Uncut-GEMMs : Communication-aware matrix multiplication on multi-GPU nodes

Petros Anastasiadis†, Nikela Papadopoulou*, Nectarios Koziris† and Georgios Goumas†
†Computing Systems Laboratory, National Technical University of Athens, Greece
*School of Computing Science, University of Glasgow, United Kingdom
Email: †{panastas,goumas,nkoziris}@cslab.ece.ntua.gr, *nikela.papadopoulou@glasgow.ac.uk

***Abstract*—General Matrix Multiplication (GEMM) is one of the most common kernels in high-performance computing (HPC) and machine-learning (ML) applications, frequently dominating their execution time, rendering its performance vital. As multi-GPU nodes have become common in modern HPC systems, GEMM is usually offloaded on GPUs as its compute-intensive nature is a good match for their architecture. On the other hand, despite the GEMM kernel itself being usually compute-bound, execution on multi-GPU systems also requires fine-grained communication and task scheduling to achieve optimal performance. While numerous multi-GPU level-3 BLAS libraries have faced these issues in the past, they are bound by older design concepts that are not necessarily applicable to modern multi-GPU clusters, resulting in considerable deviation from peak performance. In this work, we thoroughly analyze the current challenges regarding data movement, caching, and overlap of multi-GPU GEMM, and the shortcomings of previous solutions, and provide a fresh approach to multi-GPU GEMM optimization. We devise a static scheduler for GEMM, enabling a variety of algorithmic, communication, and auto-tuning optimizations, and integrate those in an end-to-end open-source multi-GPU GEMM library. Our library is evaluated on a multi-GPU NVIDIA HGX system with 8 NVIDIA A100 GPUs, achieving on average a 1.37x and 1.29x performance improvement over the state-of-the-art multi-GPU GEMM libraries, for double and single precision, respectively.**

I. Introduction

General Matrix Multiplication (GEMM) is a fundamental operation in high-performance computing (HPC) and machine learning (ML) applications, rendering its performance vital for their optimization. The inherent parallelism and high operational intensity of GEMM make it well-suited for GPU acceleration. Consequently, GEMM operations are often performed on powerful multi-GPU systems, where advances in GPU and interconnect technologies have enabled unprecedented performance peaks. A straightforward way to execute GEMM on a multi-GPU compute node is to utilize existing libraries for distributed GEMM that have been extended with GPU support [1]–[7]. However, these libraries result in inferior intra-node performance [8], [9], as they are tied to the distributed paradigm and designed for scalability on hundreds or thousands of workers, but not optimized for single-node performance. To obtain the maximum achievable *performance* on a single multi-GPU compute node, specialized libraries have been designed specifically for this case [8]–[12].

A compute-intensive, highly parallel operation like GEMM should theoretically reach the peak performance of a multi-GPU compute node. However, contrary to this belief, in practice, distributing the problem to multiple GPUs introduces *communication overheads*, *scheduling overheads*, and *imbalance*, resulting in a deviation from the expected peak performance for increasingly more problem sizes in modern multi-GPU nodes. To achieve near-optimal performance for multi-GPU GEMM, libraries use a number of optimizations targeting these overheads. First, communication optimizations are applied to reduce the communication volume, increase throughput via routing messages through faster links, and overlap communication with computation. Load balancing needs to be applied for both computation and communication. At the same time, any decision-making for scheduling needs to be lightweight, to minimize management and scheduling overheads. Multi-GPU GEMM optimization usually involves a specific subset of the aforementioned techniques, leaving room for significant improvements for the rest [12]. For example, the state-of-the-art XKBLAS [9] library 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 achieves good performance for regular problems, but does not perform well in nodes with irregular interconnects and non-square problem shapes. On the other hand, the state-of-the-art PARALiA [12] library focuses on advanced routing techniques and maximizes communication overlap and GPU load balance, using performance models, but this comes at the cost of increased communication volume, no communication balancing mechanisms, and higher scheduling overheads. Consequently, PARALiA provides robust performance for regular and irregular problems, but sacrifices some performance for generality and exhibits low performance in smaller problems.

In this work, we focus on the GEMM operation, as it constitutes the main building block for Level-3 BLAS and is ubiquitous in both HPC and machine-learning applications. We target multi-GPU compute nodes and offer an optimized library that achieves near-optimal performance for all problem sizes, shapes, and data placements. 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.

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.

II. Related work

The work related to single-node multi-GPU GEMM can be split into two categories: 1) distributed GEMM algorithms, which define the basis for GEMM communication optimization techniques for multiple workers, 2) libraries that support multi-GPU execution.

A. Distributed GEMM

Distributed GEMM optimization is a problem with decades of previous work, with different algorithms focusing mainly on communication reduction. Cannon's algorithm [13] is the first parallel GEMM for square problems, later extended by Fox et al. [14] to more problem shapes and processor grids. PUMMA [15] adds support for transposed matrices and explores data placement optimization for the first time. SUMMA [16] builds on top of PUMMA, optimizing its communication and adding blocking and communication/computation overlap, marking the commonly known *2D GEMM algorithm*. Agarwal et al. [17] show that a 3D decomposition of GEMM is better if memory is not a constraint. Solomonik and Demmel [18] extend this idea for practical use, introducing the 2.5D GEMM algorithm that balances decomposition between 2D and 3D, based on available memory, resulting results in optimal communication for square matrices. CARMA [19] uses a recursive algorithm to also optimize GEMM asymptotically for irregular shapes. Finally, COSMA [4] identifies that the CARMA algorithm can lead to increased communication, and provides an algorithm based on the red-blue pebble game that is communication-optimal for any shape and process number (M, N, K, p). From these approaches, only COSMA provides an implementation that also supports GPUs, based on CUDA-aware MPI.

B. GEMM on multi-GPU compute nodes

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. Therefore, a number of Level-3 BLAS libraries, supporting GEMM, focus on scheduling, overlap, and communication optimization techniques. Early approaches extend existing multi-core or distributed BLAS implementations with GPU support. Specifically, Kazushige et al. [20], propose a tiled BLAS algorithm, distributing tiles to GPUs, with no GPU-specific optimizations. MAGMA [1] extends a multi-core BLAS library using a static tile scheduler to target GPUs. DPLASMA [2] combines a distributed baseline with the optimized DAG-based task scheduler of PaRSEC [3] for tile distribution and computation on GPUs and introduces the notion of caching and reusing tiles in GPU memory buffers. All these approaches use a tiled data layout, bit are not compatible with LAPACK and follow a full-offload paradigm, assuming all input data resides on the CPU memory.

NVIDIA's cuBLASXt [10] and its BLAS-compliant wrapper NVblas [11] overcome this limitation, enabling LAPACK layout input matrices on CPU or any GPU memory and serving as the *state-of-practice* multi-GPU offload baseline. While cuBLASXt increases programmability and is a good drop-in replacement option for nodes with a single GPU, its decomposition and scheduler design are simplistic and result in severe performance degradation on multi-GPU nodes. To overcome this, BLASX [8] targets both programmability and robust performance for multi-GPU nodes, sharing the BLAS-compliant layout of NVBLAS, but internally decomposing the problem data to tiles similarly to PaRSEC. BLASX improves the GEMM inter-GPU communication pattern with a 2D-block cyclic decomposition, introduces a cache-like hierarchy that favors point-to-point $GPU \leftrightarrow GPU$ instead of $CPU \leftrightarrow GPU$ transfers whenever possible, and includes a work-stealing mechanism for GPUs to improve load balancing. XKBLAS [9] provides a modern level-3 multi-GPU BLAS library based on the lightweight DAG-based XKaapi [21] scheduler, that reduces scheduling overheads, fights imbalance more effectively than BLASX. An extension of XKBLAS [22] improves its routing algorithm from $GPU \leftrightarrow GPU$ preference to a distance-based heuristic that also considers the type of point-to-point connection between GPUs, resulting in state-of-the-art performance, when all matrices initially reside on the CPU. PARALiA [12] focuses on providing good performance regardless of the data placement, problem shape

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

and system homogeneity, through modeling and auto-tuning. PARALiA improves routing with a more accurate bandwidth-based routing algorithm, and can automatically use fewer devices based on modeling the problem bottlenecks to conserve energy, which considerably increases energy efficiency at the cost of higher scheduling overhead compared to DAGs. PARALiA outperforms XKBLAS in mixed data configurations and provides a better trade-off between energy and performance. While both XKBLAS and PARALiA provide solid performance for their targeted problems, they still fall short in reaching the peak performance of modern systems, for a considerable number of explored problems that remain communication-bound.

Alternative approaches target large-scale systems with multi-GPU nodes, and support very large problem sizes, with their data being pre-distributed to GPUs in PBLAS-like layouts. SLATE [5] implements the SUMMA [16] algorithm for GPUs, distributing problem data on their memories with a PBLAS user-friendly C++ data layout. PARSEC [3] offers a similar approach, which can support larger problems than SLATE with better scalability. cuBLASMp [7] is the most recent NVIDIA implementation for multi-node multi-GPU GEMM for pre-distributed PBLAS data layout.

To summarize, GEMM optimization is a complex problem with extensive previous work both for distributed and multi-GPU systems. Distributed approaches focus primarily on communication-optimal algorithms and scalability for PBLAS data layouts, while approaches specifically targeting multi-GPU nodes focus on intra-node performance for the BLAS LAPACK layout. In this work, we target multi-GPU compute nodes, aiming to provide robust, near-peak performance for any problem size and type, and any initial data placement. We note that state-of-the-art multi-GPU libraries for linear algebra [8], [9], [12], [22] already implement some of the optimizations introduced in the next section, like *caching* (Section III-C), *overlap* (Section III-B) and *BW-based routing* (Section III-D). 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 III-D2), *RONLY-fetch batching* (Section III-E), *lazy WR-tile fetching* (Section III-F) and *ETA-based sub-kernel ordering* (Section III-G).

III. Implementation

In this section, we describe the design of a static schedule inspired by distributed and multi-GPU approaches, that optimizes GEMM for multi-GPU. From the available BLAS or PBLAS standard input layouts for GEMM, we follow the BLAS standard [23], with the input matrices stored in LAPACK layout, residing either on host or GPU memory [8]–[10], [12]. A GEMM routine following the BLAS definition performs the operation:

$$C_{out} = a \cdot A \times B + b \cdot C_{in} \quad (1)$$

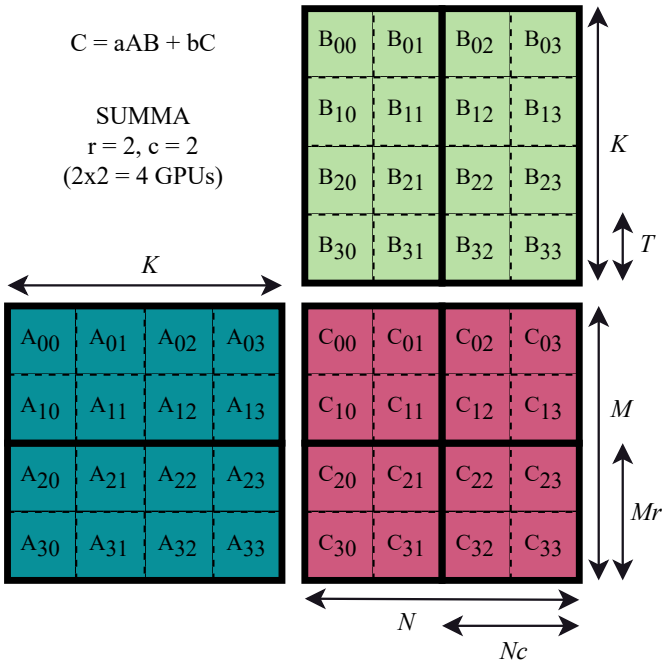

Fig. 1. An example of our GEMM 2-level hierarchical decomposition for a square problem (M, N, K) based on the SUMMA blocking algorithm [16]. The first level depends on the number of workers (here: 4 *GPUs*) split to a 2D grid $(r, c) = (2, 2)$ which decomposes (M, N) to (M_r, N_c) chunks, leaving K untouched. 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.

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.

A. 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 [8], [9], [12] . We select a 2D decomposition similar to the SUMMA [16] algorithm, as a simple and practical solution, upon which we apply a number of optimizations.

Figure 1 depicts our proposed *hierarchical decomposition* based on the blocking version of the SUMMA distributed algorithm [16]. 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 [16]:

$$C_{i,j} = b \cdot C_{i,j} + \sum_{k=0}^{\frac{K}{T}} a \cdot A_{i,k} \times B_{k,j} \quad (2)$$

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

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 [12], [24]–[27]. 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 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 $T = 128 \rightarrow min(M_r, N_c, K)$.

B. 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 2 depicts a simplified example of scheduling the first four sub-problems of the aforementioned decomposition of Figure 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

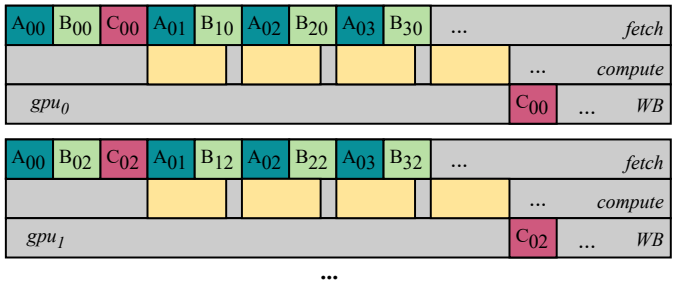

Fig. 2. The process of executing the tasks of the first four sub-problems as seen in Figure 1 on two GPUs in parallel. Tasks of different types (*fetch*, *compute*, *WB*) are placed on different streams and overlapped in a pipelined manner for each GPU, using different streams for intra-GPU parallelism.

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 $t_{total} \approx max((t_{fetch(A)} + t_{fetch(B)} + t_{fetch(C)}), t_{compute}, t_{WB}(C))$ [26]. 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 [28] 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 [29] similarly to previous approaches [8], [9], [12].

C. Data caching / Communication avoidance

To avoid excessive communication, most multi-GPU BLAS implementations [8], [9], [12] *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 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.

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

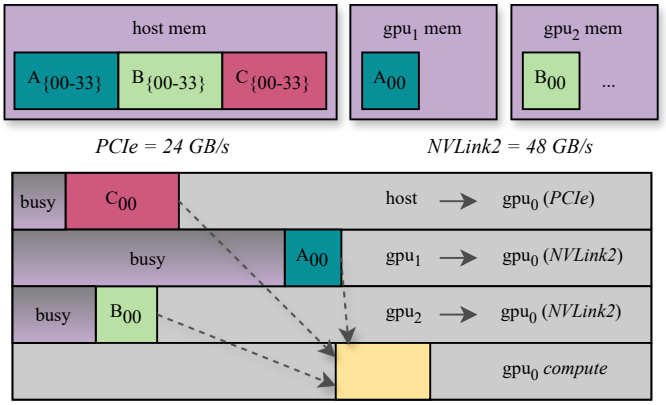

Fig. 3. An example of bandwidth-based routing misprediction that results in increased GPU idle time. When the sub-problem with input dependencies (A_{00}, B_{00}, C_{00}) is scheduled in gpu_0, A_{00} and B_{00} are already available in gpu_1 and gpu_2 from previous tile fetches. Since BW-based routing 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 $gpu_1 \rightarrow gpu_0$.

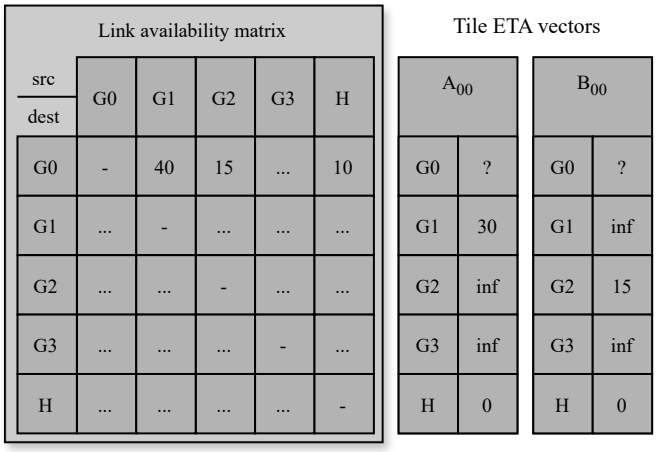

Fig. 4. An example state of LAM and ETA vectors for two tiles A_{00} and B_{00}. The tile ETA vectors show when these tiles should be available to GPUs 1 and 2, respectively (initial on host memory - ETA[h] = 0), while the LAM stores an estimation for the interconnect load up to that scheduling state.

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 III-A. 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.

D. 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. $fetch^{gpu_i}_{in\_loc}(A_T)$, 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.

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 [8], [12], [22]. 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 $BW^{gpu_i}_{in\_loc}$ and $BW^{gpu_j}_{gpu_i}$ 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 [12].

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 3, where A_{00} needs to be fetched to gpu_0. Bandwidth-based routing selects to fetch the tile from gpu_1, since $BW^{gpu_0}_{gpu_1} > BW^{gpu_0}_{host}$, 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 $t^{dst}_i = \frac{size}{BW^{dst}_i}$, searching

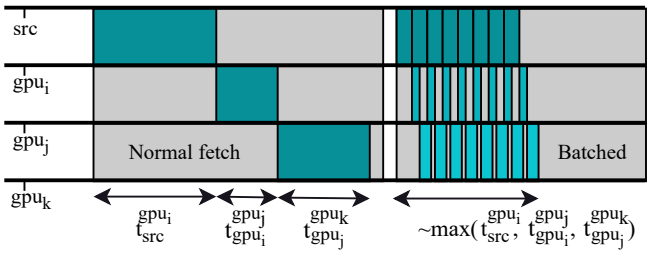

Fig. 5. An example of performing three fetches of the same data to three GPUs either one by one (left) or with a simultaneous broadcast-like batched fetch (right) that uses $p = 8$ sub-transfers to overlap the process in a pipelined manner. Batching all fetch operations together results in the same fetch cost for gpu_i, but considerably decreases the fetch costs for gpu_j and gpu_k.

for the source i with the lowest ETA_{min}, where:

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

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 write-backs to take into account their interconnect load as well but without routing optimization. Figure 4 shows the LAM for the example in Figure 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.

E. Optimizing RONLY tile transfers with batching

The standard approach for communication routing in previous work [8], [9], [12], [22] 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 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 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.

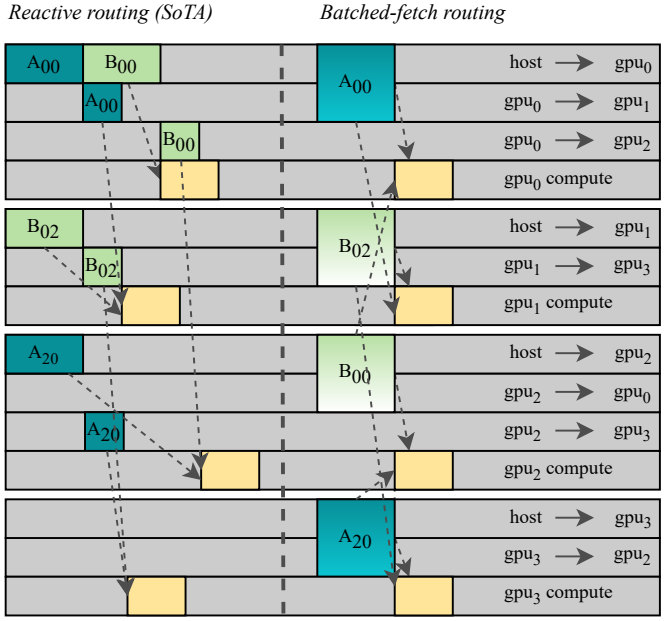

Fig. 6. An example of routing the first GEMM sub-problem on each GPU using a) reactive routing (left) employed by the SoTA and b) ETA-based proactive routing combined with RONLY batched-fetch (right) used in this work. Reactive routing optimizes the effective bandwidth by using faster links whenever 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.

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 III-D2 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. $fetch_{src}^{gpu_i}, fetch_{gpu_i}^{gpu_j}(T), ...$) 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 ($fetch_{src}^{\{gpus\_best\_order\}}$).

To show the importance of this optimization in multi-GPU GEMM routing, Figure 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 III-F). 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

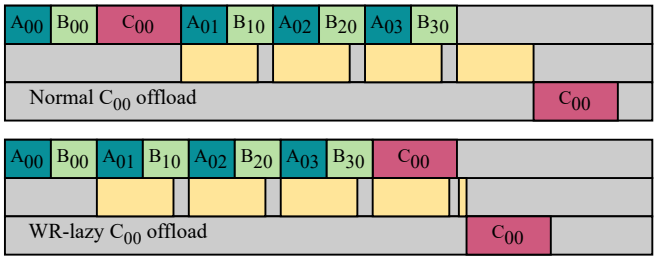

Fig. 7. Scheduling dependencies and computing four sub-problems on C_{00}, with a normal offload (top) or by using the WR-lazy fetch approach (bottom). WR-lazy fetching reduces GPU idle time, removing C_{00} from the input dependencies of the first sub-problem on the cost of a lightweight extra computation before writing back the result.

perfectly load-balanced execution in all GPUs.

F. 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 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 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 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 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 $B_{loc} == C_{loc}$.

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()
```

G. 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 III-B. The fetch tasks are optimized (in lines 14-15) as described in Sections III-C,III-D, and III-E 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.

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 [3], [9], [12]. A common

TABLE I
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

technique to optimize the sub-problem order is to prioritize the sub-problems with the minimum fetch operations [9]. 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 III-D2. 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.

IV. 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.

A. Experimental Setup

1) Testbed: For the performance evaluation, we use an NVIDIA HGX system, which is part of the acceleration nodes of the Karolina HPC cluster [30], and is described in Table I. 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).

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.

3) Dataset: For performance evaluation, we use two datasets: a *regular* dataset with square problems, as also reported in related work [6], [8], [9], 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 [12]. 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 $(0, 0, 0)$, therefore transfers can directly use the NVLink. The *irregular* dataset is described in Section IV-E.

B. Evaluation of performance optimizations

We first evaluate the optimizations described in Section III incrementally, to assess how each contributes to total performance. Figure 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 III-A) 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×. 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,

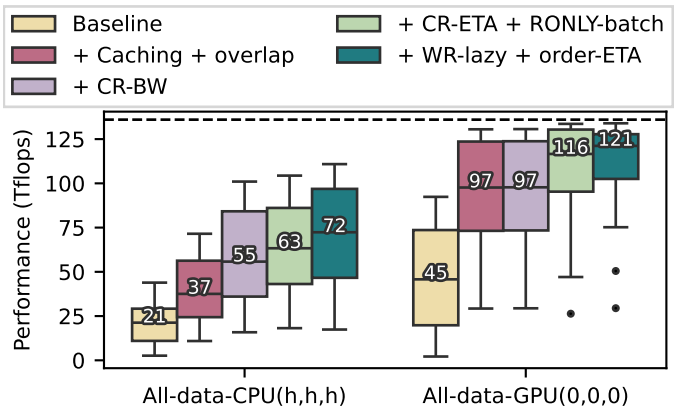

Fig. 8. The performance of each optimization described in Section. III for FP64 square GEMM (M=N=K) using all 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line), for the regular dataset of Section IV-A. The two clusters correspond to different configurations. Optimizations listed on the legend 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.

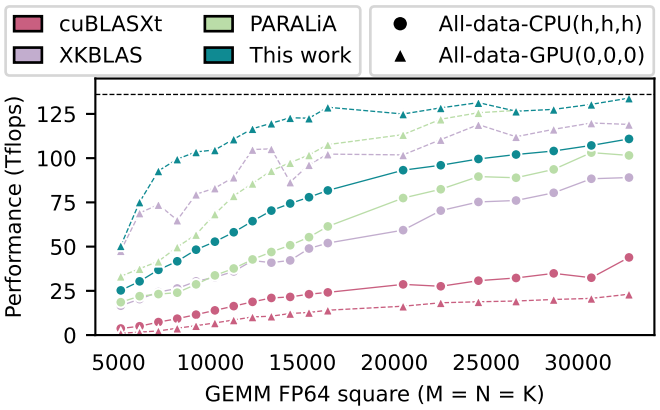

Fig. 9. The square GEMM (M=N=K) FP64 performance for the regular dataset of Section IV-A for 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line). Our approach offers robust performance regardless of the data placement, avoids imbalance, and outperforms all previous approaches, being more effective in communication-bound problem sizes (12 leftmost data points).

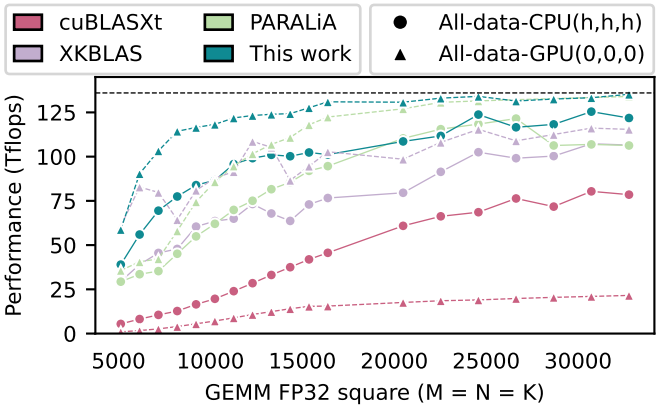

Fig. 10. The square GEMM (M=N=K) FP32 performance for the regular dataset of Section IV-A for 8 GPUs on our NVIDIA HGX testbed (system peak = dashed line). Using FP32 results in more compute-bound problems due to half the communication volume, coupled with the same FP32 performance peak. Our approach adjusts to the new ratios better than previous libraries, reaching the peak faster and providing superior performance for all problems.

increasing overlap and reducing GPU idle time. Finally, WR-lazy fetching, together with an improved ETA-based order selection for firing sub-problems results in an additional 1.1x speedup due to reduced GPU idle time.

C. 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 I. In particular, we evaluate XKBLAS [9] and PARALiA [12] and exclude previous approaches that they outperform [6], [8]. We also evaluate cuBLASXt [10], as the state-of-practice library, despite its inferior performance [8], [9], [12]. 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 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 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, XKBLAS and PARALiA by 2.7x, 1.37x and 1.28x for the *all-data-CPU* case and 10.8x,

[2]not to be confused with TF32

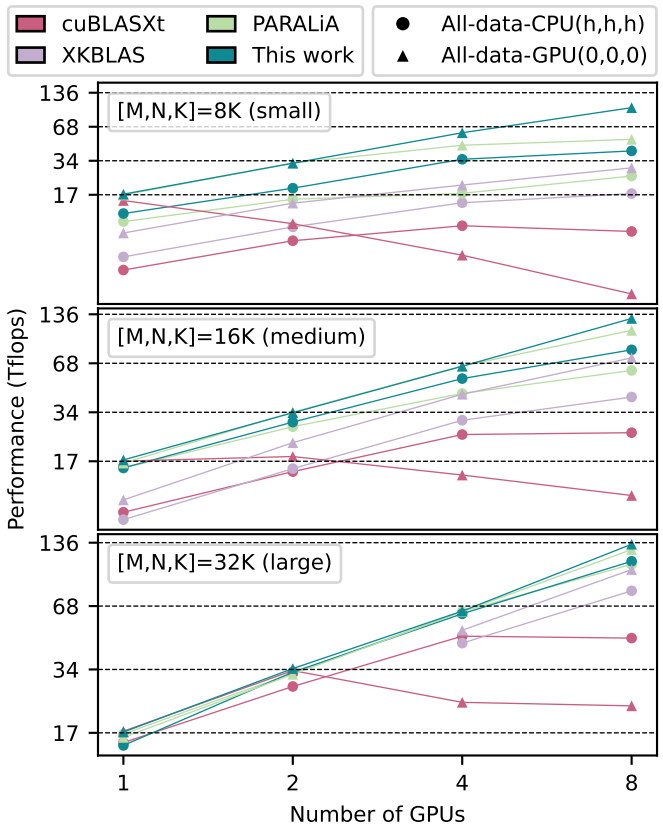

Fig. 11. Strong scaling analysis of three square GEMM (M=N=K) FP64 problem sizes for two data placements and variable number of GPUs on our NVIDIA HGX testbed (y-axis in log scale, system {1,2,4,8} GPU peak = dashed lines). Our approach provides the best performance for all configurations, and scales better than state-of-the art libraries as the number of GPUs increase, especially for the smaller more communication-bound problems.

1.42x and 1.31x for the *all-data-GPU* case, respectively.

D. 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 $\leq$ 4 GPUs. Therefore, scaling from 4 $\rightarrow$ 8 GPUs becomes more challenging than 1 $\rightarrow$ 2 and 2 $\rightarrow$ 4.

Figure 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 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 to inefficient communication routing that passes through the PCIe instead of using the much faster NVLink [12], [23]. 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 communication optimization, combined with lightweight scheduling, directly targets and effectively enhances performance and scalability.

E. 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 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

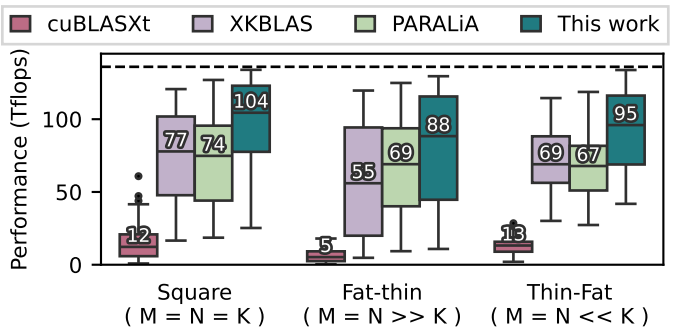

Fig. 12. Comparison of GEMM FP64 performance robustness against the state-of-the-art, using the expanded *irregular* dataset, divided in three clusters, according to the matrix shapes. Our approach outperforms all existing libraries, regardless of problem irregularity and data placement, providing a uniformly superior solution for multi-GPU GEMM.

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.

V. Conclusion

In this work, we contribute an optimized GEMM implementation tailored for efficient execution on multi-GPU compute nodes. 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 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.

VI. Acknowledgments

Funded by the European Union. This work has received funding from the European High-Performance Computing Joint Undertaking (EuroHPC JU) and Poland, Germany, Spain, Hungary, France, and Greece under grant agreement number 101093457. This publication expresses the opinions of the authors and not necessarily those of the EuroHPC JU and Associated Countries which are not responsible for any use of the information contained in this publication.

We also acknowledge the EuroHPC Joint Undertaking for awarding us access to the *accelerated compute nodes* hosted by Karolina at IT4Innovations, Czech Republic.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in GPU Computing Gems, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2. [Online]. Available: https://hal.inria.fr/inria-00547847

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

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

[4] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "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.

[5] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "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, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356223

[6] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra, "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.

[7] "developer.nvidia.com/cublasmp-downloads."

[8] L. Wang, W. Wu, J. Xiao, and Y. Yang, "BLASX: A high performance level-3 BLAS library for heterogeneous multi-gpu computing," CoRR, vol. abs/1510.05041, 2015. [Online]. Available: http://arxiv.org/abs/1510.05041

[9] T. Gautier and J. V. F. Lima, "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.

[10] "developer.nvidia.com/cublasxt."
[11] "docs.nvidia.com/cuda/nvblas."
[12] P. Anastasiadis, N. Papadopoulou, G. Goumas, N. Koziris, D. Hoppe, and L. Zhong, "Paralia: A performance aware runtime for auto-tuning linear algebra on heterogeneous systems," ACM Trans. Archit. Code Optim., vol. 20, no. 4, dec 2023. [Online]. Available: https://doi.org/10.1145/3624569
[13] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, USA, 1969, aAI7010025.
[14] G. Fox, S. Otto, and A. Hey, "Matrix algorithms on a hypercube i: Matrix multiplication," Parallel Computing, vol. 4, no. 1, pp. 17–31, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819187900603
[15] J. Choi, D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," Concurrency: Practice and Experience, vol. 6, no. 7, pp. 543–570, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330060702
[16] R. A. Van De Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," Concurrency: Practice and Experience, vol. 9, no. 4, pp. 255–274, 1997.
[17] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," IBM Journal of Research and Development, vol. 39, no. 5, pp. 575–582, 1995.
[18] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," 01 2011, pp. 90–109.
[19] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 261–272.
[20] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," ACM Trans. Math. Softw., vol. 35, no. 1, jul 2008. [Online]. Available: https://doi.org/10.1145/1377603.1377607
[21] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "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.
[22] T. Gautier and J. V. F. Lima, "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.
[23] J. Dongarra, "Basic linear algebra subprograms technical (blast) forum standard ii," IJHPCA, vol. 16, pp. 1–111, 05 2002.
[24] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance models for asynchronous data transfers on consumer graphics processing units," Journal of Parallel and Distributed Computing, vol. 72, no. 9, pp. 1117 – 1126, 2012, accelerators for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731511001468
[25] C. Gregg and K. Hazelwood, "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.
[26] B. v. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for cpu-gpu data transfers," in 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014, pp. 11–20.
[27] P. Anastasiadis, N. Papadopoulou, G. Goumas, and N. Koziris, "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.
[28] "developer.nvidia.com/cublas."
[29] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[30] "it4i.cz/en/infrastructure/karolina."