despite its inferior performance [9, 57, 157]. We evaluate GEMM FP64 (double) and FP32 (float) performance. We note that our implementation also supports FP16 (half) precision, but there are no previous multi-GPU libraries that support FP16 for comparison, so we omit this from our results.

Figure 4.9 shows the performance of this work against the state of the art for FP64 GEMM using 8 GPUS for the *regular* dataset. For the case where all matrices initially reside on the host memory, which is bound by the PCIe bandwidth, our work offers high performance to smaller problem sizes. On average, our work outperforms cuBLASXt, XKBLAS and PARALiA by 3.42×, 1.4× and 1.31×, respectively. For the case where the data initially reside on a GPU, and PCIe transfers are avoided, PARALiA results in high overheads for smaller problem sizes, and XKBLAS offers lesser and non-robust performance because of load imbalance. Our implementation mitigates both types of overheads effectively, outperforming cuBLASXt, XKBLAS, and PARALiA by 10.3x, 1.23x and 1.26x, respectively, on average.

On a similar note, Figure 4.10 shows the performance for FP32[2] GEMM. We note that the peak performance is the same for FP32 and FP64, as only `cuBLASDGEMM` (FP64) internally utilizes the FP64 tensor-cores of the A100 GPUs, unlike `cuBLASSGEMM` (FP32). Coupled with half the communication volume for FP32 transfers, this results in less communication-bound problems for the same dataset. While the performance of previous approaches increases somewhat faster than FP64 with problem size, XKBLAS still faces imbalance and PARALiA faces high overhead issues. Our approach, on the other hand, adapts well to the new communication/computation ratio for all problem sizes, approaching peak performance faster and still outperforming cuBLASXt,

[2]not to be confused with TF32

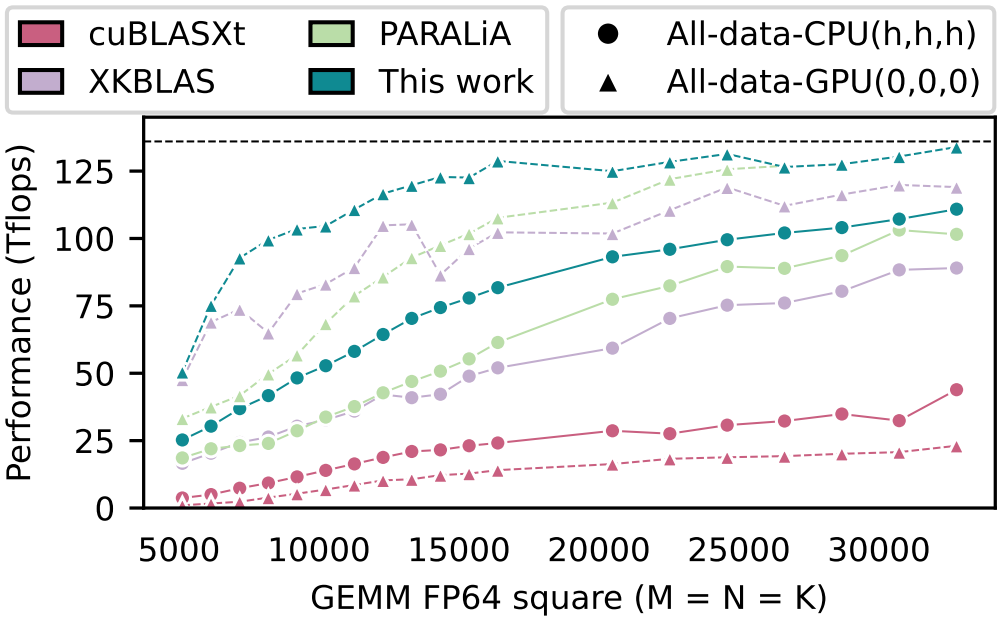


Figure 4.9: The square GEMM (M=N=K) FP64 performance for the regular dataset of Section 4.3.1 for 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line). Our approach offers robust performance regardless of the data placement, avoids imbalance, and outperforms all previous approaches, being more effective in communication-bound problem sizes (12 leftmost data points).

XKBLAS and PARALiA by 2.7x, 1.37x and 1.28x for the *all-data-CPU* case and 10.8x, 1.42x and 1.31x for the *all-data-GPU* case, respectively.

4.3.4 Strong-scaling analysis

We then present a comprehensive strong-scaling analysis to evaluate the performance efficiency of our approach as we increase the number of utilized GPUs for a given problem size. We compare the performance of our approach against the cuBLASXt, XKBLAS, and PARALiA. We note that GPUs in our NVIDIA HGX testbed partially share the PCIe bandwidth in pairs (GPU 0 with GPU 1, GPU 2 with GPU 3, etc.), therefore the peak CPU-GPU bandwidth is the same when using 4 and 8 GPUs (GPU-GPU bandwidth is not affected). We employ the placement that maximizes bandwidth for any number of GPUs, ($1\ GPU \rightarrow [0]$, $2\ GPUs \rightarrow [0, 2]$, $4\ GPUs \rightarrow [0, 2, 4, 6]$), however we highlight that problems that utilize the PCIe (e.g. all data on the CPU) are less communication-bound when using ≤ 4 GPUs. Therefore, scaling from $4 \rightarrow 8$ GPUs becomes more challenging than $1 \rightarrow 2$ and $2 \rightarrow 4$.

Figure 4.11 shows the performance of GEMM FP64 for three different square problems and two different data configurations, one where all data initially reside on the host memory and

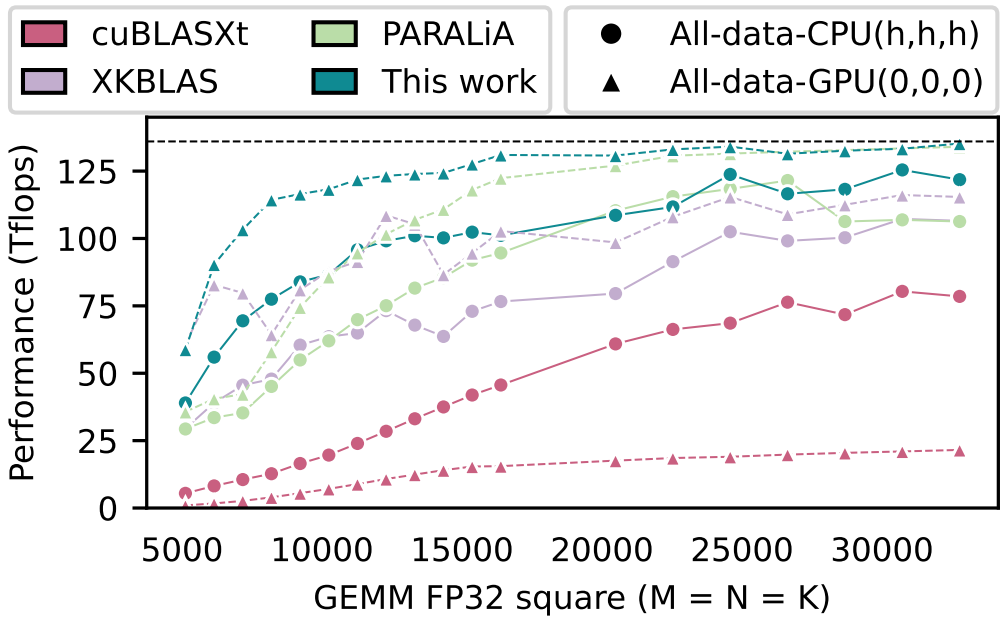


Figure 4.10: The square GEMM (M=N=K) FP32 performance for the regular dataset of Section 4.3.1 for 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line). Using FP32 results in more compute-bound problems due to half the communication volume, coupled with the same FP32 performance peak. Our approach adjusts to the new ratios better than previous libraries, reaching the peak faster and providing superior performance for all problems.

one where all data initially reside on GPU memory, for 1, 2, 4, and 8 GPUs. XKBLAS encounters memory errors on 1 and 2 GPUs for the larger problem size of $M = N = K = 32K$ and fails to complete the execution. First, in the scenario where all data initially reside on host memory, with a matrix size of $[8K, 16K, 32K]$, the speedup on $(2, 4, 8)$ GPUs is $[(1.8, 2.5, 2.2), (1.9, 2.9, 2.1), (1.9, 3.2, 3.1)]\times$ for cuBLASXt, $[(1.8, 3.0, 3.6), (2.2, 4.1, 5.7), (-, -, -)]\times$ for XKBLAS, $[(1.6, 1.8, 2.5), (1.8, 2.9, 4.0), ([1.9, 3.8, 6.4)]\times$ for PARALiA and $[(1.7, 3.0, 3.6), (2.0, 3.5, 5.3), (2.2, 4.2, 7.5)]\times$ for our implementation. In general, 1) our implementation has similar single-GPU performance with PARALiA but offers better scalability, and 2) similar scalability with XKBLAS, but with a $[2.4, 2.1, 1.1]\times$ better single-GPU performance baseline. In the more compute-bound scenario where all data reside on GPU memory, for a matrix size of $[8K, 16K, 32K]$, the speedup on $(2, 4, 8)$ GPUs is $[(0.6, 0.3, 0.15), (1.1, 0.8, 0.62), (1.9, 1.4, 1.3)]\times$ for cuBLASXt, $[(1.8, 2.7, 3.8), (2.3, 4.5, 5.0), (-, -, -)]\times$ for XKBLAS, $[(1.9, 2.8, 3.1), (2.1, 4.0, 6.5), ([2.0, 3.9, 7.8)]\times$ for PARALiA and $[(1.9, 3.5, 5.8), (2.0, 4.0, 7.4), (2.0, 3.9, 7.8)]\times$ for our implementation. In this scenario, all libraries have similar single-GPU performance baselines since there are no transfers, and use the same computation back-end (`cuBLASDgemm`). We note that cuBLASXt faces a scalability break on multiple GPUs in this scenario, We attribute this

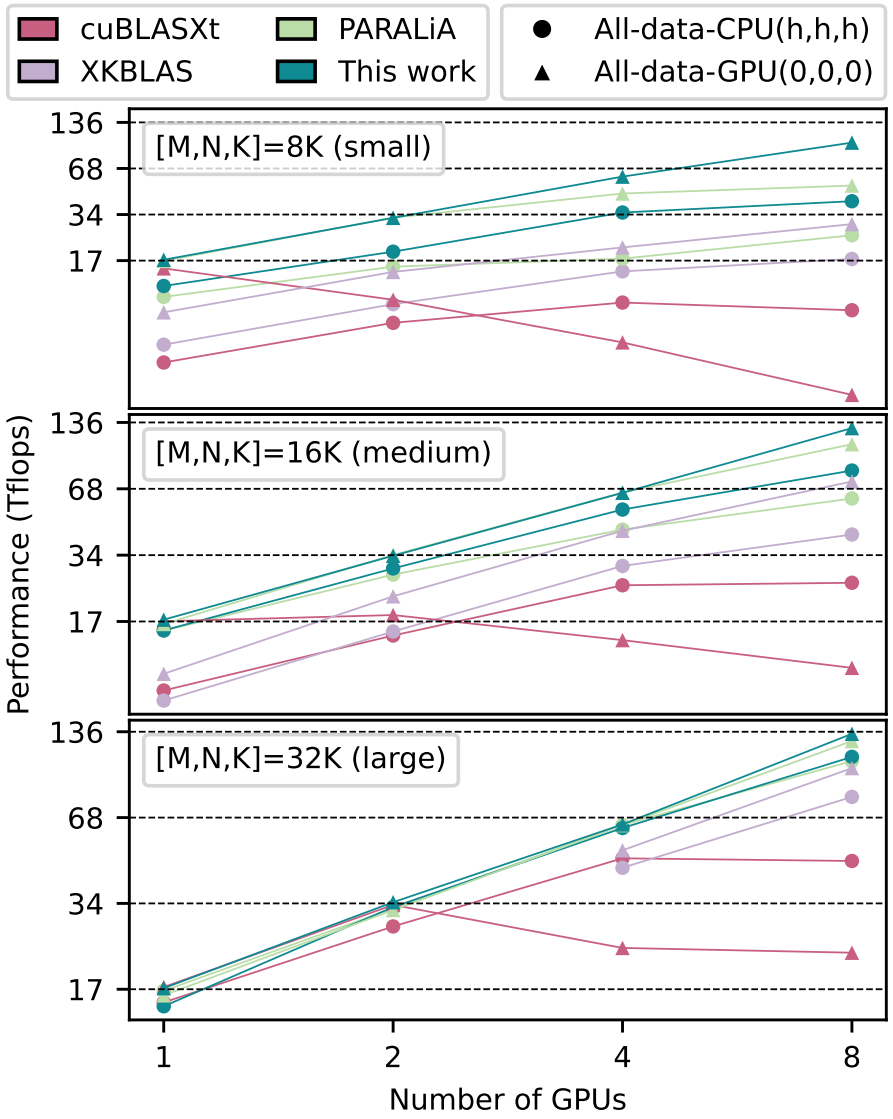


Figure 4.11: Strong scaling analysis of three square GEMM (M=N=K) FP64 problem sizes for two data placements and variable number of GPUs on our NVIDIA HGX testbed (y-axis in log scale, system {1,2,4,8} GPU peak = dashed lines). Our approach provides the best performance for all configurations, and scales better than state-of-the art libraries as the number of GPUs increases, especially for the smaller more communication-bound problems.

to inefficient communication routing that passes through the PCIe instead of using the much faster NVLink [9, 157]. Regardless, our approach achieves higher speedups than XKBLAS and on par with PARALiA for the medium and large, compute-bound problem sizes. For the small, communication-bound problem, our approach shows the best scalability. This is because our

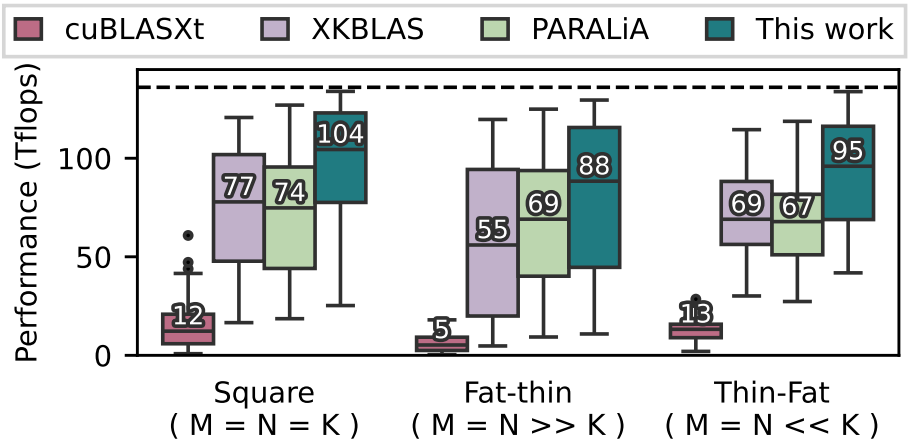


Figure 4.12: Comparison of GEMM FP64 performance robustness against the state-of-the-art, using the expanded *irregular* dataset, divided in three clusters, according to the matrix shapes. Our approach outperforms all existing libraries, regardless of problem irregularity and data placement, providing a uniformly superior solution for multi-GPU GEMM.

communication optimization, combined with lightweight scheduling, directly targets and effectively enhances performance and scalability.

4.3.5 Performance robustness under irregular problems

Finally, to confirm the robustness of our approach across irregular GEMM problem characteristics, we extend the *regular* dataset with three additional initial matrix location configurations: $(4, 2, h), (h, h, 0), (4, 2, 7)$, and two irregular problem shapes (fat-thin and thin-fat GEMM). For fat-thin problems, we use $(M_{fat} = N_{fat} = (16384 \xrightarrow{\text{step=4096}} 40960), Kthin = \frac{M_{fat}}{r}, r \in [4, 8, 16])$ and for thin-fat $(M_{thin} = N_{thin} = (5120 \xrightarrow{\text{step=1024}} 11264), Kfat = M_{thin} \times r, r \in [4, 8, 16])$. This results in an *irregular* dataset of 305 data points.

Figure 4.12 shows the GEMM FP64 performance of cuBLASXt, XKBLAS, PARALiA and our work on the *irregular* dataset, categorized based on the problem shape (square, fat-thin, and thin-fat). cuBLASXt is the slowest implementation for all problem shapes, and its performance degrades further on the irregular dataset due to the diverse initial matrix placements. XKBLAS, on the other hand, performs well on square and thin-fat problems for the various placements, but for fat-thin matrices, the much larger C matrix creates WR-communication slowdowns. PARALiA offers better performance for all shapes, consistent with its targeted performance robustness, but it performs worse than XKBLAS on square and thin-fat problems. Finally, our work achieves better performance for all problems in the irregular dataset, on average outperforming cuBLASXt, XKBLAS, and PARALiA by 11.8x, 1.45x, and 1.37x (8.7x, 1.35x, 1.4x for square problems, 17.6x, 1.6x, 1.28x for fat-thin problems and 7.31x, 1.38x, 1.42x for thin-fat problems), respectively.

CHAPTER 5

Literature review

In this chapter, we provide a comprehensive overview of previous approaches and methodologies relevant to this thesis. Figure 5.1 shows a high-level overview of the research concepts that we explore in this thesis, split in two categories. First, we discuss BLAS optimization across different computational architectures, including multi-core, GPU, hybrid, distributed, and multi-GPU systems. We go through the evolution of BLAS autotuning, highlighting key approaches to internal parameter tuning, communication optimization and hybrid workload selection, and finish with literature specific for GEMM decomposition relevant to our last work PARALiA-GEMMex [8]. Second, we examine performance modeling techniques (yellow) relevant to our work, exploring BLAS kernel performance models and GPU & distributed communication modeling. Figure 5.1 also outlines the order in which we explored these concepts, based on their relevance to our subsequent publications: CoCoPeLia [7], PARALiA [9] and PARALiA-GEMMex (Uncut-GEMMs) [8].

5.1 A brief history of BLAS optimization

5.1.1 The birth of BLAS

The concept of Basic Linear Algebra Subprograms (BLAS) is first introduced 45 years ago when Whaley and Dongarra [101] establish the foundational level of level-1 BLAS routines. Their work defines essential vector operations, such as vector addition, dot products, and scaling, specifically for numerical software written in Fortran, ensuring high portability and efficiency. Four

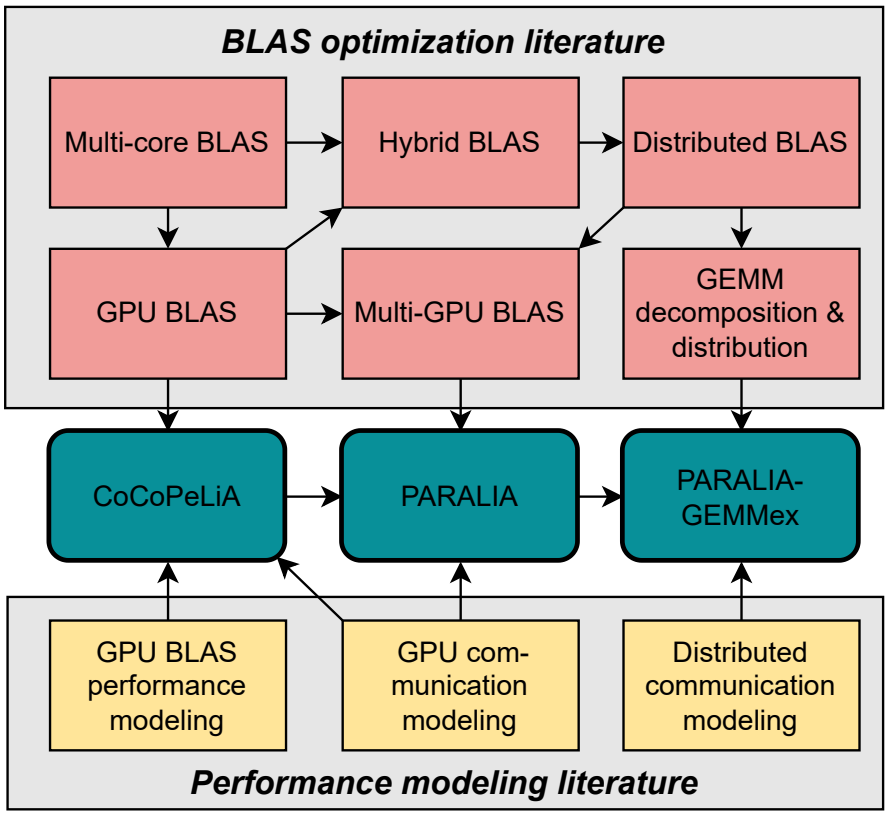


Figure 5.1: A high-level overview of the related work of this thesis.

years later, Dongarra et al. [46,47,49] expand upon this original framework by introducing additional routines, including support for complex numbers and defining level-2 BLAS operations for matrix-vector interactions. On a parallel path, Lawson et al. [48] explore strategies for efficiently implementing linear algebra algorithms on vector pipeline machines, focusing on optimizing performance for dense matrix operation. Then, in response to the growing demands of high-performance computing applications requiring large-scale linear algebra computations, Dongarra et al. [44,45] further extend BLAS to include level-3 routines, with a focus on matrix-matrix operations, particularly the general matrix multiplication (GEMM). Building on this, Kågström et al. [90] provided performance models and benchmarks to demonstrate how using GEMM as a foundational building block for level-3 BLAS can achieve high performance across various architectures. Later, as new vector processors and parallel computing systems emerge in the HPC landscape, BLAS routines are updated to optimize performance on these modern hardware architectures [40]. Finally, these advancements are formalized by Dongarra [39] in the Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard II, which standardizes the implementation of BLAS across different platforms to ensure consistent performance, robustness,

and portability. From that point until now, the list of BLAS routines remains the same with only minor modifications.

5.1.2 Multi-core BLAS optimization

Despite the establishment of a common standard for DLA optimization through BLAS, the emergence of new computing systems and paradigms still requires considerable engineering afford to revisit and fine-tune BLAS implementations. Consequently, after establishing the basic BLAS standard and optimizing BLAS routines for multi-core systems, research focus turned towards *performance portability*.

First steps - autotuning for performance portability: To achieve BLAS portability to new systems, Whaley et al. [29] first introduce the ATLAS project, which implements the Automated Empirical Optimization of Software (AEOS) paradigm to automate the tuning process for BLAS routines. 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. Yi et al. [164] develop POET, a tool for parameterizing and automating empirical tuning of software optimizations. POET performs a guided optimization space exploration for performance-critical code and adjusts internal BLAS parameters automatically, enabling optimization without deep knowledge of the underlying code. Then, Yi et. al [163] further expand this approach by combining POET with ATLAS, revealing important interactions between different transformations and coupling ATLAS *micro-benchmarks* with POET *modeling* to provide a *best-of-both-worlds* autotuning solution. Siek et al. [140] propose a different "build-to-order" approach, where the user specifies their target performance and hardware constraints, and the system then automatically generates optimized kernels with automatic code generation. Goto et al. [60,61] follow a different approach to BLAS optimization: they focus on optimizing the GEMM kernel empirically for each architecture, and deducing the most intensive computations of level-3 BLAS to GEMM operations through tiling. They use this technique to provide GotoBLAS, a level-3 BLAS library, which requires only tuning GEMM in new system to achieve high performance. exceptional performance across various cache-based architectures. Belter et al. [15] propose the view of multiple BLAS operations as a single, optimized kernel, reducing the overhead of composing operations and autotuning. Building on these efforts, Whaley et al. [160] finalize the ATLAS project, setting the standard for BLAS routine autotuning. Wang et al. [158] introduce AUGEM, a framework that automatically generates optimized assembly code for dense linear algebra kernels on x86 CPUs. AUGEM uses template-based optimization techniques to ensure the portability of generated code. Finally, Lang [96] presents a model-based approach for autotuning scientific applications, emphasizing data-aware techniques that

optimize execution and energy efficiency by integrating analytical models for performance and energy estimation.

Following the widespread adoption of auto-tuning as the standard approach for ensuring BLAS portability, recent CPU BLAS libraries can be broadly categorized into open-source and vendor-specific libraries. While both types employ auto-tuning for performance portability, we emphasize on open-source solutions, as the details of vendor libraries are typically proprietary and not accessible to users. Vendor libraries are only listed for reference.

Open-source CPU BLAS libraries: The first open-source library that implements a subset of BLAS routines is the PLASMA project [5, 22]. PLASMA implements parallel tiled algorithms for linear algebra operations with matrices, including Cholesky, LU, and QR factorizations for multicore architectures. Internally, it represents operations as sequences of small, block-wise tasks that are dynamically scheduled based on dependencies and resource availability, allowing out-of-order execution and improved parallelism compared to traditional LAPACK algorithms. More recently, Dongarra et al. [43] describe an updated version of the PLASMA library that uses task-based programming with OpenMP. Regarding libraries that implement the entirety of BLAS, Zhang et al. [128, 162] introduce *OpenBLAS*, an open-source CPU BLAS library for high performance. OpenBLAS uses fine-tuned assembly routines for various processor architectures to achieve performance, and runtime autotuning to adapt to different systems. OpenBLAS is still the most widespread CPU BLAS library used by the scientific computing community. Following the development of OpenBLAS, Van Zee et al. [154, 155] introduce the BLIS framework, which also targets BLAS routines but with a focus on automating portability. BLIS employs flexible, modular tiled kernels that adapt to new architectures and applications, but requires empirical auto-tuning. To overcome this limitation, Low et al. [107] explore the use of analytical modeling as an alternative to empirical auto-tuning in BLIS. They demonstrate that modeling can achieve performance levels comparable to empirical methods, significantly simplifying optimization and ensuring automatic portability. Finally, Van Zee and Smith [141, 152, 153] enhance BLIS by optimizing its internal multi-threaded tiled implementation for matrix multiplication, addressing various data types and further refining the framework's performance in more recent architectures.

Vendor CPU BLAS libraries: The second category, vendor libraries, consist of highly-tuned close-source code that targets the specific architectural characteristics (vector extensions, caches, etc) of their corresponding target hardware. *IBM's Engineering and Scientific Subroutine Library (ESSL)* is the first vendor BLA targeting IBM Power systems [32]. *Intel's Math Kernel Library (Intel MKL)* [33] is the library most widely used in scientific computing and machine learning applications specifically tuned for Intel processors. *AMD's Core Math Library (ACML)* [30] was the first close-source optimized BLAS library for AMD processors. It was replaced by *AMD's Optimizing CPU Libraries (AOCL)* in 2019, which is mostly open-source and mainly targets the

most recent EPYC and Ryzen series CPUs [31]. *Apple's Accelerate Framework* [82] defines BLAS and LAPACK routines optimized for Apple's ARM-based processors and is integrated in macOS. Finally, *ARM's Performance Libraries*, provide BLAS implementations optimized for ARM processors, targeting both embedded and HPC systems [105].

Takeaway The emergence of BLAS, initiated over 45 years ago, has led to extensive literature on BLAS optimization. In multicore systems, optimized BLAS libraries employ autotuning to boost performance and portability across various systems and problem sets.

5.1.3 GPU BLAS

The emergence of GPUs marks a significant milestone in high-performance computing. Due to the data-centric nature of BLAS, and the high operational intensity of level-3 BLAS, they were very suitable for GPU acceleration.

The early age of GPU wizards: In the early days, GPUs were not programmable so accelerating BLAS operations required indirectly utilizing the GPU's native graphics capabilities to perform computational tasks. This approach involved encoding scientific problems as graphics and texture operations to offload them to GPUs. Larsen et al. [97] present a method for performing large matrix-matrix multiplications on low-cost graphics hardware by disguising matrix operations as texture mapping and blending operations. Their approach serves as a proof-of-concept for using GPUs for dense linear algebra (DLA) but faces precision limitations due to the hardware's focus on graphical processing. Rumpf et al. [135] adopt a similar strategy for quantized Finite Element Method (FEM) computations. To address the growing complexity of developing GPU code, Krüger et al. [91] introduce a framework that implements linear algebra operations on GPUs, utilizing a CPU-GPU stream model for basic BLAS operations that exploits GPU parallelism and optimizes communication between the CPU and GPU. Fatahalian et al. [50] focus on a fixed pipeline for matrix operations, mapping matrices to texture elements and employing the GPU's rasterization features when available. Harris et al. [67] explore another fixed pipeline, featuring a cloud dynamics simulation, where they use the GPU's texture memory to store simulation data and implement computational kernels using pixel and vertex shaders. Galoppo et al. [53] emphasize performance optimization by introducing LU-GPU, a framework that leverages GPU-specific memory coalescing and loop unrolling to solve level-3 BLAS problems.

BLAS on programmable general purpose GPUs (GPGPUs): As a response to the growing demand for GPU programmability, NVIDIA introduced General Purpose GPUs (GPGPUs). This considerably eased the implementation of BLAS operations by utilizing CUDA [126] and later, OpenCL [146]. The first example is NVIDIA cuBLAS library [122], an optimized implementation of BLAS specifically tailored for NVIDIA GPUs. cuBLAS is a closed-source library that is

still maintained by NVIDIA, ensures high performance in GPUs and is used up to this date by scientists. In parallel, Volkov et al. [156] revisit BLAS autotuning in GPUs, using empirical tests to tune dense linear algebra operations - with a focus on GEMM - for the NVIDIA Tesla architecture. Their work combines empirical performance testing with algorithmic optimizations like blocking and register tiling to refine BLAS routines for GPU acceleration.

The MAGMA project: As GPUs increased in popularity, the need for BLAS kernels optimized for CPU, GPU and future hybrid systems sparked the MAGMA project initiative [41, 42, 93, 104, 118–121, 148, 149]. MAGMA is a collection of years of research, focusing on redesigning and optimizing dense linear algebra algorithms to fully exploit the capabilities of both multi-core CPUs and GPUs. We first focus on GPU-specific MAGMA work, with work on heterogeneous/hybrid MAGMA presented afterward. Nath et al. [119] introduce the first MAGMA GPU optimizations, implementing techniques like pointer redirecting and recursive blocking to address performance variability in GPU kernels. This adjustment leads to significant speedups in GEMM and SYMV routines. Later, Nath et al. [121] extend these optimizations for the new NVIDIA Fermi GPU architecture. Taking advantage of Fermi's new capabilities, they introduce kernel fusion and data reuse within the GPU's shared memory, improving throughput considerably. In parallel, Li et al. [104] contribute to MAGMA by exploring GEMM auto-tuning for GPUs. Their work uses an autotuning framework, integrated in MAGMA, which dynamically adjusts block sizes and kernel configurations based on the specific GPU architecture. Their framework uses a performance model to guide autotuning and manages to outperform manually-tuned versions. Kurzak et al. [93] further refine this framework for Fermi GPUs, incorporating adaptive algorithms that optimize memory access patterns and kernel launch configurations. Finally, Dongarra et al. [42] further extend the MAGMA library by implementing GPU-specific optimizations such as task-based decomposition and mixed-precision algorithms.

> **Takeaway** With the emergence of GPUs in HPC, BLAS research shifted to adapting routines for the increased parallelism and complex architecture of GPUs. This resulted in specialized libraries and autotuning strategies that target memory efficiency and parallelism, with a focus on computate-intensive level-3 BLAS routines.

5.1.4 Hybrid BLAS: The birth of dynamic workload selection

All GPUs utilized in high-performance computing (HPC) are discrete GPUs, functioning as co-processors alongside the CPU(s). As a result, in addition to GPU BLAS optimization, significant research has been dedicated to *hybrid CPU-GPU BLAS*, where the CPU and GPU are used simultaneously for BLAS computations. Hybrid CPU-GPU BLAS is the initial motivation of the

MAGMA project, where Tomov et al. [5] propose redesigning LAPACK routines to optimize performance for CPU, GPU, and hybrid CPU-GPU execution. To that end, they propose employing *task scheduling* during runtime to manage the parallelism and heterogeneity between the CPU and GPU. Then, Tomov et al. [148] refine this approach, restructuring algorithms to achieve better load balancing and enabling the overlap of computation with communication. Subsequently, Tomov et al. [149] improve MAGMA's dynamic scheduling with a more lightweight task implementation, significantly reducing data transfer and kernel launch overhead. Finally, Dongarra et al. [41] revisit the task scheduling algorithms within MAGMA and implement static runtime management for even more lightweight scheduling.

Parallel to the MAGMA project, several other research efforts have focused on hybrid CPU-GPU BLAS. Luk et al. [108] propose Qilin, a framework designed to exploit BLAS data-centric parallelism on heterogeneous CPU/GPU systems. They split BLAS execution into smaller tasks and employ a mix of adaptive mapping strategies via task graph optimization and model-based static workload selection to schedule them to the CPU and GPU, ensuring good load balance. Similarly, Humphrey et al. [80] present CULA, a framework that accelerates linear algebra routines for hybrid GPU-CPU systems by dynamically partitioning tasks between the CPU and the GPU. Spagnoli et al. [145] later extend this work to sparse linear algebra. Subsequently, Humphrey et al. [79] refine CULA by integrating more recent GPU and CPU optimizations into the framework. In a related approach, but with a different optimization target, Ma et al. [110] propose GreenGPU, which focuses on energy efficiency when balancing workloads between the CPU and the GPU, rather than purely on performance. Tsai et al. [150] then port the concept of tuning block size from CPU BLAS autotuning to hybrid BLAS, specifically optimizing the block size used for decomposing an initial QR factorization problem into sub-problems for execution on CPU-GPU hybrid systems. Bernabé et al. [16] build upon this previous work to provide a complete solution for autotuning BLAS on hybrid CPU/GPU platforms to maximize overall performance. They propose a framework that utilizes performance models to dynamically adjust parameters such as block size and CPU/GPU workload ratio, aiming to minimize execution time. Notably, their approach is the first to incorporate data transfer overhead into the modeling of total performance. Due to the increased complexity of modeling and their diverse autotuning targets, they employ a combination of empirical testing and analytical models to fine-tune their routines.

Takeaway A considerable amount of GPU BLAS research focuses on hybrid CPU-GPU approaches. These approaches use BLAS *domain decomposition*, *task scheduling*, and *autotuning* to dynamically allocate and balance the varying workload between the CPU and the GPU.

5.1.5 Distributed and multi-GPU BLAS

Although this thesis focuses on multi-GPU BLAS, optimizing BLAS for multi-GPU systems bears many similarities to distributed BLAS optimization. Both involve handling parallel workers with discrete memories, therefore introducing the concepts of decomposition, distribution, communication, and scheduling. Unlike multi-GPU, which is a relatively new concept, distributed BLAS literature dates back several years before the existence of GPUs. Thus, this sub-section starts with the early research on distributed BLAS optimization that preceded multi-GPU BLAS optimization and then moves on to recent developments and techniques for multi-GPU systems.

From multi-core to multi-node: The optimization of distributed BLAS branches from multi-core BLAS when Choi et al. introduce PBLAS [27] and ScaLAPACK [26]. Parallel Basic Linear Algebra Subprograms (PBLAS) [27] is a new routine standard that supports pre-distributed data on distinct memories, designed to simplify the parallelization of vector and matrix operations in distributed memory MIMD systems. ScaLAPACK [26] is a linear algebra library designed for distributed memory parallel computers, extending the functionality of BLAS to handle operations across multiple processors and distributed data. With the rapid increase of core and node count in HPC clusters during the next years, multi-core approaches face scalability issues due to the different paradigm they were designed for. To alleviate this, Gunnels et al. [64] proposed the FLAME framework, which uses formal derivation techniques to develop and implement linear algebra algorithms. FLAME focuses on code modularity and maintainability coupled with high performance for sequential architectures through its structured approach to code generation. Later, Chan et al. [24] proposed SuperMatrix, an *out-of-order* scheduling technique specifically designed for SMP and multi-core architectures, which divides matrix operations into smaller, independently executable tasks (similar to Section 5.1.4). These tasks are then dynamically scheduled and executed based on data dependencies, allowing the efficient use of processing resources and better overlap of computation with communication. Quintana-Ortí et al. [133] port the SuperMatrix runtime system to multicore platforms with multiple hardware accelerators by coupling it with FLAME, leading to significantly higher performance due to dynamic scheduling and better memory coherence. Lastly, Ayguadé et al. [12] presented GPU Superscalar (GPUSs), an extension of the StarSs [132] model for parallelizing applications on systems with multiple GPUs, which addresses architecture heterogeneity and memory management while achieving notable performance results.

Distributed task scheduling for scalability: After the initial approaches for distributed systems, dynamic runtime systems and task scheduling optimizations become the prevalent methods for addressing heterogeneity and portability in BLAS. Building upon the idea of SuperMatrix, Augonnet et al. [11] introduce the StarPU runtime. StraPU manages task scheduling and data movement across multicore processors and GPUs, with fine-grained task creation and adaptive

resource allocation, resulting in better end-to-end performance. Then, Augonnet et al. [10] further enhance StarPU with data-aware task scheduling targeting multi-accelerator environments, specifically optimizing data locality and minimizing inter-accelerator communication. Marking the first BLAS-specific approach using these schedulers, Agullo et al. [3] integrate StarPU in MAGMA, which schedules linear algebra tasks across hybrid CPU-GPU systems, using the runtime system to handle the complexities of memory management and task distribution efficiently. Extending this, Haidar et al. [66] present a methodology for unified development of task runtimes in mixed multi-GPU environments, and integrate their takeaways in MAGMA, increasing task scheduling performance.

In parallel, Bosilca et al. [20] develop DAGuE, a distributed DAG engine for task orchestration, efficient data transfers, and load balance. DAGuE targets distributed systems with similar workers, in contrast to StarPU which targets heterogeneity, employing more targeted optimizations. Consequently, this deems it more applicable to homogeneous multi-node and multi-GPU systems that are used in modern HPC clusters. Building on this, Bosilca et al. [18] present DPLASMA, an extension of PLASMA for distributed systems that utilizes DAGuE to optimize scheduling, communication and data locality. In another approach, Song et al. [144] propose a framework for heterogeneous multi-core and multi-GPU systems using a multi-level block cyclic data distribution method, enhancing parallelism and minimizing communication with hybrid tile algorithms and auto-tuning techniques. Later, Song and Dongarra [143] extend this work by focusing on heterogeneous GPU-based clusters, adding multi-level data partitioning and dynamic scheduling to address communication bottlenecks and scalability issues. In parallel, Gautier et al. [56] introduce XKaapi, a runtime system designed for data-flow task programming that manages task execution and data movement on heterogeneous architectures with a data-centric model. Finally, building upon all these approaches, Bosilca et al. [19] present the PaRSEC framework, which utilizes hierarchical scheduling and task partitioning to manage tasks across heterogeneous and distributed computing resources efficiently. The framework enables dynamic adaptation to varying computational loads and resource availability, enhancing scalability and performance for complex workloads. Then, Wu et al. [161] extend PaRSEC by incorporating hierarchical DAG scheduling to optimize task execution in hybrid distributed systems. Their approach refines the framework's ability to handle more complex dependencies and balances computational workloads, further improving scalability and efficiency.

Takeaway Multi-GPU BLAS shares many principles with distributed BLAS as both involve managing parallel workers with discrete memories. Initial research in both fields revolves around advanced runtime systems that optimize task scheduling and data locality for multi-node clusters, potentially with accelerators.

The performance road: single-node multi-GPU BLAS libraries: After task schedulers and runtimes became the standard for distributed, hybrid, or accelerated BLAS optimization, multi-GPU BLAS research split into two categories, with both targeting level-3 BLAS only. The first category focuses on single-node, multi-GPU approaches, aiming to maximize performance within a single compute node. These approaches concentrate on optimizing memory management, data distribution, and scheduling at the hardware level to exploit the full potential of the available GPUs within a node. This begins with NVIDIA's cuBLASXt library [124], an extension of cuBLAS that supports multi-GPU execution with data residing on the host or any of the GPUs, and its LAPACK-compatible wrapper NVBLAS [125]. However, cuBLASXt uses a simple round-robin scheme to distribute 2D tile-based sub-kernels to devices, resulting in unnecessary communication. As an answer to this, Wang et al. [157] propose BLASX, a high-performance Level-3 BLAS library for heterogeneous multi-GPU computing that uses dynamic scheduling to optimize load balancing across GPUs. Additionally, BLASX uses a hierarchical hardware abstraction of the discrete GPU memories as a cache hierarchy to improve locality and increase communication throughput. BLASX outperforms cuBLASXt but its performance degrades in more recent clusters that differ from its hardware abstraction. To account for this, Gautier and Lima [57] introduce XKBLAS, a multi-GPU level-3 BLAS library that utilizes lightweight task-based parallelism, implemented using XKaapi [56]. XKBLAS uses topology-aware heuristics, further refined by Gautier and Lima [58], which adapt to system interconnect and GPU characteristics to optimize data transfer routing and maximize data locality. XKBLAS offers significant performance improvement from cuBLASXt and BLASX, particularly for smaller matrix operations where data movement is a critical bottleneck.

The scalability road: distributed multi-GPU BLAS libraries: The second category encompasses multi-node multi-GPU BLAS approaches. These solutions build on the aforementioned distributed schedulers, specifically targeting multi-GPU nodes and addressing large-scale problems, with a focus on scalability and efficient communication rather than optimizing performance for a low number of processes. While multi-node execution is beyond the scope of this thesis, we give a brief overview of the most recent approaches since some of their scheduling optimization is relevant for our last work PARALiA-GEMMex [8]. Influenced by task-based runtimes, Kurzak et al. [92] introduce PULSAR, a distributed programming abstraction optimized for large-scale multi-node environments that enables lightweight scheduling and communication optimization. Agullo et al. [4] extend the StarPU runtime system with an inter-node data management layer to support high-level sequential task-based programming on large HPC clusters, achieving performance comparable to existing MPI-based and task-based implementations. Herault et al. [68] enhance the classical tile-based outer-product GEMM approach with control dependencies for improved data reuse and optimized communication flow, and integrate it in the PaRSEC runtime system to achieve near-peak performance. Gates et al. [54] introduce SLATE,

a library designed to replace ScaLAPACK for modern distributed high-performance systems. SLATE utilizes communication-avoiding algorithms, lookahead panels for overlapping communication and computation, and task-based scheduling, and is implemented as a C++ framework that supports a wide range of BLAS and LAPACK routines. Lastly, Gates et al. [55] refine SLATE to enhance its portability and efficiency in more recent HPC clusters, incorporating advanced heuristics for task management and data locality.

Takeaway Modern Multi-GPU BLAS libraries can be divided into two categories: single-node and multi-node approaches. Single-node approaches follow the BLAS standard, using various software and hardware optimizations with a focus on *performance*. Multi-node approaches, based on either the BLAS or PBLAS standard, prioritize *scalability* by utilizing advanced communication-optimal algorithms.

5.1.6 GEMM decomposition and distribution algorithms

In this subsection, we review key developments in GEMM decomposition and distribution algorithms, which motivated the communication optimization techniques employed in PARALiA-GEMMex (Section 4). Cannon [23] introduces the first parallel GEMM approach for square matrices by employing a cyclic data shift mechanism to distribute computation evenly and reduce inter-processor communication. Dekel et al. [37] further optimize this algorithm for cube-connected processors, improving time complexity through efficiently utilizing processing elements. Fox et al. [51] extend this method to accommodate various matrix shapes and processor grids, employing a block-cyclic decomposition and optimizing communication for hypercube architecture. Berntsen [17] also optimizes GEMM for hypercubes by using a nearest-neighbor communication logic viewing the hypercube as a set of subcubes, reducing communication costs. Other approaches focus on the hardware aspect of their systems. Johnsson [87] develops algorithms that minimize communication time for GEMM on multiprocessor systems, offering communication speedups by effectively using the network's topology. Agarwal et al. [2] design a high-performance GEMM algorithm for distributed-memory systems that reduces communication time by overlapping communication and computation. The PUMMA algorithm [28] adds support for transposed matrices and explores data placement optimization, marking the next step forward in GEMM algorithm development.

Building on top of PUMMA, Agarwal et al. [1] show that a 3D decomposition of GEMM is advantageous when memory is not a constraint, leading to improved performance. The SUMMA algorithm [151] also builds on PUMMA by optimizing communication, adding blocking, and overlapping communication and computation, establishing it as the standard 2D GEMM algorithm. Later, Irony and Tiskin [84] gather all available GEMM algorithms and establish the

theoretical communication lower bounds for distributed-memory GEMM to guide the development of more efficient algorithms. The 2D (SUMMA) and 3D GEMM algorithms were the standard for over 10 years and are still used today for regular problem shapes. Building on top of them, Solomonik and Demmel [142] introduce the 2.5D GEMM algorithm, which balances decomposition between 2D and 3D based on available memory, achieving optimal communication for square matrices. Demmel et al. [38] present a communication-optimal parallel algorithm for rectangular matrix multiplication, minimizing data movement by employing a recursive hierarchical partitioning strategy. Schatz et al. [139] provide a comprehensive taxonomy of distributed-memory parallel GEMM algorithms and discuss extending traditional 2D methods to 3D configurations. Finally, COSMA [94] identifies inefficiencies in the CARMA algorithm related to increased communication and introduces a new approach based on the red-blue pebble game, offering a communication-optimal solution for any matrix shape and processor configuration (M, N, K, p).

5.2 Related performance modeling literature

This section transitions from BLAS methodologies and autotuning to performance modeling towards enabling autotuning. We begin by reviewing the performance modeling literature relevant to GPU BLAS Then, we follow the development of distributed communication modeling from early models to advanced approaches addressing communication-computation overlap, heterogeneity, and parameter autotuning. Finally, we review GPU communication modeling, covering generic transfer models, communication-computation overlap, and benchmarking methods.

5.2.1 BLAS kernel performance modeling

We only provide a brief overview of BLAS kernel performance modeling, since our PARALiA [9] and PARALiA-GEMMex [8] approaches rely exclusively on micro-benchmarks coupled with value-lookup tables for assessing GPU BLAS kernel performance.

Generic GPU performance modeling: We first explore the relevant literature on GPU performance modeling not specifically for BLAS. Ryoo et al. [136] present a methodology that simplifies GPU application optimization by pruning the optimization space with static code metrics to find Pareto-optimal configurations. Schaa and Kaeli [138] examine the design space for multi-GPU systems, highlighting the trade-offs and performance impacts of different design choices. Sunpyo et al. [78] introduce an analytical model based on a kernel's parallel memory requests to estimate GPU performance. Baghsorkhi et al. [13] develop a model that evaluates GPU performance by analyzing kernel utilization of GPU features. Zhang et al. [165] apply an empirical micro-benchmarking approach to gather data on instruction pipelines, shared memory, and

global memory access for performance estimation. Meng et al. [113] introduce GROPHECY, a framework that estimates GPU performance based on CPU code skeletons. Iakymchuk and Bientinesi [81] propose a performance modeling approach that emphasizes memory stalls to better understand and predict application performance on modern hardware. Konstantinidis et al. [89] propose a modified roofline model and a black-box micro-benchmarking method for predicting application performance across different GPUs. Finally, O'Neal et al. [127] introduce HAWLPE, a predictive modeling framework for GPU performance that utilizes performance statistics from current-generation GPUs to forecast the performance of next-generation GPUs, providing an efficient alternative to costly cycle-accurate simulations.

GPU BLAS kernel performance modeling: In addition to general approaches, some literature focuses specifically on BLAS modeling. Luszczek and Dongarra [109] propose a method to reduce tuning time for parallel dense linear algebra routines through partial execution and performance modeling. Meswani et al. [114] present a method for modeling and predicting the performance of high-performance computing applications on hardware accelerators, focusing on improving performance insights and optimization. Peise and Bientinesi [131] explore performance modeling techniques specifically for dense linear algebra operations, offering insights into accurate performance predictions. Cámara et al. [35] provide an empirical model for shared-memory linear algebra routines, aiming to enhance the understanding of performance characteristics in such computational environments. Martell et al. [111] develop a tool for GPU kernel design assistance that uses a semi-empirical linear model to predict critical performance factors, including Global-to-Shared transfer time, Shared-to-Private transfer time, and Processing Units Time.

> **Takeaway** GPU BLAS performance modeling approaches usually employ a mix of analytical modeling and micro-benchmarks, to adapt the generic models to specific architectural characteristics.

5.2.2 Distributed communication modeling

Before addressing the newer developments in GPU communication modeling, we review the established models for distributed system communication prediction that preceded GPU modeling. We begin with foundational models like LogP and its extensions, which provide a basis for understanding communication delays, bandwidth, and processing overheads. We then assess advancements in simulation tools and more recent communication models that improve performance prediction for large-scale distributed systems.

The LogP model family : The birth of the Log models starts with the introduction of the LogP model by Culler et al. [34]. LogP defines four parameters —computation bandwidth, communication bandwidth, communication delay, and coupling efficiency— to better represent real-world parallel systems and facilitate the development of efficient parallel algorithms. Alexandrov et al. [6] show LogP accuracy is limited when long messages are involved and extend the LogP model accordingly to LogGP by incorporating long message modeling. Frank et al. [52] develop the LoPC model, which builds on LogP by including contention costs in parallel algorithms, offering a more accurate performance prediction for cases with irregular communication patterns. Kielmann et al. [88] present a method for efficiently measuring LogP parameters on message-passing platforms, optimizing the process to minimize measurement time and intrusiveness. Moritz and Frank [117] introduce LoGPG, a model that extends LogGP to account for network contention in message-passing programs, providing increased accuracy for estimating performance impacts due to network delays. Ino et al. [83] propose the LogGPS model, an extension of LogGP that includes synchronization costs for long messages, improving performance prediction accuracy. Hoefler et al. [73] present the LogfP model, tailored for small messages in InfiniBand networks that previous models cannot account for. Then, Hoefler et al. [72] introduce an approach for assessing LogGP parameters in modern interconnection networks with low overhead. Hoefler et al. [75] continue with an in-depth analysis of the LogGP model, revisiting and exploring its effectiveness in modern interconnection networks and collective operations. Chen et al. [25] present LogGPO, a communication model designed to predict MPI program performance by accounting for communication-computation overlap. Finally, Hoefler et al. [76] introduce LogGOPSim, a simulation framework for large-scale applications that uses the LogGOPS (LogP, LogGP, or LogGPS) models to estimate distributed system performance.

Communication-computation overlap, heterogeneity and parameter autotuning: In addition to the models mentioned above, many approaches focused on other aspects of distributed communication performance, like communication-computation overlap and heterogeneous communication, or how to improve and automate model parameter autotuning. Goumas et al. [62] explore techniques for minimizing the execution time of nested for-loops using a tiling transformation. This is performed by finding an efficient time hyperplane, overlapping communication and computation, and assigning tiles to processors based on tile space boundaries. Bell et al. [14] improve performance in bandwidth-limited scenarios with one-sided communication and overlap, reducing communication overhead and enhancing bandwidth utilization. Danalis et al. [36] suggest transformations to parallel codes through runtime environment adjustments, to enable low-latency and efficient communication-computation overlap. Sancho et al. [137] perform a quantitative analysis of the benefits of overlapping communication with computation in large-scale scientific applications, utilizing empirical data to demonstrate improvements in execution

times and other performance metrics. Geoffray and Hoefler [59] propose dynamic adaptive routing strategies for high-performance networks, addressing issues such as head-of-line blocking and bisection bandwidth to enhance network efficiency. Hoefler et al. [74] explore collective operation modeling for large scale systems, using benchmarking techniques and synchronization analysis to increase prediction accuracy. Lastovetsky et al. [98] develop a detailed communication model for heterogeneous clusters with switch-enabled Ethernet networks. Later, Lastovetsky and Rychkov [99] revisit the heterogeneous communication challenge, proposing a set of experiments designed to accurately and efficiently derive model parameters for clusters with varying interconnection characteristics. Finally, Lastovetsky et al. [100] combine their previous work in a new P2P communication model for heterogeneous clusters accompanied by a software tool for automated parameter estimation.

Using a different approach, Martinasso and Méhaut [112] introduce a contention-aware model for InfiniBand networks that uses a dynamic contention graph and a linear system of bandwidth distribution to improve prediction accuracy. Hoefler and Snir [77] present a topology mapping strategy for large-scale parallel architectures that uses a graph similarity-based heuristic, considerably increasing prediction accuracy. Hoefler et al. [71] then provide a detailed review of systematic performance tuning through performance modeling for high-performance computing, detailing methodologies for accurate model development and micro-benchmarking. Using an alternative approach, Jain et al. [86] propose using supervised learning based on communication features for application performance prediction, coupled with decision trees and simpler models for contention and task mapping. Zhu et al. [166] develop a communication model for hierarchical Ethernet networks, incorporating asymmetric network properties to enhance communication prediction accuracy. Papadopoulou et al. [129] present a machine-learning approach for predicting the communication time of parallel applications, using a few easily extracted parameters from the application and process mapping. They propose a simple benchmark that feeds a multiple-variable regression model, which is used to predict the communication time of applications for varying process counts. Finally, Zhu et al. [167] re-examine asymmetric communication models for Ethernet networks, emphasizing the need for tuning below the TCP layer to address performance issues caused by network asymmetries in modern systems.

Takeaway Distributed communication modeling is a well-established field, with numerous approaches addressing various system and application-specific nuances. These approaches focus on precise communication time prediction, modeling communication-computation overlap, managing interconnect heterogeneity, and initializing model parameters in new systems.

5.2.3 GPU communication modeling

While GPU-specific transfer modeling is relatively simpler than distributed modeling due to the limited number of parallel workers, communication affects total application performance considerably.

Generic GPU transfer modeling: Gregg and Hazelwood [63] first highlight the lack of transfer consideration in performance reporting, emphasizing the critical role of the data placement and transfer overhead when comparing CPU and GPU performance. Boyer et al. [21] improve GROPHECY [113] by incorporating latency-bandwidth data transfer modeling in their GPU performance predictions. Meswani et al. [115] present a performance-modeling framework that assesses an application's performance compatibility with accelerators like GPUs and FPGAs, considering data transfer throughput and latency. Finally, Riahi et al. [134] compare analytical and machine learning models for predicting data transfer times, recommending a hybrid approach that uses a latency-bandwidth model for large data and machine learning for smaller data transfers.

Modeling communication-computation overlap: A common GPU offload optimization technique is decomposing a problem into smaller sub-problems, and overlapping the communication and computation of different sub-problems in a pipeline manner. This is supported in modern GPUs support via asynchronous communication primitives (CUDA streams, OpenCL queues etc), and includes H2D and D2H transfer overlap with GPU computation (3-way overlap). Gómez-Luna et al. [65] first introduce a modeling approach to optimize CUDA asynchronous data transfers. They use a simple latency-bandwidth model for transfers and an empirical approach for computation time to predict the total overlapped time based on the number of sub-problems and streams. Based on this, they can adjust the number of streams and sub-problems for improved communication-computation overlap. Werkhoven et al. [159] 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. [106] 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.

Benchmarking GPU transfers for performance insights: An alternative to modeling for estimating an application's transfers, and their contribution to offload performance, is through *micro-benchmarks*. With the introduction of unified memory in CUDA 6.0, micro-benchmarks became the preferred method for performance estimation due to their accuracy over generic

modeling. Landaverde et al. [95] first investigate unified memory access in CUDA, emphasizing its role in improving data transfer performance and identifying key factors affecting runtime and acceleration through benchmarking. Li et al. [103] assess unified memory tradeoffs, focusing on memory management and performance implications for heterogeneous computing. Mishra et al. [116] explore unified memory for OpenMP GPU offloading, modifying benchmark codes to evaluate its performance impact when used instead of normal transfers. They conclude that while unified memory offers comparable performance to traditional GPU offloading, it incurs significant overhead with large data reuse.

In addition to unified memory, some work evaluates GPU interconnects when using normal transfers through microbenchmarks. Tallent et al. [147] compare PCIe and NVLink interconnects for deep learning workloads, showing that NVLink generally provides better performance due to its superior bandwidth and lower latency, with PCIe remaining competitive for certain workloads. Li et al. [102] propose a multi-GPU benchmark suite to evaluate modern GPU interconnects, analyzing performance and identifying nuances in bandwidth and latency across different network topologies. Finally, Pearson et al. [130] introduce Comm|Scope, a comprehensive set of microbenchmarks for understanding CUDA data transfer performance, addressing factors such as data placement, interconnect hardware, and system-level optimizations, and offering insights into improving transfer efficiency and application performance.

Takeaway GPU communication modeling typically focuses on estimating transfer bandwidth or overlap performance and can be approached analytically, empirically through benchmarking, or with semi-empirical methods. However, current approaches are either overly tailored to specific tasks or too generic, lacking the precision required for accurate BLAS communication modeling.

CHAPTER 6

Conclusions

In this thesis we aimed to achieve portability, near-optimal performance, and efficient resource utilization for single- and multi-GPU BLAS. GPU BLAS execution introduces a variety of *internal parameters* that can be tuned to each individual BLAS call, and requires important *decisions* for decomposition, communication and scheduling during runtime that effect application performance. Due to the prohibitive complexity of the problem we propose a model-driven approach, where we utilize modeling to estimate performance, communication and overlap characteristics for each candidate system. Then, these models are used for parameter and software autotuning, as well as for choosing which resources to utilize based on the problem and system characteristics of each routine invocation during runtime. Our work consists of three parts: 1) we introduce various performance models for GPU BLAS offload and 2) we provide a high-performance library for multi-GPU BLAS based on insights and autotuning deriving from those models and 3) we implement a GEMM implementation that couples model-driven knowledge with distributed programming techniques to optimize communication and achieve peak performance.

In the first part of our this thesis, we outline that 3-way concurrency is currently not well utilized in BLAS GPU offload libraries, since the efficient split tile size depends on routine, problem, and system-specific parameters. Since part of these only become available during runtime, we propose a) two models for the 3-way overlap offload time as a function of tile size, b) a micro-benchmark approach for initializing the empirical model parameters offline, and c) a runtime tile scheduler for efficient 3-way overlap and data reuse. We combine these into an end-to-end GPU BLAS framework, CoCoPeLia [7], and demonstrate its use for `dgemm`, `sgemm` and `daxpy`; our

evaluation shows that it achieves lower errors than previous approaches and is usable in practice for efficient tile prediction. Furthermore, our BLAS wrapper with runtime tile prediction offers considerable performance improvement over previous offload approaches for all tested routines.

In the second part, we present PARALiA [9], 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 and resource utilization via model-driven auto-tuning. To that end, we expand the modeling of CoCoPeLia [7] for multi-GPU performance prediction and introduce a generic hardware abstraction called `LinkMap`, that accurately estimates interconnect and communication characteristics for each BLAS problem. PARALiA utilizes these models to optimize communication and to perform careful device selection, based on a pre-set optimization target, which can be performance or some energy-related metric, avoiding resource waste. We evaluate our approach 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 our approach outperforms the state-of-the-art BLASX and XKBLAS multi-GPU BLAS frameworks with a (geo)mean improvement of 1.7x and 2.4x 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, our approach is able to achieve high performance coupled with better energy efficiency, with a (geo)mean improvement of 2.5x versus BLASX and 3.4x versus XKBLAS. Finally, our approach adjusts well to a heterogeneous system with different compute capabilities among the GPUs, offering improved performance and superior energy efficiency over BLASX and XKBLAS.

In the last part, we provide an optimized GEMM implementation tailored for efficient execution on multi-GPU compute nodes [8]. Our approach focuses on the mitigation of the communication and scheduling overheads, and load imbalance of multi-GPU GEMM, using a variety of optimization techniques. Our implementation is based on a static schedule, which is constructed ahead of execution, whenever a routine is invoked, and can therefore utilize the specific problem characteristics to minimize communication, increase throughput, maximize overlap, and load-balance communication and computation. We employ a hierarchical problem decomposition and offer a heuristic for tile size selection. We utilize multiple streams to effectively overlap computation and communication, and cache tiles to be reused, avoiding communication where possible. We optimize communication routing by considering the availability of point-to-point links and scheduling tile transfers accordingly to ensure that tiles arrive at their destination GPUs at the earlier possible time. We additionally implement batched transfers for read-only tiles and optionally enable lazy transfers for the tiles of the output matrix. We evaluate our approach on an

NVIDIA HGX system, which features 8 NVIDIA A100 GPUs, interconnected with NVLink3 and NVSwitch2. Our experimental results show the effectiveness of our optimizations in the performance of GEMM. Our implementation outperforms state-of-the-art libraries (including our previous PARALiA implementation), by 1.29× and 1.37× on average, for FP32 and FP64 GEMM respectively. Moreover, our implementation offers high performance for irregular matrix shapes and varying initial data placements, significantly outperforming existing implementations.

We conclude that contrary to the popular belief that GEMM can easily reach the peak performance of multi-GPU compute nodes, in practice, it is communication-bound for many problem sizes, and requires a communication-aware implementation to overcome this limitation. In the future, we aim to extend this work to multiple multi-GPU compute nodes, combining our intra-node implementation with distributed techniques targeting scalability. We are working towards supporting other input data layouts, like PBLAS, which is commonly used in multi-node, large-scale systems.

We conclude that, despite the common conception that BLAS routines are well-suited for single- and multi-GPU systems, high performance for any problem on any system is very hard to achieve, and requires taking a plethora of parameters into account and fine-tuning execution for that exact problem/system configuration. Our solution for this is to use runtime autotuning, coupling modeling with micro-benchmarks to select values for internal routine parameters and optimize communication routing, overlap, scheduling and resource selection.

In the future, we aim to extend PARALiA with more sophisticated scheduling techniques, and we will work towards utilizing the acquired knowledge it already processes for further performance improvements in communication. We also plan to extend this work to multiple multi-GPU compute nodes, combining our intra-node implementation with distributed techniques targeting scalability. Finally, we are working towards supporting other input data layouts, like PBLAS, which is commonly used in multi-node, large-scale systems.

CHAPTER 7

Εκτεταμένη Περίληψη

7.1 Εισαγωγή

Οι πυκνές πράξεις γραμμικής άλγεβρας αποτελούν τα θεμελιώδη δομικά στοιχεία πολλών υπολογιστικών αλγορίθμων που εμφανίζονται σε πληθώρα εφαρμογών υψηλής απόδοσης (HPC), όπως η αριθμητική ανάλυση, η Υπολογιστική Δυναμική Ρευστών (CFDs), η μοντελοποίηση του κλίματος, η μοριακή δυναμική, η επεξεργασία εικόνας, η μηχανική μάθηση και η υπολογιστική όραση. Επιπλέον, ενώ αυτά τα χαμηλού επιπέδου δομικά στοιχεία εμφανίζονται στον κώδικα υψηλού επιπέδου των εφαρμογών που εκτελούν και άλλες πράξεις, συνήθως κυριαρχούν σε ένα σημαντικό μέρος του συνολικού χρόνου εκτέλεσης μιας επιστημονικής εφαρμογής. Κατά συνέπεια, η αύξηση της απόδοση ς και της απόδοσης των πυρήνων πυκνής γραμμικής άλγεβρας επηρεάζει άμεσα την συνολική αποτελεσματικότητα των εφαρμογών HPC.

Αυτό οδήγησε στην τυποποίηση των Basic Linear Algebra Subprograms (BLAS) [39, 40, 44–49, 90, 101] στα πρώτα στάδια της ανάπτυξης του HPC, για να διευκολυνθεί η ανάπτυξη επιστημονικού κώδικα, επιτρέποντας στους ειδικούς να βασίζονται σε τυποποιημένα και βελτιστοποιημένα δομικά στοιχεία για την υλοποίηση πιο σύνθετων προσομοιώσεων σε μεγάλη κλίμακα. Το πρότυπο BLAS καθορίζει ένα σύνολο ρουτινών "black box" που πρέπει να ακολουθούν μια συγκεκριμένη διάταξη εισόδου/εξόδου και να βελτιστοποιούνται από προμηθευτές και παρόχους βιβλιοθηκών με τρόπο διαφανή προς τον χρήστη, χωρίς να απαιτείται πρόσθετη βελτιστοποίηση απόδοσης.

Ωστόσο, ενώ οι βιβλιοθήκες BLAS που χρησιμοποιούν την προσέγγιση του "black box" διευκολύνουν την ανάπτυξη εφαρμογών για τους ειδικούς του τομέα, εσωτερικά απαιτείται σημαντική προσπάθεια για την υλοποίηση και βελτιστοποίηση των ρουτινών BLAS. Επιπλέον, ακόμα και αν όλες οι ρουτίνες BLAS είναι εξαντλητικά βελτιστοποιημένες για ένα σύστημα, η εμφάνιση νέων συστημάτων παρουσιάζει ένα πρακτικό πρόβλημα: οι μηχανικοί απόδοσης πρέπει να επανεξετάσουν όλες τις υλοποιήσεις BLAS και να τις βελτιστοποιήσουν εκ νέου.

Μεταφερσιμότητα CPU BLAS μέσω αυτόματης βελτιστοποίησης: Μια κοινή λύση σε αυτό το πρόβλημα είναι η *αυτοματοποίηση* μέρους της διαδικασίας βελτιστοποίησης που είναι ειδική για το σύστημα, γνωστή ως *αυτόματη βελτιστοποίηση.* Η αυτόματη βελτιστοποίηση έχει καθιερωθεί από τα πρώτα στάδια της ύπαρξης των CPU BLAS ως μια τυπική διαδικασία για την αύξηση της απόδοσης και της μεταφερσιμότητας σε νέα συστήματα [15, 29, 60, 61, 96, 107, 140, 158, 160, 163, 164]. Ενώ η ακριβής διαδικασία για τις BLAS διαφέρει ανά ρουτίνα, τύπο αρχιτεκτονικής και υλοποίηση, η ιδέα της αυτόματης βελτιστοποίησης είναι να προσαρμόζει αυτόματα την εκτέλεση προς έναν στόχο βελτιστοποίησης (χρόνος εκτέλεσης, ενεργειακή απόδοση ή άλλες μετρικές απόδοσης). Συγκεκριμένα για τις BLAS, συνήθως περιλαμβάνει την εύρεση των *εξωτερικών παραμέτρων ρουτίνας* (διαστάσεις προβλήματος, σημαίες κ.λπ.) που επηρεάζουν την απόδοση και την προσαρμογή των *εσωτερικών παραμέτρων* (μέγεθος μπλοκ/κελίδων, παράγοντες unrolling βρόχων κ.λπ.) κατά την εκτέλεση, με βάση τις δεδομένες εξωτερικές παραμέτρους. Αυτό απαιτεί τη δημιουργία μιας *σχέσης* μεταξύ των τιμών των εσωτερικών και εξωτερικών παραμέτρων, η οποία μπορεί να επιτευχθεί μέσω *πειραμάτων επίδοσης ή μοντελοποίησης.* Τα πειράματα επίδοσης αποδίδουν τα πιο ακριβή αποτελέσματα αλλά συνήθως έχουν έναν απαγορευτικό χώρο αναζήτησης, ενώ η μοντελοποίηση είναι ταχύτερη και δεν απαιτεί εξαντλητική εμπειρική δοκιμή, αλλά είναι λιγότερο ακριβής. Επιπλέον, μια πρακτική *ενδιάμεση* λύση είναι η χρήση ενός συνδυασμού των δύο μέσω της σύνδεσης γενικών μοντέλων με μικρό-πειράματα.

Takeaway Οι BLAS είναι σημαντικές για τις εφαρμογές του HPC και πρέπει να έχουν υψηλή απόδοση και μεταφερσιμότητα. Αυτό επιτυγχάνεται συνήθως μέσω αυτόματης βελτιστοποίησης, καθοδηγούμενης από μοντελοποίηση και μικρό-πειράματα.

Εκτέλεση BLAS σε GPU: Η εισαγωγή των GPU στο HPC άλλαξε το τοπίο της βελτιστοποίησης των BLAS. Η εύκολα παραλληλοποιήσιμη μορφή των ρουτινών BLAS τις έκανε κατάλληλες για τις GPU, γεγονός που οδήγησε στην ανάπτυξη πολλών GPU-BLAS βιβλιοθηκών, η πιο κοινή από τις οποίες είναι η *cuBLAS*, μια βιβλιοθήκη τύπου CUDA για τις NVIDIA GPU [122]. Η *cuBLAS* προσφέρει βελτιστοποιημένες βασικές πράξεις BLAS, αλλά σε αντίθεση με το απλούστερο παράδειγμα της CPU, απαιτεί τα δεδομένα εισόδου να βρίσκονται στη μνήμη της GPU. Αυτό σημαίνει ότι ο χρήστης πρέπει επίσης να διαχειριστεί τη μεταφορά δεδομένων προς και από την GPU πριν και μετά την εκτέλεση (εφεξής αναφερόμενη ως *offload*), εισάγοντας έναν νέο

περιορισμό απόδοσης [7, 21, 63, 103, 115, 116, 130, 134, 138]. Επιπλέον, η εκτέλεση προβλημάτων σε δεδομένα που δεν χωράνε στη μνήμη της GPU αναγκάζει τον χρήστη να διασπάσει τα αρχικά δεδομένα του προβλήματος σε μικρότερα τμήματα και να τα μεταφέρει στην GPU με σωληνοειδή τρόπο. Αυτή η τεχνική εισάγει επιπλέον υπολογιστικά και επικοινωνιακά κόστη, αλλά επιτρέπει επίσης την επικάλυψη επικοινωνίας με υπολογισμό για να αυξηθεί η απόδοση του offload [21, 63, 65, 159]. Αυτά τα νέα χαρακτηριστικά του GPU BLAS offload εισάγουν μια σειρά από αποφάσεις βελτιστοποίησης, όπως η απόφαση αν μια ρουτίνα θα πρέπει να μεταφερθεί σε συγκεκριμένη GPU, η απόφαση του ποσοστού του φόρτου εργασίας που θα πρέπει να εκτελεστεί στην CPU σε αντίθεση με τη GPU αν εκτελείται με υβριδικό τρόπο [5, 16, 41, 79, 80, 108, 110, 145, 148–150], και η επιλογή του κατάλληλου μεγέθους τμήματος για τον διαχωρισμό του προβλήματος [21, 63, 65, 106, 159]. Παρόλο που η προηγούμενη έρευνα έχει αντιμετωπίσει αυτά τα θέματα για συγκεκριμένες διαμορφώσεις συστημάτων, η ραγδαία πρόοδος της τεχνολογίας GPU την τελευταία δεκαετία έχει καταστήσει αυτές τις προσεγγίσεις ξεπερασμένες και μη μεταφέρσιμες. Παρόλο που αυτό το πρόβλημα θα μπορούσε να λυθεί με αυτόματη βελτιστοποίηση, αποτελεί μια μεγάλη πρόκληση λόγω του χάσματος μεταξύ της μοντελοποίησης της απόδοσης των πυρήνων GPU [13, 35, 78, 81, 89, 109, 111, 113, 114, 127, 131, 136, 138, 165], της μοντελοποίησης της μεταφοράς δεδομένων [6, 21, 25, 34, 52, 63, 72, 73, 75, 76, 83, 88, 113, 115, 117, 134], της μοντελοποίησης επικάλυψης [21, 63, 65, 106, 159] και των πραγματικών υλοποιήσεων GPU BLAS που υποστηρίζουν επικάλυψη [4, 54, 55, 57, 68, 92, 124, 125, 157].

> **Takeaway** Η αυτόματη βελτιστοποίηση μπορεί να ωφελήσει σημαντικά το GPU BLAS, αλλά είναι πιο δύσκολο να εφαρμοστεί λόγω της πρόσθετης πολυπλοκότητας του offload και του χάσματος μεταξύ των υλοποιήσεων GPU και της μοντελοποίησης.

Εκτέλεση BLAS σε συστήματα με πολλαπλές GPU: Η επιτυχία των GPU σε εργασίες HPC οδήγησε στην ευρεία υιοθέτηση κόμβων με πολλές GPU, οι οποίοι συνήθως αποτελούνται από 4-8 GPU που είναι διασυνδεδεμένες με κάποια ιδιαίτερη τοπολογία. Λόγω των εξαιρετικών υπολογιστικών δυνατοτήτων αυτών των συστοιχιών, είναι ιδιαίτερα κατάλληλες για τις υπολογιστικά απαιτητικές πράξεις επιπέδου 3 των BLAS. Ωστόσο, η βελτιστοποίηση των πράξεων BLAS για συστήματα με πολλές GPU διαφέρει σημαντικά από αυτήν των συστημάτων με μία GPU, καθώς απαιτεί επίσης την αποτελεσματική κατανομή και διαχείριση των *δεδομένων* και των υπολογιστικών *εργασιών* μεταξύ πολλών *εργατών* (των GPU) με ξεχωριστές μνήμες. Επιπλέον, η ύπαρξη ξεχωριστών μνημών GPU περιπλέκει περαιτέρω την εκτέλεση, καθώς τα δεδομένα εισόδου μιας ρουτίνας μπορεί να βρίσκονται στη μνήμη του host, στη μνήμη της GPU ή σε συνδυασμό και των δύο. Κατά συνέπεια, η βελτιστοποίηση για πολλές GPU εισάγει νέες αλγοριθμικές έννοιες στο BLAS, παρόμοιες με την κατανεμημένη επεξεργασία, όπως η αποσύνθεση δεδομένων, ο προγραμματισμός εργασιών και η επικοινωνία. Όλα αυτά προσθέτουν στην εγγενή

πολυπλοκότητα του BLAS και έχουν ωθήσει πολλές βιβλιοθήκες να υποστηρίξουν την εκτέλεση σε πολλές GPU, είτε επεκτείνοντας προηγούμενες κατανεμημένες προσεγγίσεις με υποστήριξη GPU [4, 5, 11, 18, 19, 54, 55, 68, 92, 94, 161] είτε με εξειδικευμένες βιβλιοθήκες σχεδιασμένες ειδικά για αυτήν την περίπτωση [9, 57, 124, 125, 157].

Παρόμοια με την εκτέλεση σε μία GPU, αλλά ακόμα περισσότερο λόγω της πρόσθετης αλγοριθμικής πολυπλοκότητας της δρομολόγησης σε πολλές GPU, τα κυρίαρχα προβλήματα επίδοσης κάθε προβλήματος BLAS ποικίλλουν ανάλογα με τις εξωτερικές παραμέτρους του (διαστάσεις προβλήματος, θέση δεδομένων) και τα χαρακτηριστικά του υλικού (διασύνδεση, χωρητικότητα μνήμης host/GPU, δυνατότητες GPU). Κατά συνέπεια, η εκτέλεση σε πολλές GPU απαιτεί επίσης *αποφάσεις* με βάση το συγκεκριμένο πρόβλημα και το σύστημα, που μπορούν να βελτιώσουν την απόδοση και τη μεταφερσιμότητα. Τέτοιες αποφάσεις είναι: η δρομολόγηση της επικοινωνίας για καλύτερη αξιοποίηση του εύρους ζώνης του δικτύου διασύνδεσης, η διαχείριση της προσωρινής αποθήκευσης δεδομένων στις GPU, οι αποφάσεις διαχείρησης εργασίας για την αντιμετώπιση της ανισορροπίας, ο προγραμματισμός και η σειρά εκτέλεσης των εργασιών για την ελαχιστοποίηση του αποκλεισμού λόγω εξαρτήσεων εισόδου/εξόδου και οι αποφάσεις που σχετίζονται με την ορθή κατανομή πόρων για την αποφυγή περιττής χρήσης της GPU. Επιπλέον, λόγω της μεγαλύτερης πολυπλοκότητας των προβλημάτων σε πολλές GPU, η επίδραση αυτών των αποφάσεων στην απόδοση είναι σημαντικά μεγαλύτερη από ό,τι σε μία μόνο GPU. Δυστυχώς, η πολυπλοκότητα της λήψης όλων αυτών των αποφάσεων κατά την εκτέλεση είναι απαγορευτική. Αυτό αναγκάζει τις βιβλιοθήκες είτε να επικεντρώνονται σε ένα υποσύνολο αυτών, εφαρμόζοντας ευρετικές μεθόδους και ρύθμιση για κάθε νέο σύστημα εμπειρικά, είτε να χρησιμοποιούν γενικές λύσεις, όπως η βελτιστοποίηση διαγραμμάτων εργασιών ή η κλοπή εργασιών [4, 54, 56–58, 68, 92, 157]. Η πρώτη λύση έχει υψηλή απόδοση για ένα υποσύνολο των συνολικών προβλημάτων και για ορισμένα συστήματα, αλλά με πολύ χαμηλή μεταφερσιμότητα, οδηγώντας σε σημαντική πτώση της απόδοσης σε άλλα σενάρια [9]. Η δεύτερη, από την άλλη, είναι πιο γενική και φορητή σε νέα σενάρια, αλλά υστερεί σε απόδοση σε σχέση με μια εξειδικευμένη λύση βελτιστοποιημένη για συγκεκριμένα χαρακτηριστικά υλικού και προβλήματος [8]. Παρόλο που αυτό το πρόβλημα θα μπορούσε επίσης να λυθεί με αυτόματη βελτιστοποίηση, η τρέχουσα κατάσταση της τεχνολογίας δεν προσφέρει τέτοιες λύσεις λόγω της έλλειψης μοντέλων για συστήματα με πολλαπλές GPU, σε συνδυασμό με την πολύ υψηλή πολυπλοκότητα του προβλήματος.

> **Takeaway** Η αυτόματη βελτιστοποίηση μπορεί να λύσει τα κρίσιμα προβλήματα απόδοσης και μεταφερσιμότητας του multi-GPU BLAS, αλλά η υλοποίηση της έχει αποδειχθεί εξαιρετικά περίπλοκη με τις τρέχουσες μεθόδους.

7.1.1 Διατύπωση προβλήματος

Συνοψίζοντας, οι BLAS είναι πολύ σημαντικές για εφαρμογές HPC και θα πρέπει να είναι εύκολα μεταφέρσιμες και βέλτιστες στα σύγχρονα HPC συστήμα- τα με GPUs. Δυστυχώς, ενώ *θεωρητικά* οι ρουτίνες BLAS είναι κατάλληλες για συστήματα με GPU, *στην πράξη* τα πρόσθετα κόστη επικοινωνίας, προγραμματισμού και ανισορροπίας που εισάγονται σε αυτά τα συστήματα περιορίζουν σημαντικά την απόδοση του BLAS και οδηγούν σε υποχρησιμοποίηση των πόρων. Επιπλέον, οι υπάρχουσες λύσεις BLAS δεν μπορούν να προσαρμοστούν δυναμικά στην υποκείμενη αρχιτεκτονική υλικού και στον φόρτο εργασίας, βασιζόμενες έντονα στη χειροκίνητη ρύθμιση από το χρήστη, με αποτέλεσμα να έχουν χαμηλή απόδοση και μεταφερσιμότητα. Τέλος, ενώ η μοντελοποίηση και η αυτόματη βελτιστοποίηση μπορούν να βελτιώσουν σημαντικά την απόδοση, την ανθεκτικότητα και τη μεταφερσιμότητα, η *υλική* πολυπλοκότητα των σύγχρονων συστοιχιών GPU σε συνδυασμό με την *αλγοριθμική* πολυπλοκότητα της εκτέλεσης του GPU BLAS καθιστούν τη βελτιστοποίηση απόδοσης και την αυτόματη βελτιστοποίηση εξαιρετικά δύσκολη.

7.1.2 Συνεισφορές

Ο στόχος αυτής της διατριβής είναι να ξεπεράσει αυτούς τους περιορισμούς και να επιτύχει μεταφερσιμότητα, βέλτιστη απόδοση και αποτελεσματική αξιοποίηση πόρων για single- και multi-GPU BLAS. Για αυτόν τον σκοπό, επιλέγουμε μια προσέγγιση που βασίζεται σε μοντέλα, όπου τα χαρακτηριστικά του προβλήματος και του συστήματος παραμετροποιούνται και χρησιμοποιούνται για τη διαμόρφωση μοντέλων πρόβλεψης. Στη συνέχεια, εφαρμόζουμε αυτά τα μοντέλα για την αυτόματη βελτιστοποίηση των BLAS, αναπτύσσοντας μια end-to-end βιβλιοθήκη που προσαρμόζει τις παραμέτρους που επηρεάζουν την απόδοση σε κάθε ατομική κλήση BLAS, λαμβάνοντας υπόψη τα χαρακτηριστικά της ρουτίνας, του προβλήματος και του συστήματος κατά τον χρόνο εκτέλεσης και προσαρμόζοντας την εκτέλεση ανάλογα με το εκάστοτε σενάριο. Συνεπώς, οι κύριες συνεισφορές της δουλειάς μας είναι:

1. Μια μεθοδολογία για την μοντελοποίηση του δικτύου διασύνδεσης και της επικοινωνίας ρουτινών BLAS σε συστοιχίες multi-GPU και η δημιουργία μοντέλων για την εκτίμηση της απόδοσης τους.
2. Το PARALiA, μια βιβλιοθήκη υψηλής απόδοσης και ενεργειακής επίδοσης γιά multi-GPU BLAS, βασισμένη στη μοντελοποίηση απόδοσης και την αυτόματη βελτιστοποίηση.
3. Το PARALiA-GEMMx, μια υλοποίηση πολλαπλασιασμού πινάκων για multi-GPU συστήματα που βασίζεται στην καθοδήγηση από μοντέλα για την καλύτερη δρομολόγηση και την βελτιστοποίηση της επικοινωνίας.

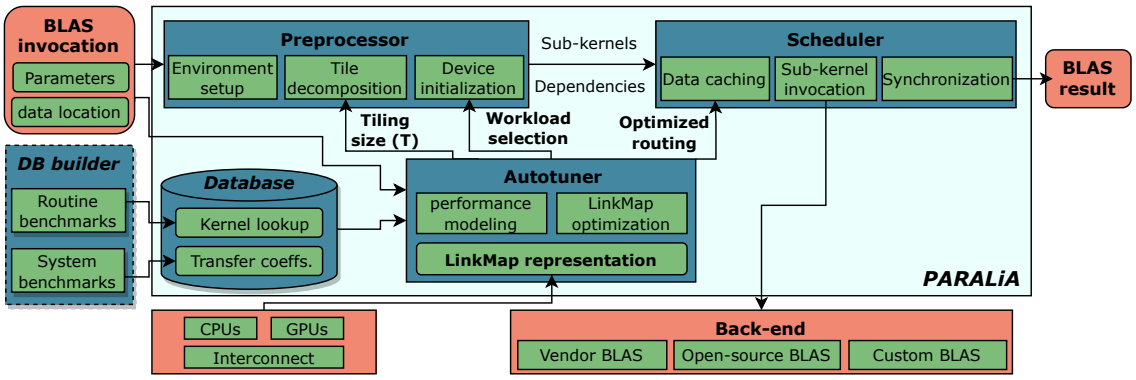


Figure 7.1: Μια επισκόπηση του PARALiA και των κύριων κομματιών του.

7.2 PARALiA: Αυτόματοποίηση BLAS σε πολλαπλές GPU

Σε αυτή την ενότητα παρουσιάζουμε το *PARALiA*, μια βιβλιοθήκη BLAS για συστήματα πολλαπλών GPU, που αντιμετωπίζει τις ελλείψεις των προηγούμενων βιβλιοθηκών συνδυάζοντας τη μοντελοποίηση με την αυτόματη βελτιστοποίηση. Ξεκινάμε με μια γενική επισκόπηση της βιβλιοθήκης και των βασικών κομματίων της και, στη συνέχεια, περιγράφουμε κάθε κομμάτι και τον ρόλο του στην βελτιστοποίηση με περισσότερες λεπτομέρειες. Η Εικόνα 7.1 δείχνει τον σχεδιασμό του PARALiA. Το PARALiA ενεργοποιείται όταν ο κώδικας του χρήστη καλεί μια ρουτίνα BLAS με τα δεδομένα της ρουτίνας να βρίσκονται στη μνήμη οποιασδήποτε από τις διαθέσιμες συσκευές. Το PARALiA αποτελείται από τρία κύρια συστατικά: έναν προεπεξεργαστή που είναι υπεύθυνος για την προετοιμασία του περιβάλλοντος του συστήματος για την εκτέλεση, έναν προγραμματιστή/δρομολογητή που είναι υπεύθυνος για τη διαχείριση των εισερχόμενων/ εξερχόμενων δεδομένων και την κλήση των πυρήνων BLAS, και έναν αυτόματο βελτιστοποιητή (sec 7.2.1) που λαμβάνει παραμέτρους συστήματος και προβλήματος από μια βάση δεδομένων, μια απεικόνιση υλικού (LinkMap, sec 7.2.2) και την κλήση της ρουτίνας BLAS, και αποφασίζει ποιες συσκευές θα χρησιμοποιηθούν για την εκτέλεση, το μέγεθος των βασικών υπολογιστικών μπλοκ (tiling size) και τη διαδρομή μεταφοράς δεδομένων. Το πλαίσιο PARALiA είναι ένα δημόσια διαθέσιμο έργο ανοιχτού κώδικα.

7.2.1 Ο αλγόριθμος του αυτόματου βελτιστοποιητή

Ο αυτόματος βελτιστοποιητής αποτελεί τον πυρήνα της βελτιστοποίησης του PARALiA. Σκοπός του είναι η βελτίωση της 1) επικοινωνίας και 2) της κατανομής φόρτου εργασίας για οποιοδήποτε συνδιασμό συστήματος/προβλήματος. Λόγω της πιο γενικής φύσης αυτού του προβλήματος, η χρήση μιας προσέγγισης βασισμένης σε ευρετικές μεθόδους θα ευνοούσε ένα υποσύνολο διαμορφώσεων, με βάση τις οποίες σχεδιάστηκαν οι ευρετικές μέθοδοι, είτε δηλαδή χαρακτηρι-

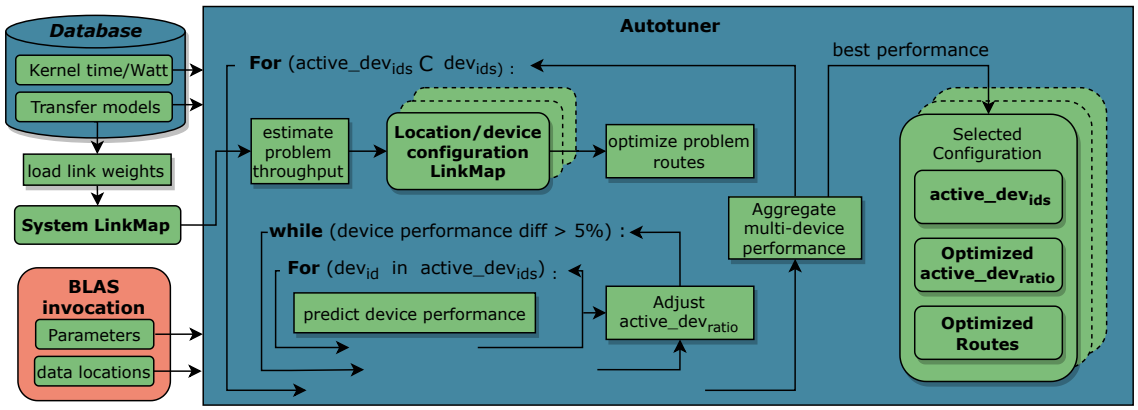


Figure 7.2: Μια επισκόπηση του αυτόματου βελτιστοποιητή του PARALiA και της αλυσίδας πρόβλεψης του.

στικά συγκεκριμένων συστημάτων (π.χ. αριθμός CPU/GPU, διασύνδεση) είτε χαρακτηριστικά προβλήματος (π.χ. μέγεθος δεδομένων, τοποθέτηση). Για το λόγο αυτό, ο αυτόματος βελτιστοποιητής χρησιμοποιεί μια προσέγγιση βασισμένη σε μοντέλο, που αντιμετωπίζει κάθε διαμόρφωση ως διαφορετικό πρόβλημα, συνδυάζοντας τα *χαρακτηριστικά του συστήματος* και *του προβλήματος* κατά την εκτέλεση.

Ο αλγόριθμος αυτόματης βελτιστοποίησης που εκτελείται κατά τη διάρκεια κάθε κλήσης ρουτίνας φαίνεται αναλυτικά στην Εικόνα 7.2. Όταν καλείται μια ρουτίνα, οι παράμετροι του προβλήματος εξάγονται από τη ρουτίνα. Ο αυτόματος βελτιστοποιητής φορτώνει προαποκτηθέντα μοντέλα μεταφοράς από τη βάση δεδομένων του PARALiA και τους χρησιμοποιεί για να κατασκευάσει μια αναπαράσταση των χαρακτηριστικών του συστήματος που ονομάζεται LinkMap. Στη συνέχεια, ο αυτόματος βελτιστοποιητής δοκιμάζει επαναληπτικά τις **υποψήφιες κατανομές φόρτου εργασίας**, εκτιμά τη συνολική τους απόδοση και επιλέγει την καλύτερη. Κάθε *κατανομή φόρτου εργασίας* αποτελείται από α) μια λίστα από συσκευές dev_{num} ($active\_dev_{ids}$), υποσύνολο των συνολικών συσκευών του συστήματος, β) μια λίστα ποσοστών ($active\_dev_{ratio}$) του συνολικού προβλήματος που προτείνονται για κάθε συσκευή και γ) έναν χάρτη διαδρομής μεταφοράς δεδομένων βελτιστοποιημένο για αυτή τη συγκεκριμένη κατανομή. Όσον αφορά το (α) και το (β), δεδομένου ότι ο συνδυασμένος χώρος αναζήτησής τους είναι πολύ μεγάλος (τα $active\_dev_{ratio}$ είναι τιμές float), τα διαχωρίζουμε επαναλαμβάνοντας τους πιθανούς συνδυασμούς συσκευών $active\_dev_{ids}$ (που είναι διακριτές) και επιλέγοντας τα $active\_dev_{ratio}$ με μια μέθοδο βασισμένη σε μοντέλο. Συγκεκριμένα, για κάθε συνδυασμό συσκευών ξεκινάμε με ίσα ποσοστά και τα προσαρμόζουμε σταδιακά με βάση μια πρόβλεψη απόδοσης για κάθε συσκευή, μέχρι να επιτευχθεί ένα $active\_dev_{ratio}$ με παρόμοια απόδοση ανά συσκευή (εντός 5%). Όσον αφορά το (γ), ο αυτόματος βελτιστοποιητής προσαρμόζει και βελτιστοποιεί το *LinkMap* σε κάθε παραπάνω σενάριο χρησιμοποιώντας τα συγκεκριμένα χαρακτηριστικά του προβλήματος

(περισσότερα στην ενότητα 7.2.2). Τέλος, η καλύτερη *κατανομή φόρτου εργασίας* επιλέγεται χρησιμοποιώντας κάποιον συναθροιστή σχετικό με μετρικές (π.χ. μέγιστο για χρόνο, άθροισμα για ενέργεια κ.λπ.) για την απόδοση κάθε συσκευής που αποκτήθηκε κατά την εκτίμηση του (β). Σημειώνουμε ότι ο αυτόματος βελτιστοποιητής επιλέγει επίσης ένα μέγεθος tile **T** για την αποσύνθεση (όπως φαίνεται στην Εικόνα 7.1), αλλά αυτή η διαδικασία είναι αποσυνδεδεμένη από το (α), (β) και (γ) και πραγματοποιείται βάσει του CoCoPeLiA [7] λόγω της μικρής της επίδρασης στην απόδοση πολλαπλών GPU.

7.2.2 Μοντελοποίηση των δικτύων διασύνδεσης: Η αναπαράσταση `LinkMap`

Δεδομένου ότι τα μοντέλα δικτύων διασύνδεσης των προηγούμενων βιβλιοθηκών στοχεύουν σε συστήματα με παρόμοιες δυνατότητες συσκευών και διασυνδέσεων, δεν είναι κατάλληλες για οποιαδήποτε κατανομή φορτίου εργασίας. Για να το μετριάσουμε αυτό, υποθέτουμε το πιο γενικό σύστημα σε μια αναπαράσταση που ονομάζεται `LinkMap`, ικανή να αναπαραστήσει οποιοδήποτε σύστημα με αυθαίρετες συσκευές και συνδέσεις μεταξύ τους.

Αναπαράσταση μορφής υλικού: Για να μοντελοποιήσουμε οποιοδήποτε πιθανό σύστημα, η αναπαράσταση `LinkMap` αποσυνδέεται από την έννοια της "CPU" και της "Κύριας μνήμης" και αντιμετωπίζει όλα τα μέρη ενός συστήματος με τον ίδιο τρόπο: οποιαδήποτε πιθανή τοποθεσία δεδομένων ή διαθέσιμος υπολογιστικός πόρος κατηγοριοποιείται ως ***συσκευή*** και συνδέεται μέσω ***συνδέσμων*** με όλες τις άλλες συσκευές, οι οποίες είναι υπεύθυνες για τις μεταφορές δεδομένων μεταξύ τους. Στην αναπαράσταση `LinkMap` κάθε *συσκευή* ορίζεται από ένα μοναδικό αναγνωριστικό (dev_{id}). Αν και δεν είναι κοινό σε τρέχοντα συστήματα, αν διαφορετικές συσκευές έχουν κοινή μνήμη ο χρόνος μεταφοράς δεδομένων μεταξύ τους είναι πάντα ίσος με μηδέν. Επιπλέον, αυτή η αναπαράσταση υποθέτει μια πλήρως συνδεδεμένη εικονική τοπολογία, ακόμη κι αν δεν υπάρχει πραγματική σύνδεση υλικού μεταξύ ενός ζεύγους συσκευών. Ως εκ τούτου, δημιουργεί ένας πλήρως συνδεδεμένος γράφος, όπου οι συσκευές είναι οι κόμβοι και οι σύνδεσμοι είναι οι ακμές: οι dev_{num} κόμβοι συνδέονται μέσω ενός δικτύου 2D πλέγματος (dev_{num}, dev_{num}) ακμών/συνδέσμων. Η αναπαράσταση `LinkMap` υλοποιείται σε C++ ως κλάση της οποίας τα μέλη και οι συναρτήσεις φαίνονται στον Πίνακα 7.1. Αποτελείται από πέντε 2D πίνακες $link_{\{lat,bw,bw-shared,route,sl\}}$ που κρατούν τις τιμές του και τρεις συναρτήσεις που χρησιμοποιούνται κατά την αυτόματη βελτιστοποίηση για την ενημέρωσή τους.

Βελτιστοποίηση δρομολόγησης κατά την εκτέλεση: Η αναπαράσταση `LinkMap` από μόνη της δεν περιέχει καμία πληροφορία, απλώς αναπαριστά την πιο γενική περίπτωση. Η χρησιμότητά της έγκειται στην προσαρμοστικότητά της σε οποιοδήποτε σύστημα και διάταξη δεδομένων προβλήματος, που συμβαίνει κατά την εκτέλεση. Αυτή η διαδικασία έχει τρεις βασικές φάσεις, οι οποίες υλοποιούνται ως συναρτήσεις του LinkMap στον Πίνακα 7.1. Πρώτον, μία φορά ανά πρόγραμμα κατά την πρώτη κλήση ρουτίνας, το $load\_link\_weights()$ φορτώνει τους συντελε-

Table 7.1: Μέλη και συναρτήσεις του `LinkMap` που χρησιμοποιούνται για τη βελτιστοποίηση επικοινωνίας.

Συστήματα:	
$link_{lat}(dest_{id}, src_{id})$	Η καθυστέρηση κάθε συνδέσμου.
$link_{bw}(dest_{id}, src_{id})$	Το απομονωμένο εύρος ζώνης κάθε συνδέσμου.
$link_{sl}(dest_{id}, src_{id}, s\_dest_{id}, s\_src_{id}))$	Η επιβράδυνση που επιβάλλεται από ταυτόχρονη χρήση σε κάθε ζεύγος συνδέσμων.
Προσαρμογή προβλήματος:	
$link_{bw-shared}(dest_{id}, src_{id})$	Το βιώσιμο εύρος ζώνης κάθε συνδέσμου για μια διαμόρφωση συσκευής/δεδομένων.
$link_{route}(dest_{id}, src_{id})$	Η υποκείμενη διαδρομή που πρέπει να ακολουθήσουν όλες οι μεταφορές που περνούν μέσω ενός συνδέσμου.
Συναρτήσεις:	
$load\_link\_weights()$	Αρχικοποιεί το $link_{bw/lat/sl}$ από τη βάση δεδομένων.
$estimate\_problem\_throughput()$	Εκτιμά το $link_{\{bw-shared\}}$ για μια διαμόρφωση συσκευής/δεδομένων.
$optimize\_problem\_routes()$	Επαναδρομολογεί την επικοινωνία για τους 'κακούς' συνδέσμους για μια διαμόρφωση συσκευής/δεδομένων.

στές μεταφοράς $link_{\{lat,bw,sl\}}$ από τη βάση δεδομένων. Αυτό παρέχει ένα βασικό *LinkMap Συστήματος* που περιέχει εμπειρικές εκτιμήσεις για το σύστημα γενικά. Στη συνέχεια, κατά τη διάρκεια της αυτόματης βελτιστοποίησης, το $estimate\_problem\_throughput()$ προσαρμόζει τα εύρη ζώνης του LinkMap ($link_{bw-shared}$) σύμφωνα με την τρέχουσα διαμόρφωση συσκευής/δεδομένων. Συγκεκριμένα, υποθέτει ότι όλοι οι σύνδεσμοι που συνδέουν τις dev_{num} συσκευές ($active\_dev_{ids}$) με τις $data_{num}$ τοποθεσίες δεδομένων ($data_{locs}$) εκτελούν μεταφορές για ολόκληρη την εκτέλεση της ρουτίνας, και εφαρμόζει μια επιβράδυνση για την ταυτόχρονη χρήση στο εύρος ζώνης κάθε τέτοιου συνδέσμου:

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

Η τελική φάση βελτιστοποίησης είναι η εφαρμογή ενός απλοποιημένου αλγόριθμου *συντομότερης διαδρομής* σε αυτό το γράφημα, παρόμοιου με τον αλγόριθμο Floyd–Warshall αλλά με έναν μέγιστο αριθμό *hops* (ενδιάμεσες τοποθεσίες). Πιο συγκεκριμένα, θέλουμε να επαναδρομολογήσουμε μεταφορές που θα περνούσαν από συνδέσμους με χαμηλό εύρος ζώνης σε *σειρές συνδέσμων*

με υψηλότερο εύρος ζώνης. Για παράδειγμα, χρησιμοποιώντας μέγιστο αριθμό 3 συσκευών, αν $link_{bw-shared}(0 \rightarrow 2) = 2Gb/s$, $link_{bw-shared}(0 \rightarrow 1) = 3Gb/s$ και $link_{bw-shared}(1 \rightarrow 2) = 4Gb/s$ το $link_{route}(0 \rightarrow 2)$ θα αλλάξει από $\{0 \rightarrow 2\}$ σε $\{0 \rightarrow 1 \rightarrow 2\}$. Αυτή η βελτιστοποίηση αυξάνει το διαθέσιμο εύρος ζώνης δρομολογώντας μεταφορές μέσω καλύτερων συνδέσμων, και τοποθετεί και τα δεδομένα στις ενδιάμεσες συσκευές που βρίσκονται κατά μήκος της διαδρομής μεταφοράς, βελτιώνοντας σημαντικά τη συνολική επίδοση της βιβλιοθήκης.

Εκτίμηση απόδοσης για την επιλογή κατανομής φορτίου εργασίας: Για να επιλέξει ο auto-tuner την κατανομή φορτίου εργασίας που θα χρησιμοποιήσει για την εκτέλεση μίας ρουτίνας, πρέπει να εκτιμήσει την απόδοση κάθε συσκευής γιά ένα πρόβλημα. Η μοντελοποίηση ξεκινά από μία βασική προσέγγιση πλήρους επικάλυψης (full-overlap) που είναι συντηρητική και αναπαριστά μια ανώτερη επίδοση για την/τα τρέχουσα συσκευή/δεδομένα. Συγκεκριμένα, το μοντέλο χρησιμοποιεί: 1) τις τιμές εκτέλεσης και κατανάλωσης ισχύος που ανακτήθηκαν εμπειρικά από τη βάση δεδομένων, 2) τον αριθμό δεδομένων που συμμετέχουν στη ρουτίνα και την/τα τρέχουσα συσκευή/δεδομένα κατά την εκτέλεση, και 3) προβλέψεις του μοντέλου για την τιμή της μέτρησης απόδοσης (π.χ. χρόνο εκτέλεσης, κατανάλωση ισχύος) κάθε συσκευής, όπως και το συνολικό χρόνο εκτέλεσης (αθροίζοντας τη συνολική απόδοση). Η κύρια μεθοδολογία που χρησιμοποιείται για τη μοντελοποίηση παρατίθεται στον Πίνακα 7.2.

Πρώτα, συνδυάζουμε το ανώτατο όριο *full-overlap* [159] με τη βάση δεδομένων του PARALiA για να λάβουμε μια ειδική πρόβλεψη πλήρους επικάλυψης για τη συνολική απόδοση της ρουτίνας σε *κάθε* συσκευή:

$$pred\_t_{base}(dev_{id}) = max(t_{exec}(dev_{id}, dims), \\ t_{h2d}(dev_{id}, \sum_{i}^{is_R} bytes(i)), t_{d2h}(dev_{id}, \sum_{j}^{is_W} bytes(j))) \tag{7.1}$$

όπου το *h2d* αναφέρεται σε μεταφορές από τον υπολογιστή στη συσκευή και το *d2h* σε μεταφορές από τη συσκευή στον υπολογιστή, και $\sum_{\{i,j\}}^{is_{\{R,W\}}}$ είναι τα υποσύνολα των πινάκων/διανυσμάτων $data_{num}$ που αποτελούν τις εισόδους και τις εξόδους του προβλήματος, αντίστοιχα. Για να προσαρμόσουμε το μοντέλο σε κατανομή φόρτου εργασίας πολλαπλών συσκευών, πρέπει να αντικαταστήσουμε το *h2d* και το *d2h time* με τους χρόνους μεταφοράς όλων των *συνδέσεων* που συνδέουν τα $data_{locs}$ με κάθε συσκευή. Για να το επιτύχουμε, πρώτα, υπολογίζουμε το χρόνο μεταφοράς για κάθε σύνδεση (t_{link}) ως συνάρτηση των μεταφερόμενων $bytes$ με:

$$t_{link}(dest_{id}, src_{id}, bytes) = link_{lat}(dev_{id}, src_{id}) + \frac{bytes}{link_{bw-shared}(dest_{id}, src_{id})} \tag{7.2}$$

συνδυάζοντας την καθυστέρηση και το εύρος ζώνης κάθε σύνδεσης, χρησιμοποιώντας το κλασικό μοντέλο καθυστέρησης/εύρους ζώνης [7,16,21,65,106,159]. Στη συνέχεια, υποθέτουμε το καλύτερο

Table 7.2: Ορολογία μοντελοποίησης που χρησιμοποιείται σε αυτή την εργασία.

Εμπειρικές τιμές (από τη βάση δεδομένων):	
$t_{exec}(routine, dev_{id}, D1[, D2[, D3]])$	Ο χρόνος εκτέλεσης της $routine$ στη συσκευή dev_{id} ως συνάρτηση του μεγέθους του προβλήματος.
$W_{exec}(routine, dev_{id}, D1[, D2[, D3]])$	Η μέση ισχύς (σε Watt) της $routine$ στη συσκευή dev_{id} κατά την εκτέλεση.
Παράμετροι προβλήματος (από τη ρουτίνα):	
$dims : D1[, D2[, D3]]$	Διαστάσεις προβλήματος για τις BLAS επιπέδου-1, 2 και 3, αντίστοιχα.
$data_{num}$	Ο αριθμός των συνολικών πινάκων και διανυσμάτων που χρησιμοποιούνται από αυτή τη ρουτίνα.
$is_{\{R,W\}}(data_{num})$	Ένα flag [0,1] που υποδεικνύει αν ένας πίνακας/διάνυσμα είναι είσοδος/έξοδος, αντίστοιχα.
$data_{loc}(data_{num})$	Η τοποθέτηση δεδομένων για κάθε συμμετέχοντα πίνακα/διάνυσμα.
$bytes(data_{num})$	Το μέγεθος σε bytes όλων των πινάκων και διανυσμάτων που χρησιμοποιούνται από αυτή τη ρουτίνα.
Εκτιμώμενα (βάσει μοντέλου):	
dev_{num}	Ο αριθμός συσκευών που συμμετέχουν σε παράλληλη εκτέλεση πολλαπλών συσκευών.
$active\_dev_{ids}(dev_{num})$	Μια λίστα που περιέχει τα ids για κάθε τέτοια συσκευή.
$active\_dev_{ratio}(dev_{num})$	Το ποσοστό του συνολικού προβλήματος που δίνεται σε κάθε τέτοια συσκευή.
$pred_{metric}(dev_{id})$	Μια πρόβλεψη του $metric$ που απαιτείται για να ολοκληρώσει η συσκευή dev_{id} τους εκχωρημένους της υποπυρήνες.
$total\_pred\_metric$	Η συνολική εκτιμώμενη τιμή του $metric$ (π.χ. χρόνος, EDP) για παράλληλη εκτέλεση πολλαπλών συσκευών.

σενάριο, όπου όλοι οι πίνακες/διανύσματα εισόδου κατανέμονται ισότιμα μεταξύ των συσκευών dev_{num} συνδυάζοντας την εξίσωση 3.1 με την εξίσωση 3.2 για να γενικεύσουμε για οποιαδήποτε

αρχική τοποθέτηση δεδομένων:

$$pred\_t_{over}(...) = max(t_{exec}(...), \sum_{i}^{is_R} t_{link}(dev_{id}, data_{locs}(i), \frac{bytes(i)}{dev_{num}}), \sum_{j=0}^{is_W} t_{link}(data_{locs}(j), dev_{id}, \frac{bytes(j)}{dev_{num}})) \quad (7.3)$$

Η εξίσωση 3.3 παρέχει μια ακριβέστερη πρόβλεψη για την απόδοση πλήρους επικάλυψης μιας ρουτίνας, *εφόσον η εκτέλεση σε πολλαπλές GPUs δεν περιλαμβάνει πρόσθετες μεταφορές/κοινή χρήση δεδομένων μεταξύ συσκευών.* Αυτή η υπόθεση λειτουργεί για τις BLAS επιπέδου 1 και 2, αλλά στην αποδόμηση BLAS επιπέδου 3, κάθε tile (tile) επαναχρησιμοποιείται από πολλούς υποπυρήνες και επομένως μεταφέρεται σε πολλές συσκευές κατά τη διάρκεια της εκτέλεσης της ρουτίνας. Δεδομένου ότι το PARALiA χρησιμοποιεί μια 2D κυκλική αποδόμηση (DC_{row}, DC_{col}) για τις BLAS επιπέδου 3, θεωρούμε το βασικό σενάριο 1) ανταλλαγής ίσων μεριδίων από τα RONLY bytes μεταξύ όλων των σειρών και στηλών της αποδόμησης και 2) καμία κοινή χρήση δεδομένων εξόδου. Εκτιμούμε την αναλογική αύξηση στον όγκο μεταφορών για κάθε συσκευή ως εξής:

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

Όπου $RONLY_{num}$ είναι ο αριθμός των πινάκων/διανυσμάτων με $is_R = 1$ και $is_W = 0$, και το άθροισμα των αντίστοιχων $bytes$ είναι $RONLY\_sum\_bytes$. Αυτό αντιπροσωπεύει ένα κατώτατο όριο των πρόσθετων bytes λόγω κοινής χρήσης δεδομένων *για κάθε συσκευή.* Υποθέτουμε ότι αυτά τα bytes κατανέμονται ισομερώς μεταξύ των συσκευών, και χρησιμοποιούμε το *μέσο εύρος ζώνης* όλων των συνδέσεων για να εκτιμήσουμε τον πρόσθετο χρόνο μεταφοράς:

$$t_{extra}(dev_{id}) = extra\_transfer\_bytes \cdot \frac{dev_{num}}{\sum_{idx=0}^{dev_{num}} link_{bw-shared}[dev_{id}][idx]} \quad (7.4)$$

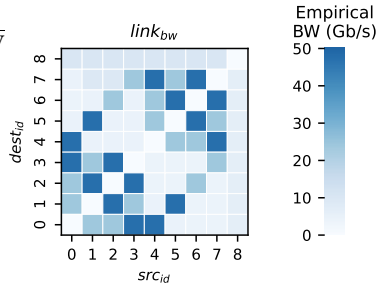
όπου η πρόσθετη επικοινωνία σε bytes για κάθε συσκευή πολλαπλασιάζεται με το αντίστροφο του μέσου εύρους ζώνης λήψης, το οποίο λειτουργεί ως μέση εκτίμηση για το αναμενόμενο εύρος ζώνης αυτών των μεταφορών. Τελικά, κατασκευάζουμε το μοντέλο πλήρους επικάλυψης που χρησιμοποιείται για την *εκτιμώμενη απόδοση κάθε GPU σε περιβάλλον πολλαπλών GPUs* προσθέτοντας τον πρόσθετο χρόνο μεταφοράς της εξίσωσης 3.4 στην εξίσωση 3.3:

$$pred_t(dev_{id}, ...) = max(t_{exec}(...), t_{extra}(dev_{id}) + \sum_{i}^{is_R} t_{link}(...), \sum_{j}^{is_W} t_{link}(...)) \quad (7.5)$$

Table 7.3: Το συστήμα **CLX-AI**.

Vulcan CLX-AI	CPU	GPU
Υπολογιστική Ικανότητα:	4 X Intel Xeon Gold 6240 CPU	8 X NVIDIA Tesla V FP peak 14 TFlop/s
	18 πυρήνες @ 2.60GHz	DP peak 7 TFlop/s
Μνήμη:	768GB DDR4	32 GB HBM2 760 GB/s
Διασύνδεση:	PCIe Gen3 x16	NVlink 1.0/2.0
ΛΣ:	Rocky Linux release 8.7	Οδηγός CUDA -
Πυρήνας:	4.18.0-425.3.1.el8.x86_64	510.108.03
Μεταγλωττιστής:	g++ 11.2.0	CUDA 11.6
Σημαίες βελτιστοποίησης:	-O3	-O3, -arch=sm_75

Figure 7.3: Χάρτης συνδέσεων **CLX-AI**.



7.2.3 Πειραματική Αξιολόγηση

Για την αξιολόγηση της απόδοσης, χρησιμοποιούμε τους κόμβους "clx-ai" του HPC cluster `Vulcan` του HLRS [69]. Τα χαρακτηριστικά του συστήματος παρουσιάζονται στον Πίνακα 7.3, μαζί με το έυρος ζωνης διασύνδεσης αποθηκευμένο στον `LinkMap` για τις 9 συσκευές (8 GPUs + CPU). Επιλέγουμε ένα μεγάλο σύνολο δεδομένων και επικεντρωνόμαστε αποκλειστικά στον πολλαπλασιασμό πινάκων διπλής ακρίβειας (`dgemm`) για την αξιολόγηση της απόδοσης. Εξερευνούμε 21 τετραγωνικά μεγέθη προβλημάτων ($M_{sq} = N_{sq} = K_{sq} = (2 \xrightarrow{\text{step=1}} 22) \cdot 2^{10}$), 21 προβλήματα τύπου fat-by-thin ($M_{fat} = N_{fat} = (8 \xrightarrow{\text{step=4}} 32) \cdot 2^{10}, Kthin = \frac{M_{fat}}{r}, r \in [2, 8, 32]$) και 21 προβλήματα τύπου thin-by-fat ($K_{fat} = (12 \xrightarrow{\text{step=4}} 36) \cdot 2^{10}, M_{thin} = N_{thin} = \frac{K_{fat}}{r}, r \in [2, 8, 32]$) για 10 συνδυασμούς τοποθεσιών (περισσότερα στο Σχήμα 7.4) για συνολικά 630 προβλήματα. Για κάθε τέτοιο πρόβλημα, μετράμε τον χρόνο εκτέλεσης t των 1) *cuBLASXt*, 2) *BLASX*, 3) *XKBLAS* και 4) δύο εκδόσεις του PARALiA (*comm_opt*,*select*(EDP_i)). Το *PARALiA comm_opt* βελτιστοποιεί μόνο την επικοινωνία χωρίς να χρησιμοποιεί επιλογή συσκευής, ενώ η άλλη έκδοση επιλέγει επίσης ποιές συσκευές να αξιοποιήθουν ανάλογα το πρόβλημα.

7.2.3.1 Απόδοση

Το Σχήμα 7.4 δείχνει τα αποτελέσματα αξιολόγησης για όλο το σύνολο δεδομένων. Το *cuBLASXt* έχει χαμηλή απόδοση λόγω της στατικής κατανομής τύπου round-robin καθώς και της απουσίας λογικής προσωρινής αποθήκευσης και επαναχρησιμοποίησης δεδομένων. Αντίθετα, η *BLASX* παρέχει καλή απόδοση για το σενάριο (h,h,h), η οποία όμως μειώνεται σημαντικά σε όλους τους άλλους συνδυασμούς τοποθεσιών. Αυτό το μοτίβο παρατηρείται για όλα τα σχήματα δεδομένων και είναι πιο εμφανές σε προβλήματα fat-thin και thin-fat επειδή απαιτούν πιό πολύ επικοινωνία από το τετράγωνο σχήμα, για το οποίο το GEMM έχει τη μεγαλύτερη αριθμητική ένταση (computational intensity). Το *XKBLAS* ακολουθεί παρόμοιο μοτίβο, με ένα μόνο χαρακτηριστικό που το διακρίνει: έχει την υψηλότερη απόδοση (h,h,h) από όλες τις βιβλιοθήκες, αλλά η μείωση της απόδοσης σε όλους τους άλλους συνδυασμούς τοποθεσιών είναι πολύ μεγαλύτερη

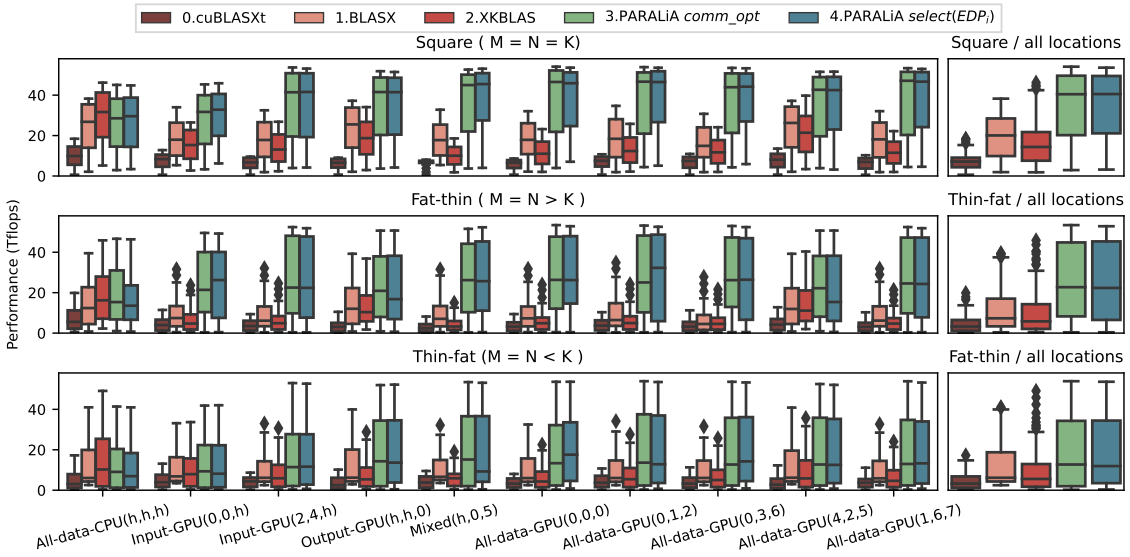


Figure 7.4: Απόδοση του dgemm για τις cuBLASXT, BLASX, XKBLAS και δύο παραλλαγές του PARALiA (μία που χρησιμοποιεί πάντα όλες τις GPUs και μία που επιλέγει την κατανομή φόρτου εργασίας για να μεγιστοποιήσει το αντίστροφο του προϊόντος καθυστέρησης-ενέργειας EDP_i). Κάθε σειρά αντιστοιχεί σε διαφορετικό σχήμα δεδομένων M, N, K και κάθε ομάδα boxplot σε διαφορετική τοποθέτηση δεδομένων, με $gemm_{loc} = (A_{loc}, B_{loc}, C_{loc})$, όπου το loc = h αντιστοιχεί σε δεδομένα στην μνήμη της CPU και loc = dev_{id} στη μνήμη της αντίστοιχης συσκευής. Το δεξί υποσχήμα συνοψίζει τα αποτελέσματα για κάθε σχήμα προβλήματος.

από του *BLASX*, με αποτέλεσμα αρκετά κατώτερη μέση απόδοση. Αυτή η συμπεριφορά επισημαίνεται στο boxplot του δεξιού υποσχήματος του Σχήματος 7.4, στο οποίο η απόδοση για τα προβλήματα τύπου $gemm_{loc} = (h, h, h)$ ακολουθούν την προαναμενόμενη κατανομή.

7.2.3.2 Ενεργειακή απόδοση

Το Σχήμα 7.5 παρουσιάζει τα αποτελέσματα της ενεργειακής απόδοση ς για το σύνολο δεδομένων μας, χρησιμοποιώντας το αντίστροφο προϊόν ισχύος-καθυστέρησης (PDP_i σε Gflops/W). Και οι δύο υλοποιήσεις του PARALiA έχουν ανώτερο PDP_i από τις άλλες βιβλιοθήκες, κάτι που στην περίπτωση του PARALiA *comm_opt* σε σύγκριση με τα *cuBLASXt*, *BLASX*, *XKBLAS* οφείλεται στη διαφορά απόδοσης, καθώς όλες χρησιμοποιούν και τις 8 διαθέσιμες GPU. Από την άλλη πλευρά, το *PARALiA select* (EDP_i) έχει το καλύτερο PDP_i για όλες τις διαμορφώσεις, προσφέροντας κατά μέσο όρο 8% υψηλότερο PDP_i από το *PARALiA comm_opt* με μόλις 0.5% μικρότερη μέση επίδοση. Επίσης, φαίνεται ότι η μέση βελτίωση του PDP_i μέσω της επιλογής συσκευών επηρεάζει κυρίως τα μικρότερα προβλήματα (τα κάτω μέρη των boxplots διαφέρουν περισσότερο) και εξαρτάται από το σχήμα του προβλήματος (Μέση βελτίωση: τετράγωνο = 1%, fat-thin = 8%, thin-fat = 15%). Και αυτές οι συμπεριφορές προέρχονται από το γεγονός

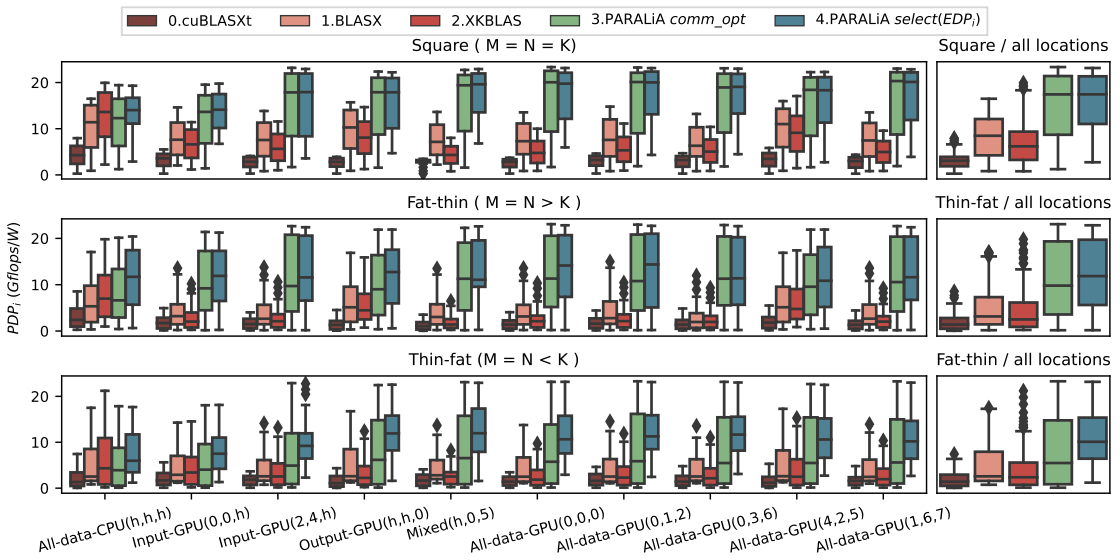


Figure 7.5: Ενεργειακή απόδοση του dgemm (Gflops/W) για όλες τις διαμορφώσεις προβλήματος που παρουσιάζονται στο Σχήμα 7.4. Οι cuBLASXt, BLASX, XKBLAS και PARALiA *comm_opt* έχουν PDP_i αντίστοιχο με την επίδοση τους (καθώς όλες χρησιμοποιούν και τις 8 GPU του συστήματος), με αποτέλεσμα ένα πολύ καλύτερο PDP_i για το PARALiA λόγω της υψηλότερης επίδοσής του. Από την άλλη πλευρά, το PARALiA *select*(EDP_i) λαμβάνει επίσης υπόψη τη σχέση ενέργειας-απόδοσης κατά την επιλογή του αριθμού συσκευών που θα χρησιμοποιηθούν και, επομένως, έχει πολύ καλύτερο PDP_i επιβάλλοντας μόνο μια μικρή διαφορά απόδοσης.

ότι η επιλογή συσκευής έχει νόημα μόνο για προβλήματα που περιορίζονται εν μέρει από την επικοινωνία, καθώς για προβλήματα που δεσμεύονται αποκλειστικά από τους υπολογισμούς, η επιλογή όλων των συσκευών θα αποφέρει πάντα το υψηλότερο EDP_i. Συνοψίζοντας, το PARALiA παρέχει την υψηλότερη ενεργειακή απόδοση για όλες τις διαμορφώσεις, συνδυάζοντας καλύτερη συνολική επίδοση με αποδοτική επιλογή συσκευών για προβλήματα περιορισμένα από την επικοινωνία.

7.3 Επέκταση PARALiA για την βελτιστοποίηση της επικοινωνίας σε πυρήνες πολλαπλασιασμού πινάκων

Στην ενότητα 7.2 επικεντρωθήκαμε στις λειτουργίες BLAS επιπέδου 3, οι οποίες είναι πιο συνηθισμένες σε συστοιχίες πολλών GPU λόγω της μεγαλύτερης αριθμητικής πολυπλοκότητάς τους. Σε αυτή την ενότητα, εμβαθύνουμε ακόμη περισσότερο επικεντρώνοντας την προσοχή μας σε έναν μόνο πυρήνα: τον γενικό πολλαπλασιασμό πινάκων (GEMM).

Γιατί επιλέξαμε τον GEMM; : *Θεωρητικά*, η εστίαση σε έναν μόνο πυρήνα περιορίζει την εφαρμοσιμότητα της προσέγγισής μας σε σχέση με τον γενικότερο στόχο του PARALiA για τις λειτουργίες BLAS επιπέδου 3. *Πρακτικά*, οι πυρήνες BLAS επιπέδου 3 είτε είναι πολλαπλασιασμοί πινάκων με διαφορετικούς τύπους/διατάξεις δεδομένων είτε εκτελούν κυρίως λειτουργίες GEMM εσωτερικά [157]. Αυτό καθιστά τον GEMM τη βασικότερη λειτουργία στις εφαρμογές υψηλής απόδοσης (HPC) και μηχανικής μάθησης (ML) και καθιστά τη βελτιστοποίησή του εξίσου σημαντική για τη γενική βελτιστοποίηση των λειτουργιών BLAS επιπέδου 3. Συνεπώς, στόχος μας είναι να πραγματοποιήσουμε μια εξαντλητική βελτιστοποίηση του GEMM, αντιμετωπίζοντας τις προκλήσεις που αφορούν τον συγκεκριμένο πυρήνα και να βελτιώσουμε τις στρατηγικές βελτιστοποίησής μας πέρα από τις πιο γενικευμένες προσεγγίσεις του PARALiA.

Περιορισμοί του PARALiA: Παρά την πρόοδο που σημειώνει το PARALiA στην βελτιστοποίηση BLAS για πολλαπλές GPU, εξακολουθεί να βρίσκεται *ενδιάμεσα* μεταξύ της απόδοσης και της ανθεκτικότητας/μεταφερσιμότητας. Για παράδειγμα, στην περίπτωση του GEMM, το PARALiA δυσκολεύεται να εκμεταλλευτεί πλήρως τις δυνατότητες των σύγχρονων κόμβων με πολλές GPU, χάνοντας 5-15% της απόδοσης σε ευνοϊκά σενάρια όπως όταν όλα τα δεδομένα είναι στη μνήμη της CPU. Η πρώτη αδυναμία του PARALiA εντοπίζεται στη βελτιστοποίηση της επικοινωνίας που προσφέρει το LinkMap. Το LinkMap λειτουργεί αντιδραστικά, λαμβάνοντας αποφάσεις δρομολόγησης κατά το χρόνο εκτέλεσης, χωρίς ολοκληρωμένη εικόνα της ροής δεδομένων. Αυτή η προσέγγιση δεν μπορεί να υποστηρίξει πρωίμη βελτιστοποίηση πέρνωντας υπόψη τη διάσπαση δεδομένων, που είναι πάντα σταθερή. Επιπλέον, το LinkMap βασίζεται αποκλειστικά στις παραμέτρους εύρους ζώνης για τις αποφάσεις δρομολόγησης, παραμελώντας το *τρέχον φορτίο* στους συνδέσμους επικοινωνίας κατά τη λήψη μιας απόφασης. Επιπροσθέτως, τα ενδιάμεσα άλματα στις αναδρομολογημένες μεταφορές που προτείνει ο αλγόριθμος βελτιστοποίησης του LinkMap αυξάνουν τον συνολικό αριθμό μεταφορών. Αυτός ο συμβιβασμός μεταξύ παραπάνω επικοινωνίας και υψηλότερου εύρους ζώνης είναι συνήθως καλύτερος για την απόδοση, αλλά εξακολουθεί να αποκλίνει από την κορυφαία δυνατή απόδοση μιας λύσης που είναι ταυτόχρονα βέλτιστη από άποψη επικοινωνίας και εύρους ζώνης. Τέλος, τα προβλήματα που επηρεάζουν περισσότερο την απόδοση του PARALiA προκύπτουν από τις τεχνικές δρομολόγησης, διάσπασης και προσωρινής αποθήκευσης που υιοθετήθηκαν από προηγούμενες εργασίες και συνδυάστηκαν με την μοντελοποίηση. Παρόλο που αυτές οι μέθοδοι έχουν βελτιωθεί με τη μοντελοποίηση απόδοσης, εξακολουθούν να περιορίζονται από την πολυπλοκότητα σχεδίασής τους και την έλλειψη πλήρους ενσωμάτωσης μεταξύ μοντελοποίησης και υλοποίησης.

Η καλύτερη προσέγγιση για τον GEMM; : Η χρήση δρομολόγησης κατά το χρόνο εκτέλεσης αποτελεί το πρότυπο για τις βιβλιοθήκες BLAS πολλαπλών GPU, καθώς πρέπει να υποστηρίζουν αυθαίρετα προβλήματα BLAS με διαφορετικές ανάγκες σε επικοινωνία, υπολογισμούς και ταυτόχρονη εκτέλεση. Ωστόσο, όταν εστιάζουμε συγκεκριμένα στον GEMM, η εκτέλεση επηρεάζεται

αποκλειστικά από τις *διαστάσεις του προβλήματος, την τοποθέτηση των δεδομένων* και τη στρατηγική *διάσπασης*. Δεδομένου ότι αυτά τα χαρακτηριστικά του προβλήματος είναι διαθέσιμα όταν καλείται η ρουτίνα και η διάσπαση πραγματοποιείται πριν την εκτέλεση, όλες οι απαιτήσεις και οι εξαρτήσεις σε επικοινωνία/υπολογισμούς μπορούν να καθοριστούν πριν την εκτέλεση. Συνεπώς, οι αποφάσεις *δρομολόγησης και επικάλυψης* μπορούν να ληφθούν *προληπτικά* πρίν την εκτέλεση, και με βάση την πραγματική επικοινωνία, όχι μια εκτίμηση μοντέλου. Επιπλέον, αυτές οι αποφάσεις μπορούν να ληφθούν λαμβάνοντας υπόψη τη διαδικασία εκτέλεσης στο σύνολό της και όχι λαμβάνοντας μεμονωμένες αποφάσεις *αντιδραστικά* κατά την εκτέλεση, ανοίγοντας την δυνατότητα για πιο σύνθετες βελτιστοποιήσεις. Δυστυχώς, ο κώδικας αυτόματης βελτιστοποίησης του PARALiA, όπως περιγράφεται στην ενότητα 7.2, είναι ανεπαρκής για αυτόν τον σκοπό, επειδή λειτουργεί ανεξάρτητα από τα τμήματα προεπεξεργασίας και δρομολόγησης. Αντίθετα, η επίτευξη του βέλτιστου αποτελέσματος απαιτεί μια συνολική προσέγγιση, η οποία να ενσωματώνει τη γνώση μοντελοποίησης στον σχεδιασμό των αλγορίθμων διάσπασης και της δρομολόγησης.

Takeaway Η μοντελοποίηση και η αυτόματη βελτιστοποίηση μπορούν να προσφέρουν μια *ενδιάμεση* λύση μεταξύ υψηλής απόδοσης και ανθεκτικότητας, αλλά υπάρχει ακόμη σημαντικός χώρος για βελτίωση όταν εφαρμόζονται στον GEMM.

Συνεισφορές: Σε αυτή την ενότητα, στοχεύουμε σε μια βελτιστοποιημένη βιβλιοθήκη GEMM για πολλαπλές GPU που επιτυγχάνει βέλτιστη επίδοση και ανθεκτικότητα. Η προσέγγισή μας βασίζεται σε έναν στατικό δρομολογητή που υπολογίζεται πριν την εκτέλεση, κάθε φορά που καλείται μια ρουτίνα GEMM με ένα νέο σύνολο παραμέτρων. Κάθε στατικό σχέδιο που δημιουργείται για ένα συγκεκριμένο πρόβλημα είναι επαναχρησιμοποιήσιμο για τις επόμενες κλήσεις ρουτίνας, μηδενίζοντας τα έξοδα δρομολόγησης για όλες εκτός από την πρώτη κλήση. Η γνώση των παραμέτρων εισόδου μας δίνει μια πλήρη εικόνα του μοτίβου επικοινωνίας και των χαρακτηριστικών δρομολόγησης της ρουτίνας, την οποία αξιοποιούμε για να ελαχιστοποιήσουμε ταυτόχρονα τον όγκο επικοινωνίας, να μεγιστοποιήσουμε την αξιοποίηση του δικτύου διασύνδεσης, να βελτιστοποιήσουμε την επικάλυψη και να ελαχιστοποιήσουμε την ανισορροπία και τον χρόνο αδράνειας των GPU. Σημειώνουμε ότι οι σύγχρονες βιβλιοθήκες πολλαπλών GPU [9, 57, 58, 157] ήδη υλοποιούν ορισμένες από τις βελτιστοποιήσεις που εισάγονται σε αυτήν την ενότητα, όπως την διατήρηση δεδομένων (*caching*) (Ενότητα 7.3.1.3), την *επικάλυψη* (Ενότητα 7.3.1.2) και τη *δρομολόγηση βάσει εύρους ζώνης* (Ενότητα 7.3.1.4). Η εργασία μας βελτιώνει αυτές τις βελτιστοποιήσεις, χρησιμοποιώντας ένα απλούστερο σχήμα caching, επιτυγχάνοντας πλήρη (αντί για μερική) επικάλυψη όλων των ροών υπολογισμών/επικοινωνίας και μειώνοντας τα κόστη έναρξης μέσω στατικής δρομολόγησης. Επιπλέον, η εργασία μας εισάγει

νέες βελτιστοποιήσεις όπως τη *δρομολόγηση βάσει ETA* (Ενότητα 7.3.1.4), το *RONLY-fetch batching* (Ενότητα 7.3.1.5), το *lazy WR-tile fetching* (Ενότητα 7.3.1.6) και την *επιλογή σειράς υποπυρήνων βάσει ETA* (Ενότητα 7.3.1.7).

7.3.1 Υλοποίηση

Αρχικά, περιγράφουμε το σχεδιασμό ενός στατικού χρονοδιαγράμματος εμπνευσμένου από προσεγγίσεις κατανεμημένων και multi-GPU συστημάτων, που βελτιστοποιούν το GEMM για multi-GPU. Από τις διαθέσιμες διατάξεις εισόδου BLAS ή PBLAS για το GEMM, ακολουθούμε το πρότυπο BLAS [39], με τους πίνακες εισόδου να αποθηκεύονται σε διάταξη LAPACK, είτε στη μνήμη του host είτε στων GPU [9, 57, 124, 157]. Μια ρουτίνα GEMM με βάση το πρότυπο BLAS εκτελεί την πράξη:

$$C_{out} = a \cdot A \times B + b \cdot C_{in} \tag{7.6}$$

Υποθέτοντας ότι το C_{out} αποθηκεύεται στη θέση του buffer C_{in}, η πράξη απαιτεί τρεις πίνακες, $A(M \times K)$, $B(K \times N)$ και $C(M \times N)$, με τους A και B να είναι μόνο ανάγνωσης (στο εξής αναφερόμενοι ως *RONLY*) και τον C να είναι ανάγνωσης και εγγραφής (στο εξής αναφερόμενος ως *WR*).

7.3.1.1 Ιεραρχική αποσύνθεση

Η αποσύνθεση των πινάκων A, B και C καθορίζει τα *υποπροβλήματα* που θα εκτελεστούν σε κάθε GPU και το μοτίβο επικοινωνίας. Στην υλοποιησή μας, επιλέγουμε να αποφύγουμε την κοινή χρήση του πίνακα C μεταξύ των GPU, καθώς αυτό οδηγεί σε επιπλέον συγχρονισμό μεταξύ των GPU και πρόσθετη επικοινωνία για την εκτέλεση [9, 57, 157]. Έτσι, επιλέγουμε μια δισδιάστατη αποσύνθεση παρόμοια με τον αλγόριθμο SUMMA [151], ως μια απλή και πρακτική λύση, πάνω στην οποία εφαρμόζουμε μια σειρά βελτιστοποιήσεων.

Το Σχήμα 7.6 απεικονίζει την προτεινόμενη *ιεραρχική αποσύνθεση* βασισμένη στον κατανεμημένο αλγόριθμο SUMMA [151]. Το πρώτο επίπεδο αποσύνθεσης βασίζεται στον αριθμό των GPU, τα οποία απεικονίζονται ως δισδιάστατο πλέγμα $r \times c$ GPUs. Αντίστοιχα, αποσυνθέτουμε τον πίνακα C σε τετράγωνα tile (από εδώ και στο εξής *tiles*) $M_r \times N_c$, δημιουργώντας υποπροβλήματα ανάλογα, και αναθέτουμε κάθε υποπρόβλημα ξεχωριστά σε μια GPU. Στη συνέχεια, αποσυνθέτουμε τους πίνακες A και B σε tile μήκους γραμμών ($M_r \times K$) και tile πλάτους στηλών ($K \times N_c$), αντίστοιχα. Τα tile γραμμών του πίνακα A ανατίθενται σε c GPUs και τα tile στηλών του πίνακα B σε r GPUs. Σε ένα δεύτερο επίπεδο, τα κομμάτια των A, B και C σε κάθε GPU αποσυντίθενται περαιτέρω σε δισδιάστατα tiles μεγέθους $T \times T$, με εφαρμογή padding όπου απαιτείται. Αυτό δημιουργεί ένα τρισδιάστατο πλέγμα $\frac{M}{T} \times \frac{N}{T} \times \frac{K}{T}$ τετραγωνικών υποπροβλημάτων GEMM που απαιτούν διαφορετικά tiles εισόδου και εξόδου. Έτσι, για να υπολογίσουμε το αποτέλεσμα της ρουτίνας GEMM C_{out}, πρέπει να υπολογίσουμε το tile-based

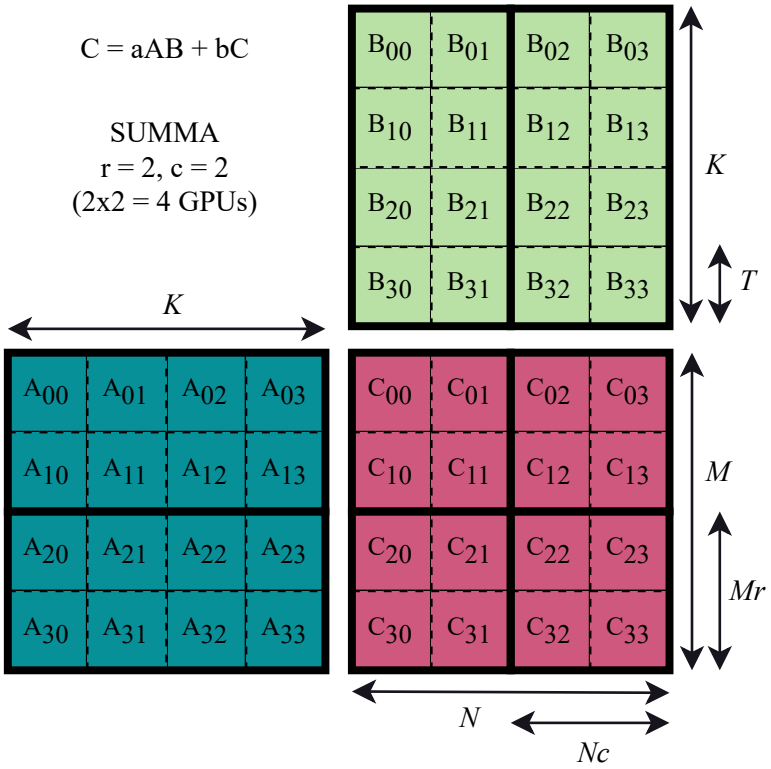


Figure 7.6: Ένα παράδειγμα της ιεραρχικής αποσύνθεσης GEMM 2 επιπέδων για ένα τετράγωνο πρόβλημα (M, N, K) βασισμένο στον αλγόριθμο αποκλεισμού SUMMA [151]. Το πρώτο επίπεδο εξαρτάται από τον αριθμό των επεξεργαστών (εδώ: 4 *GPUs*) κατανεμημένων σε δισδιάστατο πλέγμα $(r, c) = (2, 2)$ που αποσυνθέτει το (M, N) σε κομμάτια (M_r, N_c), αφήνοντας το K αμετάβλητο. Στη συνέχεια, το δεύτερο επίπεδο είναι η αποσύνθεση του (M, N, K) σε δισδιάστατα τετράγωνα *tiles*, που δημιουργούν τετράγωνα υποπροβλήματα GEMM και επιτρέπουν την επικάλυψη επικοινωνίας/υπολογισμών.

outer-product [151]:

$$C_{i,j} = b \cdot C_{i,j} + \sum_{k=0}^{\frac{K}{T}} a \cdot A_{i,k} \times B_{k,j} \tag{7.7}$$

για $i = 0 \rightarrow \frac{M}{T}$ και $j = 0 \rightarrow \frac{N}{T}$.

Η τιμή του T πρέπει 1) να αποφεύγει το υπερβολικό padding, 2) να είναι αρκετά μικρή ώστε να δημιουργεί επαρκή αριθμό υποπροβλημάτων για επικάλυψη και 3) να είναι αρκετά μεγάλη ώστε να αποφεύγονται καθυστερήσεις στην εκτέλεση, τις μεταφορές και τη δρομολόγηση [7, 9, 63, 65, 159]. Για να διασφαλίσουμε την ισορροπία μεταξύ των τριών, ελαχιστοποιούμε μια συνάρτηση κόστους βασισμένη σε τρεις ευρετικές μεθόδους. Η πρώτη ευρετική μέθοδος προσπαθεί να αποφύγει το padding, επιβαρύνοντας κάθε υπόλοιπο κατά τη διαίρεση των M, N, K

σε tiles:

$$C_{padding}(T) = \sum_{i \in \{M,N,K\}, i \nmid 0} \frac{i}{T}$$

Η δεύτερη ευρετική μέθοδος αφορά την ικανότητα επικάλυψης και επιβαρύνει το ποσοστό του προβλήματος που δεν μπορεί να επικάλυφθεί, εκτιμώμενο ως το αντίστροφο του μήκους του pipeline επικάλυψης, που είναι ίσο με τον αριθμό των υποπροβλημάτων ανά GPU:

$$C_{overlap}(T) = \frac{gpu\_num}{\frac{M}{T} \times \frac{N}{T} \times \frac{K}{T}}$$

Η τελευταία ευρετική μέθοδος χρησιμοποιεί ένα ελάχιστο προεπιλεγμένο μέγεθος tile T_{min} (default = 2048), αρκετά μεγάλο για να αποφεύγονται οι υψηλές καθυστερήσεις, και επιβαρύνει τα μικρότερα tiles αναλογικά:

$$C_{latency}(T) = \frac{T_{min} \times 0.2}{T}$$

Εφαρμόζουμε αυτές τις ευρετικές μεθόδους πριν από την αποσύνθεση του δεύτερου επιπέδου επιλέγοντας το μέγεθος tile T που ελαχιστοποιεί το $C_{total}(T) = C_{padding}(T) + C_{overlap}(T) + C_{latency}(T)$, για όλα τα $T = 128 \rightarrow min(M_r, N_c, K)$.

7.3.1.2 Επικάλυψη επικοινωνίας/υπολογισμών

Μετά την αποσύνθεση, η διαδικασία εκτέλεσης ενός υποπροβλήματος GEMM απαιτεί τη λήψη των εξαρτήσεων εισόδου, δηλαδή των απαραίτητων tile εισόδου A_T, B_T, και C_T, τον υπολογισμό του kernel και, ενδεχομένως, την εγγραφή του αποτελέσματος C_T στη θέση του αρχικού πίνακα. Επομένως, θεωρούμε κάθε υποπρόβλημα ως μια διαδικασία πέντε εργασιών: οι εργασίες (1)-(3) αφορούν τη λήψη των εξαρτήσεων εισόδου, $fetch^{dst}_{src}(A_T)$, $fetch^{dst}_{src}(B_T)$, $fetch^{dst}_{src}(C_T)$, η εργασία (4) αφορά τον υπολογισμό του kernel $compute$, και η εργασία (5) αφορά την εγγραφή του αποτελέσματος $WB^{src}_{dst}(C_T)$. Αυτά τα υποπροβλήματα επιτρέπουν την παράλληλη επεξεργασία τόσο εντός όσο και μεταξύ των GPU, καθώς μπορούν να δρομολογηθούν σε διαφορετικά streams.

Το Σχήμα 7.7 απεικονίζει ένα απλοποιημένο παράδειγμα δρομολόγησης των τεσσάρων πρώτων υποπροβλημάτων της προαναφερόμενης αποσύνθεσης του Σχήματος 7.6 στις gpu_0 και gpu_1, αντίστοιχα. Υποθέτουμε ότι οι πίνακες A, B και C βρίσκονται αρχικά στην ίδια τοποθεσία ($A_{loc} = B_{loc} = C_{loc}$). Η τοποθέτηση των διαφορετικών τύπων εργασιών ($fetch$, $compute$, και WB) σε διαφορετικά streams επιτρέπει την επικάλυψή τους σε μορφή pipeline. Αυτό μειώνει τον συνολικό χρόνο για ένα υποπρόβλημα από $t_{total} = t_{fetch(A)} + t_{fetch(B)} + t_{fetch(C)} + t_{compute} + t_{WB(C)}$ σε περίπου $t_{total} \approx max((t_{fetch(A)} + t_{fetch(B)} + t_{fetch(C)}), t_{compute}, t_{WB}(C))$ [159]. Στην υλοποίησή μας η επικάλυψη λειτουργεί με παρόμοιο τρόπο, αλλά εισάγουμε τις εργασίες μεταφοράς δεδομένων (*fetch*, *WB*) σε $(gpu\_num + 1)^2$ διαφορετικά CUDA streams,

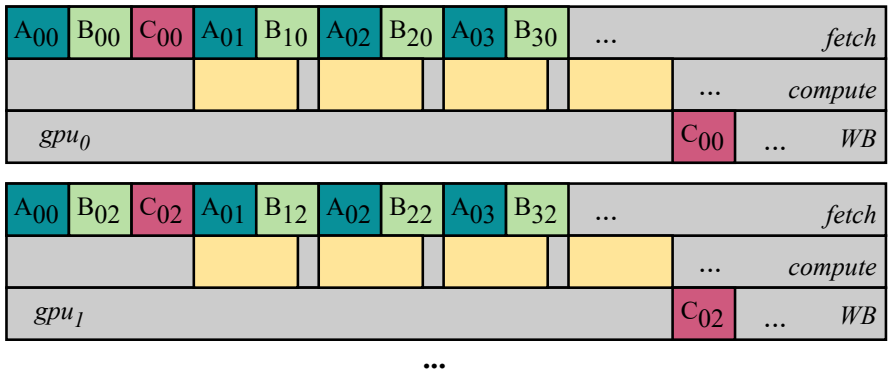


Figure 7.7: Η διαδικασία εκτέλεσης των τεσσάρων πρώτων υποπροβλημάτων του σχήματος 7.6 σε δύο GPU. Οι εργασίες διαφορετικών τύπων (*fetch*, *compute*, *WB*) τοποθετούνται σε διαφορετικά streams και επικαλύπτονται σε ένα pipeline για κάθε GPU, χρησιμοποιώντας διαφορετικά streams.

επιτρέποντας ταυτόχρονη αμφίδρομη επικοινωνία μεταξύ όλων των συσκευών και της μνήμης του host. Συνεπώς, αν τα A_{loc}, B_{loc} και C_{loc} είναι ξεχωριστές τοποθεσίες/μνήμες συσκευών, οι εργασίες *fetch* επίσης επικαλύπτονται, με αποτέλεσμα ο συνολικός χρόνος t_{total} να είναι περίπου $t_{total} \approx max(t_{fetch(A)}, t_{fetch(B)}, t_{fetch(C)}, t_{compute}, t_{WB(C)})$. Για την επικάλυψη επικοινωνίας/υπολογισμών, δρομολογούμε τις εργασίες *compute*, που εκτελούνται χρησιμοποιώντας kernels cuBLAS [122], σε έναν διαμορφώσιμο αριθμό CUDA streams ανά GPU (default = 8). Τέλος, οι εξαρτήσεις μεταξύ των streams ορίζονται και τηρούνται με CUDA events [126], παρόμοια με προηγούμενες προσεγγίσεις [9, 57, 157].

7.3.1.3 Προσωρινή αποθήκευση δεδομένων / Αποφυγή επικοινωνίας

Για την αποφυγή αχρείαστης επικοινωνίας, οι περισσότερες υλοποιήσεις multi-GPU BLAS [9, 57, 157] *προσωρινά αποθηκεύουν* τα tile που *μεταφέρονται* στη μνήμη της GPU για ένα συγκεκριμένο υποπρόβλημα, ώστε να επαναχρησιμοποιηθούν από επόμενα υποπροβλήματα. Η προσωρινή αποθήκευση είναι απαραίτητη για μεγέθη προβλημάτων που δεν χωράνε στη μνήμη της GPU και σημαντική για την απόδοση, καθώς μειώνει τον όγκο της επικοινωνίας, αλλά η διαχείρισή της προσθέτει επιπλέον υπολογιστικό κόστος. Για να αποφύγουμε αυτό το κόστος, δεν υλοποιούμε έναν πολύπλοκο μηχανισμό προσωρινής αποθήκευσης. Αντί αυτού, χρησιμοποιούμε έναν buffer ανά GPU, που σημειώνεται ως $SoftBuf[i]$, με $i = 0 \rightarrow gpu\_num$. Αυτός ο buffer δεσμεύεται κάθε φορά που καλείται για πρώτη φορά μια ρουτίνα GEMM, προσαρμοσμένος στις απαιτήσεις μνήμης της αποσύνθεσης του Σχήματος 7.6 για αυτό το συγκεκριμένο πρόβλημα. Ο buffer αποθηκεύει τα απαραίτητα tile καθ' όλη τη διάρκεια ζωής της ρουτίνας. Εάν μια επόμενη ρουτίνα GEMM έχει μεγαλύτερες απαιτήσεις μνήμης, ο buffer αυτόματα αναπροσαρμόζεται.

Εκφόρτωση προβλημάτων που υπερβαίνουν τη χωρητικότητα μνήμης: Οι σύγχρονοι multi-GPU κόμβοι διαθέτουν μεγάλες μνήμες host (της τάξης των TB), που ξεπερνούν κατά πολύ τη χωρητικότητα της μνήμης της GPU (της τάξης των δεκάδων GBs). Για την ειδική περίπτωση μεγάλων προβλημάτων, όπου οι πίνακες A, B, και C βρίσκονται όλοι στη μνήμη του host και οι απαιτήσεις μνήμης του $SoftBuf$ υπερβαίνουν τη χωρητικότητα της μνήμης της GPU, χρησιμοποιούμε ένα επιπλέον επίπεδο αποσύνθεσης στην πλευρά του host, πριν εφαρμόσουμε την ιεραρχική αποσύνθεση του Τμήματος 7.3.1.1. Αυτή η αποσύνθεση ενεργοποιείται αυτόματα κατά την κλήση της ρουτίνας, όταν το μέγεθος του προβλήματος θα είχε ως αποτέλεσμα οποιοδήποτε $SoftBuf[i]$ να είναι μεγαλύτερο από ένα προκαθορισμένο ποσοστό (default = 80%) της διαθέσιμης μνήμης GPU στο gpu_i. Αποσυνθέτουμε τις αρχικές διαστάσεις του προβλήματος M_L, N_L, K_L με τρισδιά- στατο τρόπο σε τετράγωνα tile μεγέθους T_L, δημιουργώντας ένα τρισδιάστατο πλέγμα υποπροβλημάτων GEMM μεγέθους $M \times N \times K$, με $M = N = K = T_L$. Επιλέγουμε το μέγιστο T_L που ικανοποιεί την απαίτηση μνήμης για τον buffer $SoftBuf$. Στη συνέχεια, δρομολογούμε κάθε υποπρόβλημα διαδοχικά στις GPU, με κάθε υποπρόβλημα να αξιοποιεί όλες τις βελτιστοποιήσεις που περιγράφονται σε αυτό το έργο. Σημειώνουμε ότι αυτό περιορίζει την αναλογία επικοινωνίας-υπολογισμού και, κατά συνέπεια, τη συνολική απόδοση για το πρόβλημα (M_L, N_L, K_L) σε αυτή του προβλήματος (T_L, T_L, T_L).

7.3.1.4 Δρομολόγηση Επικοινωνίας

Ο κύριος όγκος επικοινωνίας στο multi-GPU GEMM προέρχεται από τα tile read-only (RONLY) A_T, B_T των πινάκων A και B, τα οποία πρέπει να μεταφερθούν σε πολλές GPU, όπου θα εκτελεστούν τα αντίστοιχα υποπροβλήματα. Κατά την εκτέλεση, η πρώτη μεταφορά κάθε RONLY tile, έστω A_T, σε μια gpu_i απαιτεί μεταφορά από την αρχική θέση του πίνακα in_loc, έστω $fetch^{gpu_i}_{in\_loc}(A_T)$, που αποθηκεύει το A_T στο $SoftBuf[gpu_i]$. Από την άλλη, οι επόμενες μεταφορές tile σε οποιαδήποτε άλλη συσκευή gpu_j μπορούν είτε να μεταφέρουν ένα αντίγραφο του tile από το in_loc είτε από την gpu_i, κάτι που απαιτεί μια απόφαση δρομολόγησης επικοινωνίας.

Δρομολόγηση βάσει Εύρους Ζώνης: Η πιό κοινή προσέγγιση για τη δρομολόγηση της επικοινωνίας είναι η χρήση του εύρους ζώνης μεταξύ των διαφορετικών τοποθεσιών μνήμης, βασισμένη στην τοπολογία του δικτύου διασύνδεσης [9, 58, 157]. Η επιλογή της βέλτιστης διαδρομής βασίζεται στο διαθέσιμο εύρος ζώνης. Για το παραπάνω παράδειγμα, όπου το A_T μπορεί να βρίσκεται τόσο στο in_loc όσο και στο gpu_i, η απόφαση εξαρτάται από τη σύγκριση του $BW^{gpu_i}_{in\_loc}$ και του $BW^{gpu_j}_{gpu_i}$ και την επιλογή της διαδρομής με το υψηλότερο εύρος ζώνης. Εκτιμούμε το εύρος ζώνης των διαφορετικών $(gpu\_num+1)^2$ διαδρομών/ροών εμπειρικά με μικροδοκιμές, όπως στο [9].

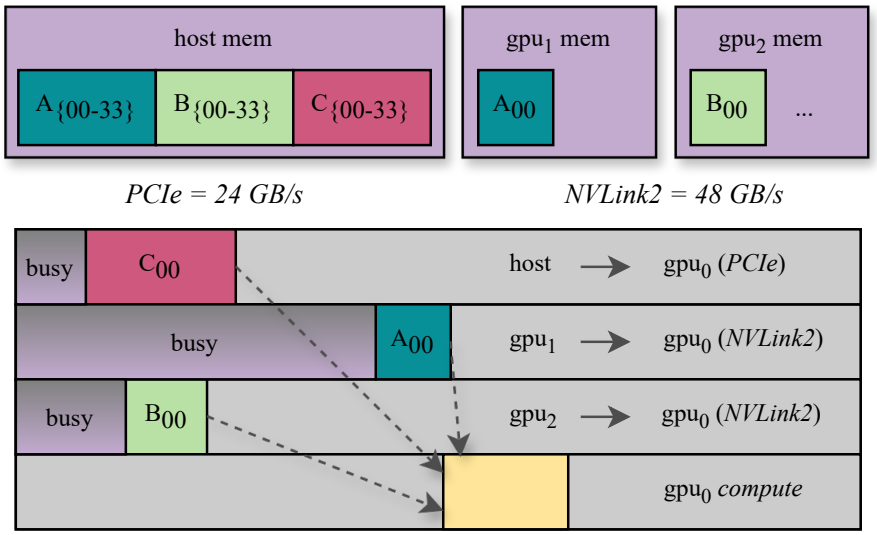


Figure 7.8: Παράδειγμα λάθους πρόβλεψης στη δρομολόγηση βάσει εύρους ζώνης που οδηγεί σε αυξημένο χρόνο αδράνειας της GPU. Όταν το υποπρόβλημα με τις εισαγόμενες εξαρτήσεις (A_{00}, B_{00}, C_{00}) προγραμματίζεται στη gpu_0, τα A_{00} και B_{00} είναι ήδη διαθέσιμα στις gpu_1 και gpu_2 από προηγούμενες μεταφορές tile. Δεδομένου ότι η δρομολόγηση βάσει εύρους ζώνης αγνοεί το φορτίο των διασυνδέσεων, αντιγράφει το A_{00} από την gpu_1 και το B_{00} από την gpu_2, καθώς αυτές οι μεταφορές χρησιμοποιούν υψηλότερο εύρος ζώνης P2P GPU. Στην περίπτωση του A_{00}, αυτό έχει ως αποτέλεσμα η gpu_0 να μπλοκάρει την εκτέλεση για περισσότερο χρόνο από ό,τι αν το A_{00} είχε μεταφερθεί από το host, λόγω του υψηλού υπάρχοντος φορτίου στη σύνδεση $gpu_1 \rightarrow gpu_0$.

Συνυπολογισμός του φόρτου: Ενώ η δρομολόγηση βάσει εύρους ζώνης μπορεί να είναι αποτελεσματική, καθώς αυξάνει τη μέση αξιοποίηση του εύρους ζώνης, ο στόχος της δρομολόγησης επικοινωνίας είναι να ελαχιστοποιήσει τον εκτιμώμενο χρόνο άφιξης (ETA) ενός tile, δηλαδή τον συνολικό χρόνο μεταφοράς των δεδομένων εισόδου στην προοριζόμενη GPU, έτσι ώστε η εργασία *compute* να ξεκινήσει το συντομότερο δυνατόν. Η δρομολόγηση βάσει εύρους ζώνης δεν λαμβάνει υπόψη το υπάρχον φορτίο σε μια *σύνδεση*, το οποίο μπορεί να καθυστερήσει τον χρόνο άφιξης αυτού του tile. Ένα παράδειγμα απεικονίζεται στο Σχήμα 7.8, όπου το A_{00} πρέπει να μεταφερθεί στη gpu_0. Η δρομολόγηση βάσει εύρους ζώνης επιλέγει να μεταφέρει το tile από την gpu_1, καθώς το $BW_{gpu_1}^{gpu_0} > BW_{host}^{gpu_0}$, χωρίς να λαμβάνει υπόψη το υψηλό φορτίο σε αυτή τη ροή από προηγούμενες αντιγραφές, με αποτέλεσμα την καθυστερημένη άφιξη του A_{00} στη gpu_0 και την αδράνεια της gpu_0.

Για να αποφύγουμε αυτό το φαινόμενο, βελτιστοποιούμε τη δρομολόγηση της επικοινωνίας λαμβάνοντας υπόψη το φορτίο των διασυνδέσεων κατά την επιλογή της διαδρομής για μια μεταφορά tile. Για να το επιτύχουμε αυτό, ορίζουμε έναν δισδιάστατο πίνακα για τη διαθεσιμότητα των συνδέσεων (εφεξής σημειώνεται ως *LAM*), ο οποίος αποθηκεύει τον εκτιμώμενο χρόνο

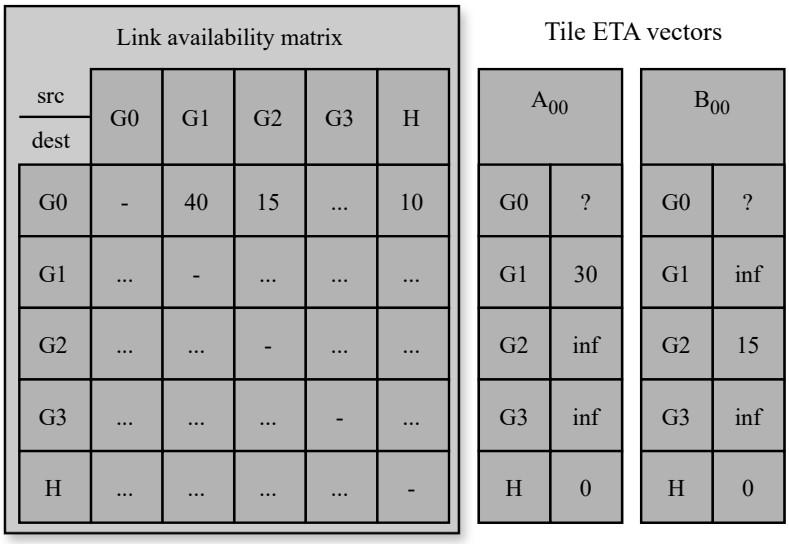


Figure 7.9: Ένα παράδειγμα κατάστασης των διανυσμάτων LAM και ETA για δύο tile A_{00} και B_{00}. Τα διανύσματα ETA των tile δείχνουν πότε αυτά τα tile θα πρέπει να είναι διαθέσιμα στις GPU 1 και 2, αντίστοιχα (αρχικά στη μνήμη του host - ETA[h] = 0), ενώ το LAM αποθηκεύει μια εκτίμηση για το φορτίο των διασυνδέσεων μέχρι αυτή την κατάσταση προγραμματισμού.

που κάθε $src \rightarrow dst$ ροή επικοινωνίας θα είναι διαθέσιμη (δηλαδή χωρίς υπόλοιπο φορτίο). Επιπλέον, ορίζουμε το *διάνυσμα ETA* για κάθε αποσπασμένο tile, το οποίο αποθηκεύει τον εκτιμώμενο χρόνο που ένα έγκυρο αντίγραφο αυτού του tile θα φτάσει σε κάθε τοποθεσία μνήμης. Αρχικά, όλα τα πεδία του LAM τίθενται στο μηδέν. Όλα τα πεδία του διανύσματος ETA τίθενται στο inf, εκτός από την αρχική τοποθεσία δεδομένων του tile, που τίθεται στο μηδέν ($ETA[in\_loc] = 0$). Κατά τον προγραμματισμό, όταν ο scheduler πρέπει να πάρει μια απόφαση δρομολόγησης για να μεταφέρει ένα tile μεγέθους $size$ bytes στον dst, ο scheduler συνδυάζει τα LAM και ETA διανύσματα με το εκτιμώμενο κόστος μεταφοράς $t_i^{dst} = \frac{size}{BW_i^{dst}}$, αναζητώντας την πηγή i με το χαμηλότερο ETA_{min}, όπου:

$$ETA_{min} = \min_{i=0}^{gpu\_num+1} (\max(ETA[i], LAM[dest][i]) + t_i^{dst})$$

Στη συνέχεια, το LAM και το διάνυσμα ενημερώνονται για τη νέα μεταφορά σε $LAM[dest][i] = ETA[dest] = ETA_{min}$. Η προαναφερθείσα ενημέρωση του LAM γίνεται επίσης για τις μεταφορές των tile του πίνακα C, λαμβάνοντας υπόψη το φορτίο των διασυνδέσεων, αλλά χωρίς βελτιστοποίηση δρομολόγησης.

Το Σχήμα 7.9 δείχνει το LAM για το παράδειγμα στο Σχήμα 7.8 πριν από τη δρομολόγηση των μεταφορών, με δύο παραδείγματα κατάστασης διανυσμάτων ETA για τα tile A_{00} και B_{00}. Ακολουθώντας τον αλγόριθμο δρομολόγησης βάσει ETA, το C_{00} θα μεταφερθεί από τη μνήμη

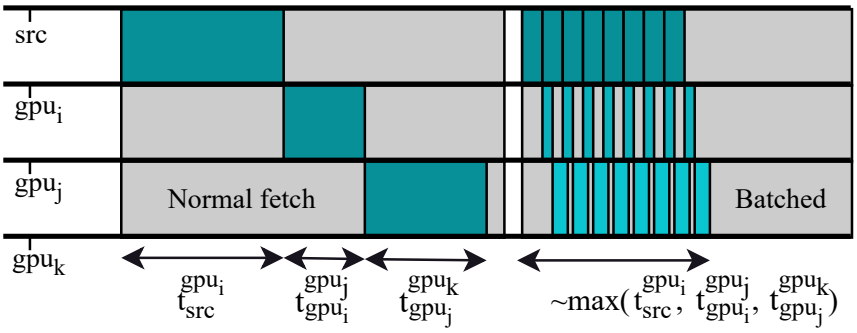


Figure 7.10: Παράδειγμα εκτέλεσης τριών fetch λειτουργιών του ίδιου δεδομένου προς τρεις GPU είτε ξεχωριστά (αριστερά) είτε με μια ταυτόχρονη ομαδοποιημένη fetch λειτουργία (δεξιά) που χρησιμοποιεί $p = 8$ υπο-μεταφορές για να επικαλύψει τη διαδικασία με σωλήνωση. Η ομαδοποίηση όλων των fetch λειτουργιών μαζί έχει ως αποτέλεσμα το ίδιο κόστος fetch για την gpu_i, αλλά μειώνει σημαντικά το κόστος fetch για τις gpu_j και gpu_k.

του host (καθώς είναι διαθέσιμο μόνο εκεί), το B_{00} θα μεταφερθεί από την gpu_2 δεδομένου ότι $\max(LAM[0][2], ETA\_B_{00}[2]) + t^0_2 < \max(LAM[0][h], ETA\_B_{00}[h]) + t^0_h$, και το A_{00} θα μεταφερθεί από το host καθώς $\max(LAM[0][h], ETA\_A_{00}[h]) + t^0_h < \max(LAM[0][1], ETA\_A_{00}[1]) + t^0_1$. Η δρομολόγηση βάσει ETA παρέχει τη βέλτιστη απόφαση με βάση τη γνώση του παρελθόντος και του παρόντος (π.χ. φορτίο και εύρος ζώνης) για τη μεταφορά κάθε RONLY tile.

7.3.1.5 Βελτιστοποίηση Μεταφορών RONLY tile με Ομαδοποίηση

Η συνηθισμένη προσέγγιση για τη δρομολόγηση επικοινωνίας σε προηγούμενες εργασίες [9, 57, 58, 157] είναι η δυναμική βελτιστοποίηση της διαδρομής κάθε *fetch* task ξεχωριστά, κατά τον προγραμματισμό της εκτέλεσής της. Αντίθετα, το στατικό χρονοδιάγραμμα μας, το οποίο κατασκευάζεται πριν από την εκτέλεση, μας επιτρέπει να ομαδοποιούμε τις μεταφορές tile σε διαφορετικές GPU με μια ταυτόχρονη *fetch* λειτουργία τύπου broadcast προς πολλαπλές GPU, π.χ. $fetch^{gpu_i,gpu_j,\cdots}_{in\_loc}(T)$.

Το Σχήμα 7.10 δείχνει ένα παράδειγμα λειτουργίας ομαδοποιημένης fetch λειτουργίας. Χωρίζουμε τη μεταφορά του ίδιου πλακιδίου σε τρεις τοποθεσίες σε p (προεπιλογή = 8) μικρότερες μεταφορές και τις επικαλύπτουμε εσωτερικά με σωλήνωση. Αυτό μειώνει το συνολικό κόστος του $fetch^{gpu_i,gpu_j,gpu_k}_{src}(T)$ από $t_{fetch} = t^{gpu_i}_{src} + t^{gpu_j}_{gpu_i} + t^{gpu_k}_{gpu_j}$ σε $t_{fetch} \approx \max(t^{gpu_i}_{src}, t^{gpu_j}_{gpu_i}, t^{gpu_k}_{gpu_j})$, μειώνοντας σημαντικά το κόστος fetch για όλες τις GPU εκτός από τον πρώτο προορισμό (gpu_i). Αυτή η βελτιστοποίηση έχει μεγαλύτερο αντίκτυπο καθώς αυξάνεται ο αριθμός των GPU, λόγω της περισσότερης κοινής χρήσης δεδομένων μεταξύ των GPU. Για παράδειγμα, σε ένα σύστημα 4-GPU, τα tile του πίνακα A μοιράζονται μεταξύ 2 GPU και απαιτούν $fetch^{gpu_i,gpu_j}_{src}(A_T)$ λειτου-

ργίες, ενώ σε ένα σύστημα 8-GPU μοιράζονται από 4 και απαιτούν $fetch_{src}^{gpu_i,gpu_j,gpu_k,gpu_l}(A_T)$ λειτουργίες. Εφαρμόζουμε αυτήν την προσέγγιση σε όλα τα RONLY tile των A και B.

Δρομολόγηση Ομαδοποιημένων Fetch Λειτουργιών: Εκτός από τη μείωση του κόστους fetch, η ομαδοποίηση των μεταφορών RONLY tile είναι επωφελής για τη δρομολόγηση της επικοινωνίας, καθώς ανοίγει επιπλέον επιλογές διαδρομών για κάθε μεταφορά. Στις ομαδοποιημένες fetch λειτουργίες, η σειρά δεν είναι σημαντική (π.χ. $fetch_{src}^{gpu_i,gpu_j}(T) \approx fetch_{src}^{gpu_j,gpu_i}(T)$), καθώς τα κόστη fetch είναι ισορροπημένα για όλους τους προορισμούς. Συνεπώς, εφαρμόζουμε τη δρομολόγηση βάσει LAM ETA από την Ενότητα 7.3.1.4 ως εξής: όταν μια ομαδοποιημένη λειτουργία $fetch_{src}^{gpu_i,gpu_j,\cdots}(T)$ προγραμματίζεται για δρομολόγηση, εξετάζουμε τα βήματα της ομαδοποιημένης λειτουργίας (π.χ. $fetch_{src}^{gpu_i}$, $fetch_{gpu_i}^{gpu_j}(T), ...$) και εφαρμόζουμε τον αλγόριθμο εκτίμησης LAM ETA επαναληπτικά, για όλους τους δυνατούς συνδυασμούς σειράς, επιλέγοντας τη σειρά $gpus\_best\_order$ που έχει το ελάχιστο ETA. Στη συνέχεια, ενημερώνουμε όλους τους ενδιάμεσους συνδέσμους LAM και όλους τους προορισμούς ETA tile με βάση το ETA_{min} της επιλεγμένης σειράς και προγραμματίζουμε τη μεταφορά της ομαδοποιημένης fetch λειτουργίας ($fetch_{src}^{\{gpus\_best\_order\}}$).

Για να δείξουμε τη σημασία αυτής της βελτιστοποίησης για τον multi-GPU GEMM, το Σχήμα 7.11 συγκρίνει την δρομολόγηση που χρησιμοποιείται σε προηγούμενες εργασίες με τη δική μας προδραστική δρομολόγηση βάσει ETA + ομαδοποιημένη fetch λειτουργία RONLY για τα πρώτα 4 προγραμματισμένα υποπροβλήματα (ένα σε κάθε GPU), εξαιρώντας τις fetch λειτουργίες tile C από τη σωλήνωση (περισσότερα για αυτό στην Ενότητα 7.3.1.6). Η προσέγγισή μας μειώνει σημαντικά τον χρόνο αδράνειας των GPU (κατά 45% κατά μέσο όρο, 60% στην καλύτερη περίπτωση), επικαλύπτει εσωτερικά όλες τις ροές επικοινωνίας, αποφεύγει μπλοκαρίσματα λόγω εξαρτήσεων μεταφοράς και παρέχει μια τέλεια ισορροπημένη εκτέλεση σε όλες τις GPU.

7.3.1.6 Βελτιστοποίηση μεταφορών tiles WR με καθυστερημένη φόρτωση

Η βελτιστοποίηση δρομολόγησης και ομαδοποίησης που συζητήθηκε στις προηγούμενες υποενότητες επικεντρώνεται στα RONLY tiles, τα οποία φορτώνονται σε πολλαπλές GPU. Από την άλλη πλευρά, τα WR tiles του πίνακα C είναι αποκλειστικά για κάθε GPU (βλ. Εικόνα 7.6) και αποτελούν έναν όγκο φόρτωσης που δεν μπορεί να μειωθεί, ο οποίος στην χειρότερη περίπτωση (χωρίς cache) για τετράγωνους πίνακες μπορεί να φτάσει το 1/3 του συνολικού όγκου φόρτωσης. Εφόσον η βελτιστοποίηση ομαδοποιημένης φόρτωσης δεν ισχύει σε αυτή την περίπτωση, επιλέγουμε να περιορίσουμε τον χρόνο αδράνειας της GPU με την *καθυστέρηση* της μεταφοράς των tiles C, για να επιτύχουμε καλύτερη επικάλυψη υπολογισμού-επικοινωνίας. Για να το επιτύχουμε αυτό, αποσυνθέτουμε την αρχική λειτουργία GEMM της Εξίσωσης 7.6 σε:

$$C' = a \cdot A \times B\ (GEMM),\ C_{out} = b \cdot C_{in} + C'\ (AXPY)$$

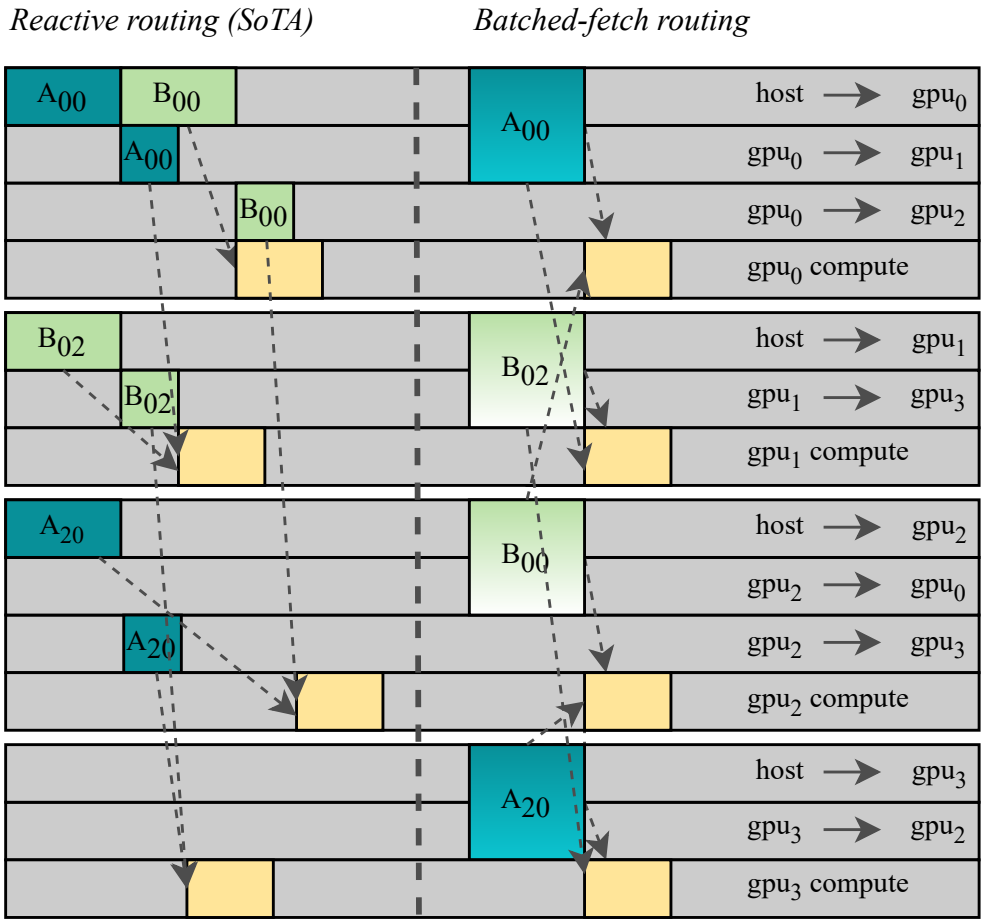


Figure 7.11: Παράδειγμα δρομολόγησης του πρώτου υποπροβλήματος GEMM σε κάθε GPU χρησιμοποιώντας α) την δρομολόγηση (αριστερά) που χρησιμοποιείται από προηγούμενες υλοποιήσεις και β) την δρομολόγηση βάσει ETA συνδυασμένη με ομαδοποιημένη fetch λειτουργία RONLY (δεξιά) που χρησιμοποιείται σε αυτή την εργασία. Η πρώτη δρομολόγηση βελτιστοποιεί το αποτελεσματικό εύρος ζώνης χρησιμοποιώντας ταχύτερους συνδέσμους όποτε είναι δυνατόν, αλλά οδηγεί σε μη ισορροπημένη χρήση των διασυνδέσεων, ροές που μένουν αδρανείς, μπλοκάρονται από εξαρτήσεις μεταφοράς και διαφορετικούς χρόνους εκκίνησης της *compute* λειτουργίας σε κάθε GPU. Αντίθετα, η προσέγγισή μας εξισορροπεί τη χρήση των διασυνδέσεων, μειώνει τις αδρανείς ροές μέσω της εσωτερικής σωλήνωσης των μεταφορών, και οδηγεί σε ταυτόχρονη έναρξη της *compute* λειτουργίας σε όλες τις GPU.

Η αρχική λειτουργία GEMM αποσυντίθεται σε 1) μια λειτουργία GEMM χωρίς τον πίνακα εισόδου C_{in} (ισοδύναμη με μια GEMM με $b = 0$) και 2) μια ελαφριά προσθετική λειτουργία που συγκεντρώνει το αποτέλεσμα της πρώτης στο πίνακα εξόδου C_{out} ακριβώς πριν την εγγραφή (ισοδύναμη με μια λειτουργία AXPY με $alpha = b$). Με αυτόν τον τρόπο, αποσυνδέουμε το

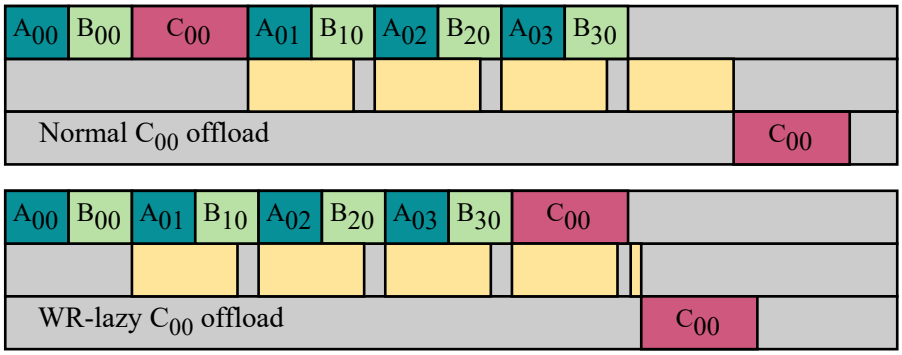


Figure 7.12: Εξαρτήσεις προγραμματισμού και υπολογισμός τεσσάρων υποπροβλημάτων στο C_{00}, με κανονική εκφόρτωση (πάνω) ή χρησιμοποιώντας την προσέγγιση WR-lazy fetch (κάτω). Η καθυστερημένη φόρτωση WR μειώνει τον χρόνο αδράνειας της GPU, απομακρύνοντας το C_{00} από τις εξαρτήσεις εισόδου του πρώτου υποπροβλήματος με κόστος μια ελαφριά επιπλέον υπολογιστική εργασία πριν την εγγραφή του αποτελέσματος.

τμήμα του πυρήνα GEMM που απαιτεί πολλούς υπολογισμούς ($a \cdot A \times B$) από την εξάρτηση εισόδου C_{in}. Η Εικόνα 7.12 δείχνει ένα παράδειγμα του πώς αυτό αλλάζει την εκφόρτωση GEMM για το C_{00}. Οι εξαρτήσεις αυτού του πρώτου υποπροβλήματος (A_{00}, B_{00}, C_{00}) αλλάζουν σε (A_{00}, B_{00}), με αποτέλεσμα λιγότερο χρόνο αδράνειας GPU, καθώς φορτώνουμε καθυστερημένα (C_{00}) στο τέλος της υπολογιστικής διαδικασίας GEMM και το ενημερώνουμε με μια λειτουργία AXPY πριν την εγγραφή του. Αυτή η βελτιστοποίηση είναι ωφέλιμη για προβλήματα όπου ο πίνακας C μοιράζεται αρχικά τη θέση του είτε με τον A είτε με τον B. Σε όλες τις άλλες περιπτώσεις, δεν βελτιώνει την απόδοση, καθώς τα tiles εισόδου A_T, B_T, C_T χρησιμοποιούν διαφορετικά streams όταν φορτώνονται, και οι μεταφορές τους επομένως επικαλύπτονται. Επιπλέον, για να επιτραπεί αυτή η βελτιστοποίηση, απαιτείται πρόσθετη μνήμη buffer μεγέθους $sizeof(C)/\ gpu\_num$ ανά GPU, καθώς τόσο το C'_T όσο και το $C_{T,in}$ πρέπει να αποθηκευτούν πριν τον υπολογισμό και την εγγραφή του $C_{T,out}$. Επομένως, εφαρμόζουμε αυτή τη βελτιστοποίηση επιλεκτικά μόνο σε περιπτώσεις όπου $A_{loc} == C_{loc}$ ή $B_{loc} == C_{loc}$.

7.3.1.7 Στατική δρομολόγηση

Τέλος, παρέχουμε μια ολοκληρωμένη υλοποίηση GEMM που συνδυάζει τις περιγραφόμενες βελτιστοποιήσεις σε έναν αλγόριθμο όπως φαίνεται στον Αλγόριθμο 2. Το πρώτο μέρος του αλγορίθμου (γραμμές 1-17) εκτελείται κάθε φορά που καλείται μια ρουτίνα GEMM με ένα νέο σύνολο παραμέτρων ($params$) (είσοδος, μέγεθος προβλήματος, θέσεις πινάκων), υπολογίζοντας μια βελτιστοποιημένη runtime task queue RTQ για το πρόβλημα αυτό. Αρχικά, το GEMM αποσυντίθεται σε υποπροβλήματα (στη γραμμή 2), τα οποία με τη σειρά τους ανατίθενται στις συσκευές και προσαρμόζονται για τον αλγόριθμο WR-lazy εάν αυτό είναι ωφέλιμο για

Algorithm 2: Ο αλγόριθμος στατικής δρομολόγησης

```
   Data: GEMM params (A, B, C, M, N, K, A_loc, B_loc, C_loc)
 1 if (νέο params) then
 2     SP[num_sp] ← decompose2D(T = T_min, gpu_num, params)
 3     if (A_loc == C_loc Ή B_loc == C_loc) then
 4         SP.adjust_SPs_WRLAZY()
 5     SoftBuf ←assertMemRequirements(SPs)
 6     LAM = {0}, sched_sp = 0
 7     Runtime task queue RTQ = []
 8     while (sched_sp < num_sp) do
 9         for (gpu_i in gpu_num) do
10             currSP = select_SP(gpu_i, SP, LAM)
11             tasklist = split_to_tasks(currSP)
12             for (task in tasklist) do
13                 if (task είναι fetch) then
14                     task.optimize_routing(LAM)
15                     LAM.update_load(task)
16                 RTQ.append(task)
17             sched_sp ← sched_sp + 1
18 for (task in RTQ) do task.fire() ;
19 sync_GPUs()
```

τις παραμέτρους του προβλήματος ($params$) (στη γραμμή 3). Στη συνέχεια, εκτελείται ένα επαναλαμβανόμενο μέρος (στις γραμμές 8-17), επιλέγοντας υποπροβλήματα στις συσκευές με σειρά round-robin μέχρι να προγραμματιστούν όλα τα υποπροβλήματα. Η σειρά των υποπροβλημάτων ανά GPU καθορίζεται από μια συνάρτηση κόστους $select\_SP$ (στη γραμμή 10) που επιστρέφει το *βέλτιστο* υποπρόβλημα με βάση την τρέχουσα κατάσταση του προγράμματος. Αφού επιλεγεί ένα υποπρόβλημα, αποσπάται σε εργασίες (στη γραμμή 11) όπως περιγράφεται στην Ενότητα 7.3.1.2. Οι εργασίες φόρτωσης βελτιστοποιούνται (στις γραμμές 14-15) όπως περιγράφεται στις Ενότητες 7.3.1.3,7.3.1.4 και 7.3.1.5 και κάθε εργασία τοποθετείται στη RTQ (γραμμή 16). Αφού ολοκληρωθεί αυτό το μέρος, η RTQ για αυτό το σύνολο παραμέτρων αποθηκεύεται εσωτερικά και επαναχρησιμοποιείται για όλα τα επόμενα προβλήματα που χρησιμοποιούν τις ίδιες παραμέτρους κατά τη διάρκεια της ζωής του προγράμματος. Το δεύτερο μέρος του αλγορίθμου (γραμμές 18-19) απλά διατρέχει την RTQ και εκτελεί όλες τις εργασίες στις αντίστοιχες ροές και GPU τους.

Βελτιστοποίηση της σειράς δρομολόγησης υποπροβλημάτων: Είναι γενικά αποδεκτό ότι η σειρά με την οποία προγραμματίζονται τα υποπροβλήματα στις GPU είναι σημαντική διότι επηρεάζει 1) τη δρομολόγηση της επικοινωνίας και 2) τον χρόνο αδράνειας της GPU [9,57,161]. Μια κοινή

Table 7.4: Χαρακτηριστικά του συστήματος NVIDIA HGX.

Karolina GPU	CPU	GPU
Υπολογιστής:	2 x AMD Zen 3, 7763 CPU 128 cores @ 2.45 GHz	8 X NVIDIA A100 FP peak 17.2* TFlop/s DP peak 17.2* TFlop/s
Μνήμη:	1TB DDR4	40 GB HBM2 1.56 TB/s
Διασύνδεση:	PCIe Gen4 x16	NVLink3 / NVSwitch2
Συμπιεστής:	g++ 11.2.0	CUDA 12.2
Σημαίες Opt.	-O3	-O3, -arch=sm_80

τεχνική για τη βελτιστοποίηση της σειράς των υποπροβλημάτων είναι η χρήση προτεραιότητας των υποπροβλημάτων με βάση την ελαχιστοποίηση των λειτουργιών φόρτωσης [57]. Χρησιμοποιούμε μια παρόμοια τεχνική για το $select\_SP$, αλλά αντί να προτιμούμε τις ελάχιστες λειτουργίες φόρτωσης, χρησιμοποιούμε την *εκτίμηση ETA* για αυτές τις φορτώσεις, αξιοποιώντας το LAM, σε συνδυασμό με τις εξαρτήσεις tile κάθε υποπροβλήματος, όπως περιγράφεται στην 7.3.1.4. Αυτή η μέθοδος είναι ευαίσθητη στην φόρτωση και έχει ως αποτέλεσμα τον ελάχιστο χρόνο αδράνειας για τις *εργασίες υπολογισμού*, καθώς προτεραιοποιεί τις εργασίες των οποίων οι εξαρτήσεις φόρτωσης αναμένονται να ικανοποιηθούν νωρίτερα.

7.3.2 Αξιολόγηση

Για την αξιολόγηση της απόδοσης, χρησιμοποιούμε ένα σύστημα NVIDIA HGX, το οποίο αποτελεί μέρος των κόμβων του HPC Karolina [85], και περιγράφεται στον Πίνακα 7.4. Κάθε κόμβος αποτελείται από 8 NVIDIA A100 GPUs συνδεδεμένες με ένα προηγμένο δίκτυο διασύνδεσης NVLink3.0 και NVSwitch2.0, που επιτρέπει ταυτόχρονη αμφίδρομη επικοινωνία μεταξύ όλων των GPUs με συνολικό έυρος ζώνης 4.8 TB/s (600 Gb/s αμφίδρομα ανά GPU). Οι GPUs συνδέονται με τη μνήμη του host μέσω PCiE με μέση χωρητικότητα 96 Gb/s (12 GB/s ανά GPU) για όλες τις επικοινωνίες CPU-GPU. Η συχνότητα ρολογιού των A100 GPUs έχει ρυθμιστεί για υψηλότερη ενεργειακή απόδοση, με αποτέλεσμα 12% λιγότερη κορυφαία απόδοση (17.2 έναντι 19.5 TFlops ανά GPU A100).

Για την αξιολόγηση της απόδοσης, χρησιμοποιούμε ένα *κανονικό* σύνολο δεδομένων με τετράγωνα προβλήματα, όπως αναφέρεται και σε σχετική εργασία [57, 68, 157], και ένα *μεικτό* σύνολο δεδομένων που επεκτείνει το κανονικό με μικτές αρχικές τοποθεσίες για τους πίνακες και επιπλέον προβλήματα fat-thin και thin-fat που αποκλίνουν από την συνήθη αναλογία επικοινωνίας/υπολογισμού GEMM [9]. Για το *κανονικό* σύνολο δεδομένων, επιλέγουμε 12 μεγέθη προβλημάτων ($M_{sq} = N_{sq} = K_{sq} = (5120 \xrightarrow{\text{step=1024}} 16384)$) που είναι περιορισμένα από την επικοινωνία στο σύστημα δοκιμών μας, με βάση την ένταση λειτουργίας τους, και 7 *μεγάλα*

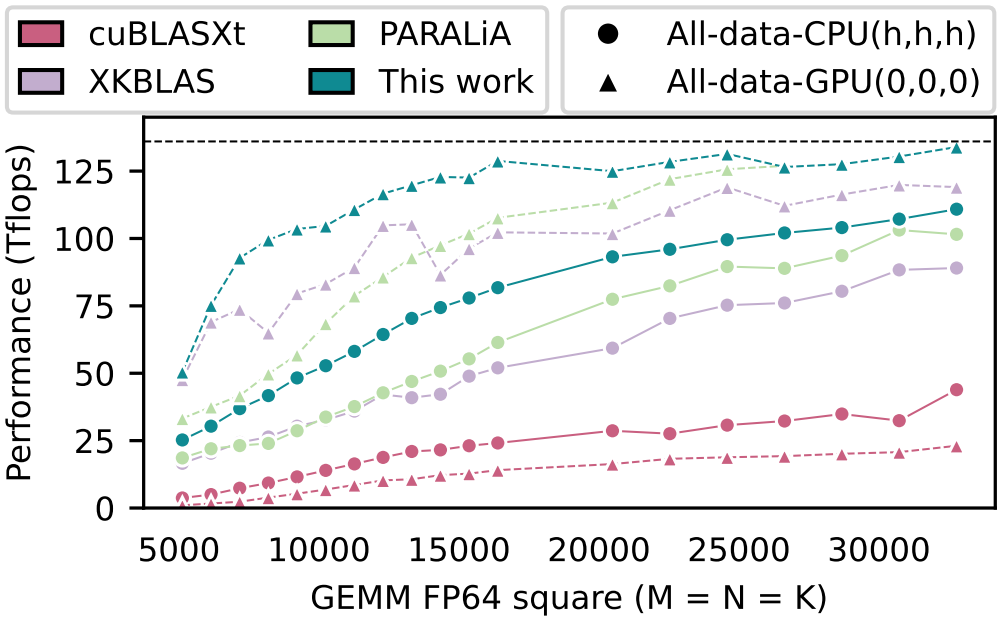


Figure 7.13: Η απόδοση GEMM (M=N=K) FP64 για το κανονικό σύνολο δεδομένων για 8 GPUs στο σύστημα δοκιμών NVIDIA HGX μας (συνολική απόδοση συστήματος = διακεκομμένη γραμμή). Η προσέγγισή μας προσφέρει ισχυρή απόδοση ανεξαρτήτως της τοποθέτησης των δεδομένων, αποφεύγει την ανισορροπία και ξεπερνά όλες τις προηγούμενες προσεγγίσεις, αποδεικνύοντας μεγαλύτερη αποτελεσματικότητα σε μεγέθη προβλημάτων περιορισμένα από την επικοινωνία (12 αριστερότερα σημεία).

μεγέθη προβλημάτων ($M_{sq} = N_{sq} = K_{sq} = (20480 \xrightarrow{\text{step=2048}} 32768)$) που αναμένονται να είναι περιορισμένα από τους υπολογισμούς.

Εκτελούμε τα επιλεγμένα μεγέθη προβλημάτων με δύο διαμορφώσεις. Στην πρώτη διαμόρφωση, όλοι οι πίνακες αρχικά βρίσκονται στη μνήμη CPU (h, h, h), συνεπώς αναμένουμε ότι το κύριο εμπόδιο θα είναι η χωρητικότητα του PCIe. Στη δεύτερη διαμόρφωση, όλοι οι πίνακες αρχικά βρίσκονται στη μνήμη του gpu_0 $(0, 0, 0)$, συνεπώς οι μεταφορές μπορούν να χρησιμοποιούν απευθείας το NVLink. Το *μεικτό* σύνολο δεδομένων περιγράφεται στην Ενότητα 7.3.2.2.

7.3.2.1 Σύγκριση με την τεχνολογία αιχμής

Συγκρίνουμε την υλοποίησή μας με τις βιβλιοθήκες multi-GPU τεχνολογίας αιχμής που επιτυγχάνουν την υψηλότερη απόδοση GEMM, εκτελώντας πειράματα για το κανονικό σύνολο δεδομένων χρησιμοποιώντας ολόκληρο τον κόμβο (8 GPUs) του Πίνακα 7.4. Ειδικότερα, αξιολογούμε τα XKBLAS [57] και PARALiA [9] και αποκλείουμε προηγούμενες προσεγγίσεις που υπερβαίνουν [68, 157]. Αξιολογούμε επίσης το cuBLASXt [124], ως τη βιβλιοθήκη που προτείνεται απο την NVIDIA, παρά την κατώτερη απόδοσή του [9, 57, 157]. Αξιολογούμε την απόδοση GEMM FP64

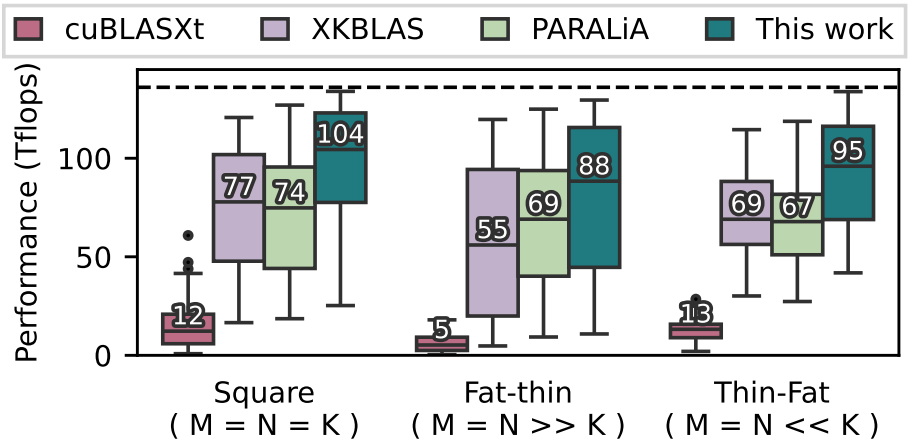


Figure 7.14: Σύγκριση αντοχής απόδοσης GEMM FP64 με την τεχνολογία αιχμής, χρησιμοποιώντας το *μεικτό* σύνολο δεδομένων, χωρισμένο σε τρεις ομάδες, σύμφωνα με τις μορφές πινάκων. Η προσέγγισή μας ξεπερνά όλες τις υπάρχουσες βιβλιοθήκες, ανεξαρτήτως της φύσης του προβλήματος και της τοποθέτησης δεδομένων, παρέχοντας μια ομοιόμορφα ανώτερη λύση για multi-GPU GEMM.

(διπλής ακρίβειας). Η Εικόνα 7.13 δείχνει την απόδοση της παρούσας εργασίας σε σύγκριση με την τεχνολογία αιχμής για GEMM FP64 χρησιμοποιώντας 8 GPUs για το *κανονικό* σύνολο δεδομένων. Στην περίπτωση όπου όλοι οι πίνακες αρχικά βρίσκονται στη μνήμη του host, η οποία περιορίζεται από τη χωρητικότητα PCIe, η εργασία μας προσφέρει υψηλή απόδοση για μικρότερα μεγέθη προβλημάτων. Κατά μέσο όρο, η εργασία μας υπερβαίνει τα cuBLASXt, XKBLAS και PARALiA κατά 3.42×, 1.4× και 1.31×, αντίστοιχα. Στην περίπτωση όπου τα δεδομένα βρίσκονται αρχικά σε μια GPU και αποφεύγονται οι μεταφορές PCIe, το PARALiA έχει υψηλά έξοδα για μικρότερα μεγέθη προβλημάτων, και το XKBLAS προσφέρει λιγότερη και μη αξιόπιστη απόδοση λόγω ανισορροπίας φορτίου. Η υλοποίησή μας αντιμετωπίζει αποτελεσματικά και τους δύο τύπους εξόδων, ξεπερνώντας τα cuBLASXt, XKBLAS και PARALiA κατά 10.3x, 1.23x και 1.26x, αντίστοιχα, κατά μέσο όρο.

7.3.2.2 Αντοχή επίδοσης σε μεικτά προβλήματα

Τέλος, για να επιβεβαιώσουμε την αντοχή της προσέγγισής μας σε διάφορα χαρακτηριστικά προβλημάτων GEMM, επεκτείνουμε το *κανονικό* σύνολο δεδομένων με τρεις επιπλέον αρχικές διαμορφώσεις τοποθέτησης πινάκων: $(4, 2, h), (h, h, 0), (4, 2, 7)$, και δύο μη-τετράγωνες μορφές προβλημάτων (fat-thin και thin-fat). Για τα προβλήματα fat-thin, χρησιμοποιούμε ($M_{fat} = N_{fat} = (16384 \xrightarrow{\text{step=4096}} 40960), K_{thin} = \frac{M_{fat}}{r}, r \in [4, 8, 16]$) και για thin-fat ($M_{thin} = N_{thin} = (5120 \xrightarrow{\text{step=1024}} 11264), K_{fat} = M_{thin} \times r, r \in [4, 8, 16]$). Αυτό έχει ως αποτέλεσμα ένα *μεικτό* σύνολο δεδομένων με 305 σημεία δεδομένων.

Η Εικόνα 7.14 δείχνει την απόδοση GEMM FP64 των cuBLASXt, XKBLAS, PARALiA και της δικής μας δουλειάς στο *μεικτό* σύνολο δεδομένων, κατηγοριοποιημένο με βάση τη μορφή του προβλήματος (τετράγωνο, fat-thin, και thin-fat). Το cuBLASXt είναι η πιο αργή υλοποίηση για όλες τις μορφές προβλημάτων, και η απόδοσή του μειώνεται περαιτέρω στο μεικτό σύνολο δεδομένων λόγω των διαφορών στις αρχικές τοποθετήσεις πινάκων. Το XKBLAS από την άλλη αποδίδει καλά σε τετράγωνα και thin-fat προβλήματα για τις διάφορες τοποθετήσεις, αλλά για πίνακες fat-thin το πολύ μεγαλύτερο C matrix προκαλεί καθυστερήσεις στην επικοινωνία WR. Το PARALiA προσφέρει καλύτερη απόδοση για όλες τις μορφές, σύμφωνα με την επιδιωκόμενη αντοχή στην απόδοση, αλλά αποδίδει χειρότερα από το XKBLAS σε τετράγωνα και thin-fat προβλήματα. Τέλος, η δική μας δουλειά επιτυγχάνει καλύτερη απόδοση για όλα τα προβλήματα στο μεικτό σύνολο δεδομένων, ξεπερνώντας κατά μέσο όρο το cuBLASXt, XKBLAS και PARALiA κατά 11.8x, 1.45x και 1.37x (8.7x, 1.35x, 1.4x για τετράγωνα προβλήματα, 17.6x, 1.6x, 1.28x για προβλήματα fat-thin και 7.31x, 1.38x, 1.42x για προβλήματα thin-fat), αντίστοιχα.

7.4 Συμπεράσματα

Στην παρούσα διατριβή, στοχεύσαμε στην επίτευξη φορητότητας, βέλτιστης απόδοσης και αποδοτικής χρήσης πόρων για την εκτέλεση της BLAS σε GPU. Η εκτέλεση BLAS σε GPU εισάγει μια ποικιλία από *εσωτερικές παραμέτρους* που μπορούν να ρυθμιστούν για κάθε κλήση BLAS, και απαιτεί σημαντικές *αποφάσεις* για την αποσύνθεση, επικοινωνία και προγραμματισμό κατά τη διάρκεια εκτέλεσης που επηρεάζουν την απόδοση της εφαρμογής. Εξαιτίας της απαγορευτικής πολυπλοκότητας του προβλήματος, προτείνουμε μια προσέγγιση βασισμένη σε μοντέλα, όπου χρησιμοποιούμε μοντελοποίηση για την εκτίμηση των χαρακτηριστικών απόδοσης, επικοινωνίας και επικαλύψεως για κάθε υποψήφιο σύστημα. Στη συνέχεια, αυτά τα μοντέλα χρησιμοποιούνται για την αυτοματοποιημένη ρύθμιση παραμέτρων και λογισμικού, καθώς και για την επιλογή των πόρων που θα χρησιμοποιηθούν με βάση τα χαρακτηριστικά του προβλήματος και του συστήματος κάθε κλήσης ρουτίνας κατά τη διάρκεια εκτέλεσης. Η εργασία μας αποτελείται από δύο μέρη: 1) εισάγουμε διάφορα μοντέλα απόδοσης για την εκφόρτωση GPU BLAS και παρέχουμε μια βιβλιοθήκη υψηλής απόδοσης για BLAS σε πολλαπλές GPU βασισμένη σε γνώσεις και αυτοματοποιημένη ρύθμιση που προέρχονται από αυτά τα μοντέλα και 2) προτείνουμε μια υλοποίηση GEMM που συνδυάζει γνώσεις βασισμένες σε μοντέλα με τεχνικές κατανεμημένου προγραμματισμού για την βελτιστοποίηση της επικοινωνίας και την επίτευξη καλύτερης απόδοσης.

Στο πρώτο μέρος, παρουσιάζουμε το PARALiA [9], Μία βιβλιοθήκη για την εκτέλεση BLAS σε πολλαπλές GPU. Παρόμοια με τις υπάρχουσες προσεγγίσεις BLAS σε πολλαπλές GPU, το PARALiA χρησιμοποιεί διαχωρισμό προβλήματος, προγραμματισμό υποπροβλημάτων και επικαλυψη επικοινωνίας-υπολογισμών για την μεγιστοποίηση της απόδοσης των ρουτινών

BLAS σε ρυθμίσεις πολλαπλών GPU. Αντίθετα με τις υπάρχουσες προσεγγίσεις, το PARALiA δίνει έμφαση στην βελτιστοποίηση της επικοινωνίας και της χρήσης πόρων μέσω αυτοματοποιημένης ρύθμισης βασισμένης σε μοντέλα. Για αυτόν τον σκοπό, εισάγουμε μια γενίκευση υλικού που ονομάζεται `LinkMap`, η οποία εκτιμά με ακρίβεια τα χαρακτηριστικά διασύνδεσης και επικοινωνίας για κάθε πρόβλημα BLAS. Το PARALiA χρησιμοποιεί αυτά τα μοντέλα για να βελτιστοποιήσει την επικοινωνία και να επιλέξει συσκευές εκτέλεσης (GPUs), με βάση έναν προεπιλεγμένο στόχο βελτιστοποίησης, ο οποίος μπορεί να είναι η επίδοση ή κάποια μετρική απόδοσης, αποφεύγοντας την σπατάλη πόρων. Αξιολογούμε την προσέγγισή μας σε ενα cluster πολλαπλών GPU που διαθέτει ετερογενείς συνδέσεις μεταξύ των συσκευών. Τα πειράματά μας επικεντρώνονται στην απόδοση του GEMM με διπλή ακρίβεια, ως εκπρόσωπος των level-3 BLAS. Η αξιολόγησή μας δείχνει ότι η προσέγγισή μας υπερτερεί των προυπάρχων βιβλιοθηκών BLASX και XKBLAS με μια μέση γεωμετρική βελτίωση της τάξης του 1.7x και 2.4x αντίστοιχα, με σημαντική βελτίωση επίδοσης για εκτελέσεις ρουτινών όπου τα δεδομένα αρχικά βρίσκονται σε διάφορες GPU. Επιπλέον, δείχνουμε πώς, με την επιλογή συσκευών και θέτοντας διάφορους στόχους βελτιστοποίησης, η προσέγγισή μας είναι ικανή να επιτύχει υψηλή επίδοση συνδυασμένη με καλύτερη ενεργειακή αποδοτικότητα, με μια μέση γεωμετρική βελτίωση 2.5x σε σύγκριση με τη BLASX και 3.4x σε σύγκριση με τη XKBLAS.

Στο δεύτερο μέρος, παρέχουμε μια βελτιστοποιημένη υλοποίηση GEMM προσαρμοσμένη για αποτελεσματική εκτέλεση σε κόμβους με πολλαπλές GPU [8]. Η προσέγγισή μας επικεντρώνεται στη βελτίωση της επικοινωνίας, της δρομολόγησης και της εξισορρόπησης φόρτου του GEMM σε πολλαπλές GPU, χρησιμοποιώντας διάφορες τεχνικές βελτιστοποίησης. Η υλοποίησή μας βασίζεται σε ένα στατικό πρόγραμμα δρομολόγησης, το οποίο κατασκευάζεται πριν από την εκτέλεση όποτε καλείται μια ρουτίνα, και μπορεί έτσι να χρησιμοποιήσει τα χαρακτηριστικά του συγκεκριμένου προβλήματος για να ελαχιστοποιήσει την επικοινωνία, να αυξήσει την απόδοση, να μεγιστοποιήσει την επικαλυψη και να ισορροπήσει την επικοινωνία και τον υπολογισμό. Χρησιμοποιούμε ιεραρχικό διαχωρισμό προβλήματος και προσφέρουμε μια ευρετική τεχνική για την επιλογή μεγέθους tile. Χρησιμοποιούμε πολλαπλά streams για να επικαλύψουμε αποτελεσματικά τον υπολογισμό και την επικοινωνία και αποθηκεύουμε tile για επαναχρησιμοποίηση, αποφεύγοντας την επικοινωνία όπου είναι δυνατόν. Βελτιστοποιούμε την δρομολόγηση επικοινωνίας εξετάζοντας τη διαθεσιμότητα συνδέσμων σημείου-σε-σημείο και προγραμματίζοντας τις μεταφορές tile αναλόγως για να διασφαλίσουμε ότι φτάνουν στις προοριζόμενες GPU το συντομότερο δυνατόν. Επιπλέον, υλοποιούμε ομαδικές μεταφορές για RONLY tiles και καθυστέρηση μεταφορών για τα tile των πινάκων εξόδου (WR). Αξιολογούμε την προσέγγισή μας σε ένα σύστημα NVIDIA HGX, το οποίο διαθέτει 8 NVIDIA A100 GPUs, διασυνδεδεμένες με NVLink3 και NVSwitch2. Τα πειραματά μας δείχνουν την αποτελεσματικότητα των βελτιστοποιήσεών μας στην απόδοση του GEMM. Η υλοποίησή μας υπερβαίνει τις βιβλιοθήκες αιχμής (συμπεριλαμβανομένης της προηγούμενης υλοποίησης PARALiA), κατά 1.29× και 1.37× κατά

μέσο όρο, για GEMM FP32 και FP64 αντίστοιχα. Επιπλέον, η υλοποίησή μας προσφέρει υψηλή απόδοση για μη-τετραγωνικές μορφές πινάκων και μεικτές τοποθετήσεις δεδομένων, υπερβαίνοντας σημαντικά τις υπάρχουσες υλοποιήσεις.

Συμπεραίνουμε ότι, παρά την κοινή αντίληψη ότι οι ρουτίνες BLAS είναι κατάλληλες για GPUs, η υψηλή απόδοση για οποιοδήποτε πρόβλημα σε οποιοδήποτε σύστημα είναι πολύ δύσκολο να επιτευχθεί και απαιτεί την κατανόηση πολλών παραμέτρων και τη λεπτομερή ρύθμιση της εκτέλεσης για κάθε συνδιασμό προβλήματος/συστήματος. Η λύση μας για αυτό είναι η χρήση αυτοματοποιημένης ρύθμισης κατά το χρόνο εκτέλεσης, συνδυάζοντας μοντελοποίηση με μικρο-δοκιμές για την επιλογή τιμών για τις εσωτερικές παραμέτρους των ρουτινών και την βελτιστοποίηση της επικοινωνίας, της επικαλυψης, της δρομολόγησης και της επιλογής πόρων. Στο μέλλον, στοχεύουμε στην επέκταση του PARALiA με πιο εξελιγμένες τεχνικές δρομολόγησης και την αξιοποίηση της γνώσης που ήδη προσφέρουν τα μοντέλα για περαιτέρω βελτιώσεις στην επικοινωνία. Σχεδιάζουμε επίσης την επέκταση σε πολλαπλούς κόμβους με πολλές GPU, συνδυάζοντας την ενδο-κομβική υλοποίησή μας με κατανεμημένες τεχνικές που στοχεύουν την κλιμάκωση. Τέλος, εργαζόμαστε για την υποστήριξη άλλων μορφών δεδομένων εισόδου, όπως το PBLAS, το οποίο χρησιμοποιείται συνήθως σε συστήματα πολλαπλών κόμβων.

Bibliography

[1] Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. A three-dimensional approach to parallel matrix multiplication. IBM Journal of Research and Development 39, 5 (1995), 575–582.

[2] Agarwal, R. C., Gustavson, F. G., and Zubair, M. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. IBM Journal of Research and Development 38, 6 (1994), 673–681.

[3] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. A hybridization methodology for high-performance linear algebra software for gpus. In GPU Computing Gems Jade Edition, W. mei W. Hwu, Ed., Applications of GPU Computing Series. Morgan Kaufmann, Boston, 2012, pp. 473–484.

[4] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. P. Achieving high performance on supercomputers with a sequential task-based programming model. IEEE Transactions on Parallel and Distributed Systems (2017), 1–1.

[5] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. Numerical linear algebra on emerging architectures: The plasma and magma projects. Journal of Physics: Conference Series 180, 1 (jul 2009), 012037.

[6] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. Loggp: incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In Proceedings of the Seventh Annual ACM Symposium on Parallel

Algorithms and Architectures (New York, NY, USA, 1995), SPAA '95, Association for Computing Machinery, p. 95–105.

[7] Anastasiadis, P., Papadopoulou, N., Goumas, G., and Koziris, N. Cocopelia: Communication-computation overlap prediction for efficient linear algebra on gpus. In 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2021), pp. 36–47.

[8] Anastasiadis, P., Papadopoulou, N., Goumas, G., and Koziris, N. Uncut-gemms : Communication-aware matrix multiplication on multi-gpu nodes. In To be published at: 2024 IEEE International Conference on Cluster Computing (CLUSTER) (2024), pp. –.

[9] Anastasiadis, P., Papadopoulou, N., Goumas, G., Koziris, N., Hoppe, D., and Zhong, L. Paralia: A performance aware runtime for auto-tuning linear algebra on heterogeneous systems. ACM Trans. Archit. Code Optim. 20, 4 (dec 2023).

[10] Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. Data-aware task scheduling on multi-accelerator based platforms. In 2010 IEEE 16th International Conference on Parallel and Distributed Systems (2010), pp. 291–298.

[11] Augonnet, C., Thibault, S., and Namyst, R. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA, Mar. 2010.

[12] Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S. An extension of the starss programming model for platforms with multiple gpus. In Euro-Par 2009 Parallel Processing (Berlin, Heidelberg, 2009), H. Sips, D. Epema, and H.-X. Lin, Eds., Springer Berlin Heidelberg, pp. 851–862.

[13] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W. An adaptive performance modeling tool for gpu architectures. SIGPLAN Not. 45, 5 (Jan. 2010), 105–114.

[14] Bell, C., Bonachea, D., Nishtala, R., and Yelick, K. Optimizing bandwidth limited problems using one-sided communication and overlap. In Proceedings 20th IEEE International Parallel and Distributed Processing Symposium (2006), pp. 10 pp.–.

[15] Belter, G., Jessup, E. R., Karlin, I., and Siek, J. G. Automating the generation of composed linear algebra kernels. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (New York, NY, USA, 2009), SC '09, Association for Computing Machinery.

[16] Bernabé, G., Cuenca, J., García, L.-P., and Giménez, D. Tuning basic linear algebra routines for hybrid cpu+gpu platforms. Procedia Computer Science 29 (2014), 30 – 39. 2014 International Conference on Computational Science.

[17] Berntsen, J. Communication efficient matrix multiplication on hypercubes. Parallel Computing 12, 3 (1989), 335–342.

[18] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemariner, P., Ltaeif, H., Luszczek, P., YarKhan, A., and Dongarra, J. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In - (Anchorage, Alaska, USA, 2011-05 2011), IEEE, pp. 1432–1441.

[19] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. J. Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science and Engineering 15, 6 (2013), 36–45.

[20] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. DAGuE: A generic distributed DAG engine for High Performance Computing. 37–51.

[21] Boyer, M., Meng, J., and Kumaran, K. Improving gpu performance prediction with data transfer modeling. In 2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (2013), pp. 1097–1106.

[22] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35, 1 (2009), 38–53.

[23] Cannon, L. E. A cellular computer to implement the kalman filter algorithm. PhD thesis, -, USA, 1969. AAI7010025.

[24] Chan, E., Quintana-Orti, E. S., Quintana-Orti, G., and van de Geijn, R. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (New York, NY, USA, 2007), SPAA '07, Association for Computing Machinery, p. 116–125.

[25] Chen, W., Zhai, J., Zhang, J., and Zheng, W. Loggpo: An accurate communication model for performance prediction of mpi programs. Science in China Series F: Information Sciences 52 (10 2009), 1785–1791.

[26] Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. Scalapack: A portable linear algebra library for distributed memory computers — design issues and performance. In Applied Parallel

Computing Computations in Physics, Chemistry and Engineering Science (Berlin, Heidelberg, 1996), J. Dongarra, K. Madsen, and J. Waśniewski, Eds., Springer Berlin Heidelberg, pp. 95–106.

[27] Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., and Whaley, R. C. A proposal for a set of parallel basic linear algebra subprograms. In Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science (Berlin, Heidelberg, 1996), J. Dongarra, K. Madsen, and J. Waśniewski, Eds., Springer Berlin Heidelberg, pp. 107–114.

[28] Choi, J., Walker, D. W., and Dongarra, J. J. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Concurrency: Practice and Experience 6, 7 (1994), 543–570.

[29] Clint Whaley, R., Petitet, A., and Dongarra, J. J. Automated empirical optimizations of software and the atlas project. Parallel Computing 27, 1 (2001), 3–35. New Trends in High Performance Computing.

[30] Corporation, A. Amd core math library (acml). http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/. Accessed: 2024-08-11.

[31] Corporation, A. Amd optimizing cpu libraries (aocl). https://developer.amd.com/amd-aocl/. Accessed: 2024-08-11.

[32] Corporation, I. Ibm engineering and scientific subroutine library (essl). https://www.ibm.com/docs/en/essl/6.3.0. Accessed: 2024-08-11.

[33] Corporation, I. Intel math kernel library (intel mkl). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Accessed: 2024-08-11.

[34] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. Logp: towards a realistic model of parallel computation. SIGPLAN Not. 28, 7 (jul 1993), 1–12.

[35] Cámara, J., Cuenca, J., García, L., and Giménez, D. Empirical modelling of linear algebra shared-memory routines. Procedia Computer Science 18 (12 2013), 110–119.

[36] Danalis, A., Kim, K.-Y., Pollock, L., and Swany, M. Transformations to parallel codes for communication-computation overlap. In SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (2005), pp. 58–58.

[37] Dekel, E., Nassimi, D., and Sahni, S. Parallel matrix and graph algorithms. SIAM Journal on Computing 10, 4 (1981), 657–675.

[38] Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., and Spillinger, O. Communication-optimal parallel recursive rectangular matrix multiplication. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (2013), pp. 261–272.

[39] Dongarra, J. Basic linear algebra subprograms technical (blast) forum standard ii. IJHPCA 16 (05 2002), 1–111.

[40] Dongarra, J. An updated set of basic linear algebra subprograms (blas). ACM Trans. Math. Softw. 28, 2 (jun 2002), 135–151.

[41] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., and Yamazaki, I. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. Springer International Publishing, Cham, 2014, pp. 3–28.

[42] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., and Yamazaki, I. Accelerating numerical dense linear algebra calculations with gpus. Numerical Computations with GPUs (2014), 1–26.

[43] Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., Yarkhan, A., Abalenkovs, M., Bagherpour, N., Hammarling, S., Šístek, J., Stevens, D., Zounon, M., and Relton, S. D. Plasma: Parallel linear algebra software for multicore using openmp. ACM Trans. Math. Softw. 45, 2 (may 2019).

[44] Dongarra, J. J., Cruz, J. D., Hammarling, S., and Duff, I. S. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. ACM Trans. Math. Softw. 16, 1 (mar 1990), 18–28.

[45] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16, 1 (mar 1990), 1–17.

[46] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. ACM Trans. Math. Softw. 14, 1 (mar 1988), 18–32.

[47] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. An extended set of fortran basic linear algebra subprograms. ACM Trans. Math. Softw. 14, 1 (mar 1988), 1–17.

[48] Dongarra, J. J., Gustavson, F. G., and Karp, A. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. SIAM Review 26, 1 (1984), 91–112.

[49] Dongarra, J. J., Moler, C. B., Bunch, J. R., and Stewart, G. W. Linpack users' guide. In ' ' (1987).

[50] Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (New York, NY, USA, 2004), HWWS '04, Association for Computing Machinery, p. 133–137.

[51] Fox, G., Otto, S., and Hey, A. Matrix algorithms on a hypercube i: Matrix multiplication. Parallel Computing 4, 1 (1987), 17–31.

[52] Frank, M. I., Agarwal, A., and Vernon, M. K. Lopc: modeling contention in parallel algorithms. In Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA, 1997), PPOPP '97, Association for Computing Machinery, p. 276–287.

[53] Galoppo, N., Govindaraju, N., Henson, M., and Manocha, D. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (2005), pp. 3–3.

[54] Gates, M., Kurzak, J., Charara, A., YarKhan, A., and Dongarra, J. Slate: design of a modern distributed and accelerated linear algebra library. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2019), SC '19, Association for Computing Machinery.

[55] Gates, M., YarKhan, A., Sukkari, D., Akbudak, K., Cayrols, S., Bielich, D., Abdelfattah, A., Farhan, M. A., and Dongarra, J. Portable and efficient dense linear algebra in the beginning of the exascale era. In 2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (2022), pp. 36–46.

[56] Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (2013), pp. 1299–1308.

[57] Gautier, T., and Lima, J. V. F. 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) (2020), pp. 1–8.

[58] Gautier, T., and Lima, J. V. F. Evaluation of two topology-aware heuristics on level- 3 blas library for multi-gpu platforms. In 2021 SC Workshops Supplementary Proceedings (SCWS) (2021), pp. 12–22.

[59] Geoffray, P., and Hoefler, T. Adaptive routing strategies for modern high performance networks. In 2008 16th IEEE Symposium on High Performance Interconnects (2008), pp. 165–172.

[60] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34, 3 (may 2008).

[61] Goto, K., and Van De Geijn, R. High-performance implementation of the level-3 blas. ACM Trans. Math. Softw. 35, 1 (jul 2008).

[62] Goumas, G., Sotiropoulos, A., and Koziris, N. Minimizing completion time for loop tiling with computation and communication overlapping. In Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001 (2001), pp. 10 pp.–.

[63] Gregg, C., and Hazelwood, K. 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 (2011), pp. 134–144.

[64] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. Flame: Formal linear algebra methods environment. ACM Trans. Math. Softw. 27, 4 (dec 2001), 422–455.

[65] Gómez-Luna, J., González-Linares, J. M., Benavides, J. I., and Guil, N. Performance models for asynchronous data transfers on consumer graphics processing units. Journal of Parallel and Distributed Computing 72, 9 (2012), 1117–1126. Accelerators for High-Performance Computing.

[66] Haidar, A., Cao, C., Yarkhan, A., Luszczek, P., Tomov, S., Kabir, K., and Dongarra, J. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 491–500.

[67] Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. Simulation of cloud dynamics on graphics hardware. In ACM SIGGRAPH 2005 Courses (New York, NY, USA, 2005), SIGGRAPH '05, Association for Computing Machinery, p. 223–es.

[68] Herault, T., Robert, Y., Bosilca, G., and Dongarra, J. Generic matrix multiplication for multi-gpu accelerated distributed-memory platforms over parsec. In 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA) (2019), pp. 33–41.

[69] HLRS. Vulcan, hpc cluster.

[70] Hoefler, T., and Belli, R. 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 (New York, NY, USA, 2015), SC '15, Association for Computing Machinery.

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

[72] Hoefler, T., Lichei, A., and Rehm, W. Low-overhead loggp parameter assessment for modern interconnection networks. In 2007 IEEE International Parallel and Distributed Processing Symposium (2007), pp. 1–8.

[73] Hoefler, T., Mehlan, T., Mietke, F., and Rehm, W. Logfp - a model for small messages in infiniband. In Proceedings 20th IEEE International Parallel and Distributed Processing Symposium (2006), pp. 6 pp.–.

[74] Hoefler, T., Schneider, T., and Lumsdaine, A. Accurately measuring collective operations at massive scale. In 2008 IEEE International Symposium on Parallel and Distributed Processing (2008), pp. 1–8.

[75] Hoefler, T., Schneider, T., and Lumsdaine, A. 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.

[76] Hoefler, T., Schneider, T., and Lumsdaine, A. Loggopsim - simulating large-scale applications in the loggops model. vol. 10, pp. 597–604.

[77] Hoefler, T., and Snir, M. Generic topology mapping strategies for large-scale parallel architectures. In Proceedings of the International Conference on Supercomputing (New York, NY, USA, 2011), ICS '11, Association for Computing Machinery, p. 75–84.

[78] Hong, S., and Kim, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. SIGARCH Comput. Archit. News 37, 3 (June 2009), 152–163.

[79] Humphrey, J. R., Price, D. K., Spagnoli, K. E., and Kelmelis, E. J. Accelerating cula linear algebra routines with hybrid gpu and multicore computing. In . (2012).

[80] Humphrey, J. R., Price, D. K., Spagnoli, K. E., Paolini, A. L., and Kelmelis, E. J. Cula: hybrid gpu accelerated linear algebra routines. In Defense + Commercial Sensing (2010).

[81] Iakymchuk, R., and Bientinesi, P. Modeling performance through memory-stalls. ACM SIGMETRICS Performance Evaluation Review 40 (10 2012).

[82] Inc., A. Apple accelerate framework. https://developer.apple.com/documentation/accelerate. Accessed: 2024-08-11.

[83] Ino, F., Fujimoto, N., and Hagihara, K. Loggps: a parallel computational model for synchronization analysis. In Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (New York, NY, USA, 2001), PPoPP '01, Association for Computing Machinery, p. 133–142.

[84] Irony, D., and Tiskin, A. Communication lower bounds for distributed-memory matrix multiplication. Journal of Parallel and Distributed Computing 64 (09 2004), 1017–1026.

[85] IT4I. it4i.cz/en/infrastructure/karolina.

[86] Jain, N., Bhatele, A., Robson, M. P., Gamblin, T., and Kale, L. V. Predicting application performance using supervised learning on communication features. In SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (2013), pp. 1–12.

[87] Johnsson, S. Minimizing the communication time for matrix multiplication on multiprocessors. Parallel Computing 19, 11 (1993), 1235–1257.

[88] Kielmann, T., Bal, H. E., and Verstoep, K. Fast measurement of logp parameters for message passing platforms. In Parallel and Distributed Processing (Berlin, Heidelberg, 2000), J. Rolim, Ed., Springer Berlin Heidelberg, pp. 1176–1183.

[89] Konstantinidis, E., and Cotronis, Y. A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing 107 (2017), 37 – 56.

[90] Kågström, B., Ling, P., and van Loan, C. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. ACM Trans. Math. Softw. 24, 3 (sep 1998), 268–302.

[91] Krüger, J., and Westermann, R. Linear algebra operators for gpu implementation of numerical algorithms. ACM Trans. Graph. 22, 3 (jul 2003), 908–916.

[92] Kurzak, J., Luszczek, P., Yamazaki, I., Robert, Y., and Dongarra, J. Design and implementation of the pulsar programming system for large scale computing. Supercomputing Frontiers and Innovations 4, 1 (Feb. 2017), 4–26.

[93] Kurzak, J., Tomov, S., and Dongarra, J. Autotuning GEMM kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems 23, 11 (November 2012), 2045–2057.

[94] Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R., and Hoefler, T. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2019), pp. 1–22.

[95] Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. An investigation of unified memory access performance in cuda. In 2014 IEEE High Performance Extreme Computing Conference (HPEC) (2014), pp. 1–6.

[96] Lang, J. Data-aware tuning of scientific applications with model-based autotuning. Concurrency and Computation: Practice and Experience 29, 4 (2017), e3885. e3885 cpe.3885.

[97] Larsen, E. S., and McAllister, D. Fast matrix multiplies using graphics hardware. In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (New York, NY, USA, 2001), SC '01, Association for Computing Machinery, p. 55.

[98] Lastovetsky, A., Mkwawa, I.-H., and O'Flynn, M. An accurate communication model of a heterogeneous cluster based on a switch-enabled ethernet network. In 12th International Conference on Parallel and Distributed Systems - (ICPADS'06) (2006), vol. 2, pp. 6 pp.–.

[99] Lastovetsky, A., and Rychkov, V. Accurate and efficient estimation of parameters of heterogeneous communication performance models. The International Journal of High Performance Computing Applications 23, 2 (2009), 123–139.

[100] Lastovetsky, A., Rychkov, V., and O'Flynn, M. Accurate heterogeneous communication models and a software tool for their efficient estimation. The International Journal of High Performance Computing Applications 24, 1 (2010), 34–48.

[101] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. 5, 3 (sep 1979), 308–323.

[102] Li, A., Song, S. L., Chen, J., Liu, X., Tallent, N., and Barker, K. Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In 2018 IEEE International Symposium on Workload Characterization (IISWC) (2018), pp. 191–202.

[103] Li, W., Jin, G., Cui, X., and See, S. An evaluation of unified memory technology on nvidia gpus. In 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2015), pp. 1092–1098.

[104] Li, Y., Dongarra, J., and Tomov, S. A note on auto-tuning GEMM for GPUs. In Proceedings of the 2009 International Conference on Computational Science, ICCS'09 (Baton Roube, LA, May 25-27 2009), Springer.

[105] Limited, A. Arm performance libraries. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux/performance-libraries. Accessed: 2024-08-11.

[106] Liu, B., Qiu, W., Jiang, L., and Gong, Z. Software pipelining for graphic processing unit acceleration. Int. J. High Perform. Comput. Appl. 30, 2 (may 2016), 169–185.

[107] Low, T. M., Igual, F. D., Smith, T. M., and Quintana-Ortí, E. S. Analytical modeling is enough for high-performance BLIS. ACM Transactions on Mathematical Software 43, 2 (Aug. 2016), 12:1–12:18.

[108] Luk, C.-K., Hong, S., and Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2009), pp. 45–55.

[109] Luszczek, P., and Dongarra, J. Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modeling. In - (09 2011), pp. 730–739.

[110] Ma, K., Li, X., Chen, W., Zhang, C., and Wang, X. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In 2012 41st International Conference on Parallel Processing (2012), pp. 48–57.

[111] Martell, M., and Sato, H. Linear performance-breakdown model: A framework for gpu kernel programs performance analysis. International Journal of Networking and Computing 5 (01 2015), 86–104.

[112] Martinasso, M., and Méhaut, J.-F. A contention-aware performance model for hpc-based networks: A case study of the infiniband network. In Euro-Par 2011 Parallel Processing (Berlin, Heidelberg, 2011), E. Jeannot, R. Namyst, and J. Roman, Eds., Springer Berlin Heidelberg, pp. 91–102.

[113] Meng, J., Morozov, V. A., Kumaran, K., Vishwanath, V., and Uram, T. D. Grophecy: Gpu performance projection from cpu code skeletons. In SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011), pp. 1–11.

[114] Meswani, M. R., Carrington, L., Unat, D., Snavely, A., Baden, S., and Poole, S. 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 (2012), pp. 1828–1837.

[115] Meswani, M. R., Carrington, L., Unat, D., Snavely, A., Baden, S., and Poole, S. Modeling and predicting performance of high performance computing applications on hardware accelerators. The International Journal of High Performance Computing Applications 27, 2 (2013), 89–108.

[116] Mishra, A., Li, L., Kong, M., Finkel, H., and Chapman, B. Benchmarking and evaluating unified memory for openmp gpu offloading. In Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (New York, NY, USA, 2017), LLVM-HPC'17, Association for Computing Machinery.

[117] Moritz, C., and Frank, M. Logpg: Modeling network contention in message-passing programs. IEEE Transactions on Parallel and Distributed Systems 12, 4 (2001), 404–415.

[118] Nath, R., Tomov, S., Dong, T. T., and Dongarra, J. Optimizing symmetric dense matrix-vector multiplication on gpus. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2011), SC '11, Association for Computing Machinery.

[119] Nath, R., Tomov, S., and Dongarra, J. Accelerating GPU kernels for dense linear algebra. In Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10 (Berkeley, CA, June 22-25 2010), Springer.

[120] Nath, R., Tomov, S., and Dongarra, J. Accelerating gpu kernels for dense linear algebra. In High Performance Computing for Computational Science – VECPAR 2010 (Berlin, Heidelberg, 2011), J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Springer Berlin Heidelberg, pp. 83–92.

[121] Nath, R., Tomov, S., and Dongarra, J. J. An improved magma gemm for fermi graphics processing units. The International Journal of High Performance Computing Applications 24 (2010), 511 – 515.

[122] NVIDIA. `developer.nvidia.com/cublas`.

[123] NVIDIA. `developer.nvidia.com/cublasmp-downloads`.

[124] NVIDIA. `developer.nvidia.com/cublasxt`.

[125] NVIDIA. docs.nvidia.com/cuda/nvblas.

[126] NVIDIA, Vingelmann, P., and Fitzek, F. H. Cuda, release: 10.2.89, 2020.

[127] O'Neal, K., Brisk, P., Shriver, E., and Kishinevsky, M. Halwpe: Hardware-assisted light weight performance estimation for gpus. In Proceedings of the 54th Annual Design Automation Conference 2017 (New York, NY, USA, 2017), DAC '17, Association for Computing Machinery.

[128] open source. Initial contributors : Wang, Q., Zhang, X., Zhang, Y., and Yi, Q. www.openblas.net.

[129] Papadopoulou, N., Goumas, G., and Koziris, N. A machine-learning approach for communication prediction of large-scale applications. In 2015 IEEE International Conference on Cluster Computing (2015), pp. 120–123.

[130] Pearson, C., Dakkak, A., Hashash, S., Li, C., Chung, I.-H., Xiong, J., and Hwu, W.-M. Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects. In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (New York, NY, USA, 2019), ICPE '19, Association for Computing Machinery, p. 209–218.

[131] Peise, E., and Bientinesi, P. Performance modeling for dense linear algebra. CoRR abs/1209.2364 (2012).

[132] Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. Hierarchical task-based programming with starss. Int. J. High Perform. Comput. Appl. 23, 3 (aug 2009), 284–299.

[133] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., and van de Geijn, R. A. Solving dense linear systems on platforms with multiple hardware accelerators. SIGPLAN Not. 44, 4 (feb 2009), 121–130.

[134] Riahi, A., Savadi, A., and Naghibzadeh, M. Comparison of analytical and ml-based models for predicting cpu–gpu data transfer time. Computing 102, 9 (sep 2020), 2099–2116.

[135] Rumpf, M., and Strzodka, R. Using graphics cards for quantized fem computations.

[136] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and Hwu, W.-m. W. Program optimization space pruning for a multithreaded gpu. In Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (New York, NY, USA, 2008), CGO '08, Association for Computing Machinery, p. 195–204.

[137] Sancho, J. C., Barker, K. J., Kerbyson, D. J., and Davis, K. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (2006), pp. 17–17.

[138] Schaa, D., and Kaeli, D. Exploring the multiple-gpu design space. In 2009 IEEE International Symposium on Parallel Distributed Processing (2009), pp. 1–12.

[139] Schatz, M. D., van de Geijn, R. A., and Poulson, J. Parallel matrix multiplication: A systematic journey. SIAM Journal on Scientific Computing 38, 6 (2016), C748–C781.

[140] Siek, J., Karlin, I., and Jessup, E. Build to order linear algebra kernels. In ' ' (05 2008), pp. 1 – 8.

[141] Smith, T. M., van de Geijn, R. A., Smelyanskiy, M., Hammond, J. R., and Van Zee, F. G. Anatomy of high-performance many-threaded matrix multiplication. In 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014) (2014).

[142] Solomonik, E., and Demmel, J. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Euro-Par Parallel Processing (01 2011), pp. 90–109.

[143] Song, F., and Dongarra, J. A scalable framework for heterogeneous gpu-based clusters. In Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (New York, NY, USA, 2012), SPAA '12, Association for Computing Machinery, p. 91–100.

[144] Song, F., Tomov, S., and Dongarra, J. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In Proceedings of the 26th ACM International Conference on Supercomputing (New York, NY, USA, 2012), ICS '12, Association for Computing Machinery, p. 365–376.

[145] Spagnoli, K. E., Humphrey, J. R., Price, D. K., and Kelmelis, E. J. Accelerating sparse linear algebra using graphics processing units. In Defense + Commercial Sensing (2011).

[146] Stone, J. E., Gohara, D., and Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science and Engineering 12, 3 (2010), 66–73.

[147] Tallent, N. R., Gawande, N. A., Siegel, C., Vishnu, A., and Hoisie, A. Evaluating on-node gpu interconnects for deep learning workloads. In High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation (Cham, 2018), S. Jarvis, S. Wright, and S. Hammond, Eds., Springer International Publishing, pp. 3–21.

[148] Tomov, S., Dongarra, J., and Baboulin, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing 36, 5-6 (June 2010), 232–240.

[149] Tomov, S., Nath, R., Ltaief, H., and Dongarra, J. Dense linear algebra solvers for multicore with GPU accelerators. In Proc. of the IEEE IPDPS'10 (Atlanta, GA, April 19-23 2010), IEEE Computer Society, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470941.

[150] Tsai, Y. M., Wang, W., and Chen, R. Tuning block size for qr factorization on cpu-gpu hybrid systems. In 2012 IEEE 6th International Symposium on Embedded Multicore SoCs (2012), pp. 205–211.

[151] Van De Geijn, R. A., and Watts, J. Summa: scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience 9, 4 (1997), 255–274.

[152] Van Zee, F. G. Implementing high-performance complex matrix multiplication via the 1m method. SIAM Journal on Scientific Computing 42, 5 (September 2020), C221–C244.

[153] Van Zee, F. G., and Smith, T. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. ACM Transactions on Mathematical Software 44, 1 (July 2017), 7:1–7:36.

[154] Van Zee, F. G., Smith, T., Igual, F. D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T. M., Marker, B., Killough, L., and van de Geijn, R. A. The BLIS framework: Experiments in portability. ACM Transactions on Mathematical Software 42, 2 (June 2016), 12:1–12:19.

[155] Van Zee, F. G., and van de Geijn, R. A. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software 41, 3 (June 2015), 14:1–14:33.

[156] Volkov, V., and Demmel, J. W. Benchmarking gpus to tune dense linear algebra. In SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008), pp. 1–11.

[157] Wang, L., Wu, W., Xiao, J., and Yang, Y. BLASX: A high performance level-3 BLAS library for heterogeneous multi-gpu computing. CoRR abs/1510.05041 (2015).

[158] Wang, Q., Zhang, X., Zhang, Y., and Yi, Q. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (New York, NY, USA, 2013), SC '13, Association for Computing Machinery.

[159] Werkhoven, B. v., Maassen, J., Seinstra, F., and Bal, H. Performance models for cpu-gpu data transfers. In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2014), pp. 11–20.

[160] "Whaley, R. C., and Padua, D. ATLAS (Automatically Tuned Linear Algebra Software). Springer US, Boston, MA, 2011, pp. 95–101.

[161] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J. Hierarchical dag scheduling for hybrid distributed systems. In 2015 IEEE International Parallel and Distributed Processing Symposium (2015), pp. 156–165.

[162] Xianyi, Z., Qian, W., and Yunquan, Z. Model-driven level 3 blas performance optimization on loongson 3a processor. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems (2012), pp. 684–691.

[163] Yi, Q., and Qasem, A. Exploring the optimization space of dense linear algebra kernels. In Languages and Compilers for Parallel Computing (Berlin, Heidelberg, 2008), J. N. Amaral, Ed., Springer Berlin Heidelberg, pp. 343–355.

[164] Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. Poet: Parameterized optimizations for empirical tuning. In ' ' (01 2007), pp. 1–8.

[165] Zhang, Y., and Owens, J. D. A quantitative performance analysis model for gpu architectures. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture (2011), pp. 382–393.

[166] Zhu, J., Lastovetsky, A., Ali, S., and Riesen, R. Communication models for resource constrained hierarchical ethernet networks. In Euro-Par 2013: Parallel Processing Workshops (Berlin, Heidelberg, 2014), D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds., Springer Berlin Heidelberg, pp. 259–269.

[167] Zhu, J., Lastovetsky, A., Ali, S., Riesen, R., and Hasanov, K. Asymmetric communication models for resource-constrained hierarchical ethernet networks. Concurrency and Computation Practice and Experience 27 (04 2015), 1575–1590.