



KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU

Hemeng Wang
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

hemeng.wang@student.cup.edu.cn

Xiaowen Tian
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

xiaowen.tian@student.cup.edu.cn

Yang Du
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

yang.du@student.cup.edu.cn

Qingxiao Sun
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

qingxiao.sun@cup.edu.cn

Sidu Li
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

sidu.li@student.cup.edu.cn

Weifeng Liu
SSSLab, Dept. of CST
China University of
Petroleum-Beijing
Beijing, China

weifeng.liu@cup.edu.cn

Abstract

Efficient general matrix-matrix multiplication (GEMM) has attracted significant research attention in HPC and AI workloads. While large-scale GEMM has nearly achieved the peak floating-point performance of GPUs, substantial opportunities for optimization remain in small and batched GEMM operations.

We in this paper propose KAMI, a set of 1D, 2D, and 3D GEMM algorithms that extend the theory of communication-avoiding (CA) techniques within a single GPU. KAMI optimizes thread block-level GEMM by utilizing tensor cores as computational units, low-latency thread registers as local memory, and high-latency on-chip shared memory as a communication medium. We provide a theoretical analysis of CA performance from the perspective of GPU clock cycles, rather than the traditional execution time. Also, we implement sparse-dense matrix-matrix multiplication (SpMM) and sparse general matrix-matrix multiplication (SpGEMM) with this compute-communication pattern. Experimental results for general, low-rank, batched, and sparse multiplication on NVIDIA, AMD, and Intel GPUs show significant performance improvements over existing libraries cuBLAS, cuBLASDx, CUTLASS, MAGMA, and SYCL-Bench.

CCS Concepts

- Computing methodologies → Parallel algorithms; Linear algebra algorithms;
- Theory of computation → Communication complexity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1466-5/25/11

<https://doi.org/10.1145/3712285.3759895>

Keywords

GPU, Communication-avoiding, GEMM, SpMM, SpGEMM

ACM Reference Format:

Hemeng Wang, Yang Du, Sidu Li, Xiaowen Tian, Qingxiao Sun, and Weifeng Liu. 2025. KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3712285.3759895>

1 Introduction

General matrix-matrix multiplication (GEMM) [88, 192] is in general the most time-consuming operation in HPC applications [13, 19, 123, 158] and AI workloads [51, 79, 195]. In recent years, many studies focused on optimizing large-scale GEMM on multi-core and many-core processors, in particular GPUs, to achieve near peak performance [106, 121, 124, 127, 140, 180, 181, 196, 202, 224]. Normally, as long as the matrix is sufficiently large, both square matrix multiplication [49, 139] and tall-and-skinny matrix multiplication [53, 163, 167, 210] often achieve near peak performance. However, small and batched matrices tend to struggle in reaching peak performance at most sizes [10, 98, 169, 209, 211].

According to research [97, 122], the main cause of this phenomenon is the excessive access to remote memory. Considering the $O(n^3)$ computational complexity and $O(n^2)$ memory access complexity of GEMM, a small value of n fails to provide sufficient arithmetic intensity to effectively utilize modern processors and thus requires better data locality. On the other hand, taking a CUDA thread on an NVIDIA Hopper GPU [57] as an example, the latency and bandwidth for accessing its registers are about 20 times and 4 times faster, respectively, compared to accessing on-chip shared memory [136]. Therefore, a more effective strategy is to ensure that the data involved in GEMM is sourced directly from registers, rather than from the significantly slower shared memory. However, existing research largely overlooks this issue and fails to effectively leverage the multiple memory hierarchies of GPUs to optimize small-scale GEMM.

Reducing the cost of remote data access has been a longstanding challenge in distributed computing. A series of communication-avoiding (CA) algorithms proposed by Demmel et al. [18, 22, 65] have demonstrated their great effectiveness across a broad spectrum of distributed problems, including matrix computations [72, 91, 92, 112, 141, 165], graph processing [172, 174], N-body simulations [77, 113], and machine learning [212, 215]. The necessity of accessing faster local memories, combined with the theoretical foundations of CA algorithms, motivates our exploration of CA techniques on a single GPU to accelerate small-scale GEMM.

In this paper, we present KAMI, to the best of our knowledge, the first attempt to extend CA theories and techniques within a single GPU to accelerate matrix multiplication. We reorganize the three primary on-chip components – tensor core units, registers, and shared memory – to formulate our 1D, 2D and 3D CA algorithms for GEMM. Specifically, tensor core units function as the computational units, registers serve as local memory for storing matrices A, B, and C, while shared memory acts as a communication medium for transferring submatrices between the computational units. Additionally, rather than relying on execution time, we employ the number of GPU clock cycles as the unit of theoretical analysis to perform a more detailed study of our CA algorithms. To exploit sparsity, KAMI also supports sparse-dense matrix-matrix multiplication (SpMM) and sparse general matrix-matrix multiplication (SpGEMM), utilizing the same CA schemes, built upon a Z-Morton order storage format.

We conduct extensive experimental evaluations on four GPUs: NVIDIA GH200 and 5090, as well as AMD 7900 XTX and Intel Max 1100, and compare KAMI with cuBLASDx [156], CUTLASS [157], cuBLAS [155], MAGMA [149] and SYCL-Bench [120]. In block-level GEMM, KAMI achieves up to 5.20x, 74.36x and 14.48x speedups over cuBLASDx, CUTLASS, SYCL-Bench for square GEMM and 6.11x and 11.61x over cuBLASDx, CUTLASS for low-rank GEMM, respectively. For batched tasks, KAMI achieves up to 713.93x and 332.02x speedups over cuBLAS and MAGMA.

This work makes the following contributions:

- We propose KAMI to extend CA algorithms within a single GPU to accelerate small-scale matrix multiplication.
- We present a new theoretical analysis scheme for communication and computation in GPU clock cycles.
- We exploit sparsity and block-wise Z-Morton storage for supporting SpMM and SpGEMM in our CA methods.
- We implement KAMI on NVIDIA, AMD and Intel GPUs, and show obviously faster performance over SOTA works.

2 Background

2.1 Matrix Multiplication

GEMM operation multiplies a dense matrix A of size m -by- k with a dense matrix B of size k -by- n , and gives a resulting dense matrix C of size m -by- n , as shown in Figure 1(a). When accounting for sparsity, GEMM can become SpMM (sparse A , dense B and C , see Figure 1(b)) and SpGEMM (sparse A , B and C , see Figure 1(c)).

Moreover, there are two additional types of matrix multiplication: 1) Low-rank GEMM (see Figure 1(d)) leverages the observation that matrices may exhibit an inherent low-rank structure and can

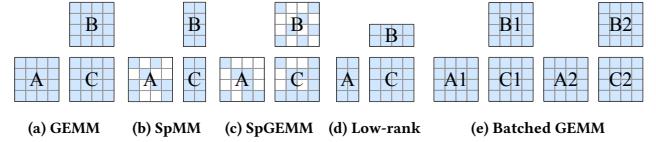


Figure 1: Different variants of matrix multiplication.

be approximated as products of smaller matrices to reduce the number of arithmetic operations [8, 133, 138]. 2) Batched GEMM (see Figure 1(e)) collects a number of independent small-scale GEMM operations and executes them in a single workload to saturate many-core processors [10, 76, 117, 127].

2.2 CA Methods

For large-scale problems executed on distributed platforms, communication often emerges as a bottleneck. CA algorithms aim to minimize data transfer between computational nodes, thereby mitigating this performance problem [18, 22, 65].

CA methods can be broadly categorized into three distinct approaches: 1) The 1D algorithm minimizes data transfer by optimizing the allocation of matrix rows across processing units, thereby reducing inter-node communication. 2) The 2D approach extends this optimization by partitioning both rows and columns, effectively minimizing communication along both dimensions of the matrix. 3) The 3D method further enhances communication efficiency by introducing a third dimension of partitioning, which improves data locality and substantially reduces memory access overhead, leading to more efficient computational performance.

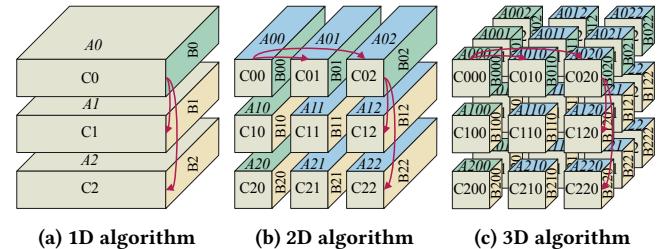


Figure 2: Three CA algorithms.

Figures 2(a), (b) and (c) show the three methods, respectively. It is worth noting that additional variants, such as 1.5D [109] and 2.5D [176], also exist. However, to maintain focus in our study, we concentrate on the classic 1D, 2D, and 3D approaches in this paper.

3 Motivation

3.1 Performance Issue of Small-Scale GEMM

In general, while large-scale GEMM is primarily computation-bound, small GEMM remains significantly constrained by memory accesses, in terms of bandwidth and latency [97, 122, 132, 150, 160]. We evaluate double precision cuBLAS [155] using square matrices of orders ranging from 1 to 8192, and cuBLASDx (a block-level extension to cuBLAS) [156] from 1 to 98 (could not be larger due to the limitation of shared memory capacity) on an NVIDIA GH200 GPU.

As illustrated in Figure 3, cuBLAS approaches near peak performance for large-scale GEMM. In contrast, when the matrix size is small, the performance of cuBLAS degrades significantly. For

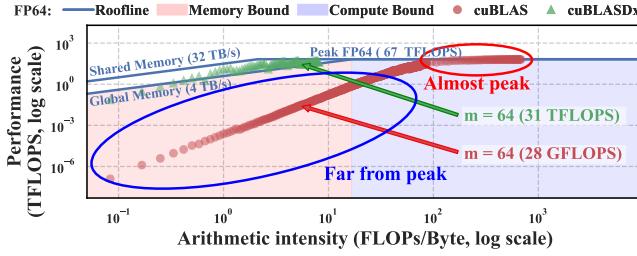


Figure 3: A roofline model of GEMM performance on an NVIDIA GH200 GPU. For cuBLAS, the kernel is repeated 1000 times to report the average, and cuBLASDx is evaluated with 16384 concurrent thread blocks, each looping 1000 times inside the CUDA kernel to ignore global I/O costs.

example, when $m = 64$, the performance drops to only 28 GFLOPS. Additionally, assuming no global memory load/store and executing a large amount of block-level small-scale GEMM in cuBLASDx, when $m = 64$, FP64 GEMM achieves only 31 TFLOPS, which corresponds to merely 46% of the theoretical peak.

Although small-scale GEMM of specific sizes can achieve near-peak performance (e.g., size $m = 128$, $n = 128$ and $k = 32, 64, 128$, depending on precision and shared memory size, used as the building block for large GEMM in CUTLASS [157]), most arbitrary sizes still exhibit substantial room for performance improvement. The importance of small-scale GEMM arises from its prevalence in real-world applications, such as low-rank approximation [90, 138], block-wise scientific solvers [82, 135, 197], batched neural network inference [86, 194], and transformer models with block-sparse attention [218]. In these scenarios, matrix sizes are typically small (often ≤ 128 in one or more dimensions), but must be computed repeatedly and in parallel, making throughput-critical optimization essential. This motivates our exploration of strategies to enhance the efficiency of small-scale GEMM on GPUs.

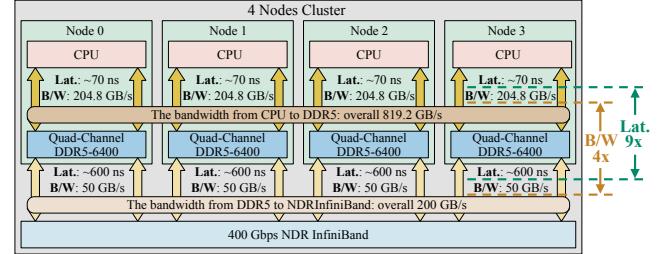
3.2 Distributed and GPU Memory Hierarchies

In distributed environments, parallelism is typically achieved at the process level, where each process stores its data in the local memory (e.g., DDR5 DRAM). When necessary, processes communicate through networks (e.g., InfiniBand). The performance is commonly evaluated by execution time.

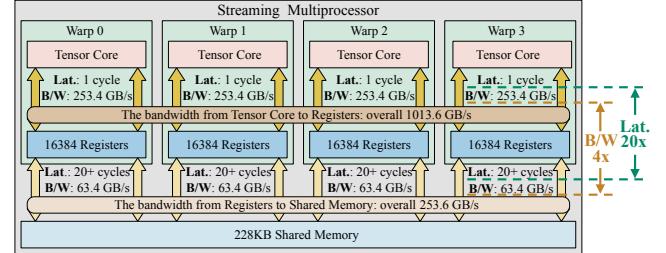
Modern GPUs are composed of multiple streaming multiprocessors (SMs), each runs several thread blocks. Within a block, a number of 32-thread warps are assigned to hardware compute units. For a warp's workload, data are stored in registers, and utilize CUDA cores or tensor cores to perform operations such as matrix multiplication. Inner-block data exchange between warps could only be achieved by shared memory with synchronizations. Also, unlike networks, where concurrent message passing is supported, broadcast between warps are performed serially due to the limited number of shared memory banks. The performance of GPU tasks can be evaluated by GPU clock cycles.

To highlight the differences and similarities, Figure 4(a) illustrates the latency and bandwidth for a 4-node cluster, whereas Figure 4(b) depicts similar metrics within an SM in a GPU.

As can be seen, distributed and GPU memory hierarchies show similar latency and bandwidth differences between local and remote



(a) Latency and bandwidth of each node on a cluster.



(b) Latency and bandwidth of each warp on an SM.

Figure 4: Latency and bandwidth comparison of the memory hierarchy of a 4-node cluster and a 4-warp SM.

storage. The latency differences are about 9x (70 ns vs. 600 ns, see Figure 4(a)) and 20x (1 cycle vs. 20 cycles, see Figure 4(b)), respectively. Moreover, the bandwidth differences are about 4x (819.2 GB/s vs. 200 GB/s, see Figure 4(a)) and 4x (1013.6 GB/s vs. 253.6 GB/s, see Figure 4(b)), respectively.

The variations in memory access across different hierarchical levels are basically consistent in both distributed environments and a single SM in a GPU. This similarity motivates us to investigate whether the distributed CA algorithms can be transferred to accelerate small-scale GEMM within a single GPU.

4 KAMI

4.1 Overview

In this paper, we propose KAMI, a set of CA algorithms accelerating small-scale GEMM, SpMM and SpGEMM of order up to about 200 within a single GPU. The interfaces are aligned to thread block level libraries such as cuBLASDx [157] and MAGMA [182].

Concept	Classic CA	KAMI (our work)
Compute unit	Process on CPU/GPU	Warp on tensor core
Local storage	DRAM	Thread register
Communication	Send/Recv by network	LD/ST on shared mem.
Perf. metric	Execution time	GPU clock cycle

Table 1: Concept of classic CA and KAMI.

Table 1 compares KAMI with classic CA methods, highlighting their differences in key concepts: 1) Classic CA operates at coarse-grained process level on CPUs or GPUs in distributed environments, whereas KAMI employs fine-grained parallelism at the warp level within GPU thread blocks calling tensor cores; 2) Classic CA stores data in the DRAM of the node, while KAMI utilizes registers for local storage; 3) Classic CA depends on inter-connection networks

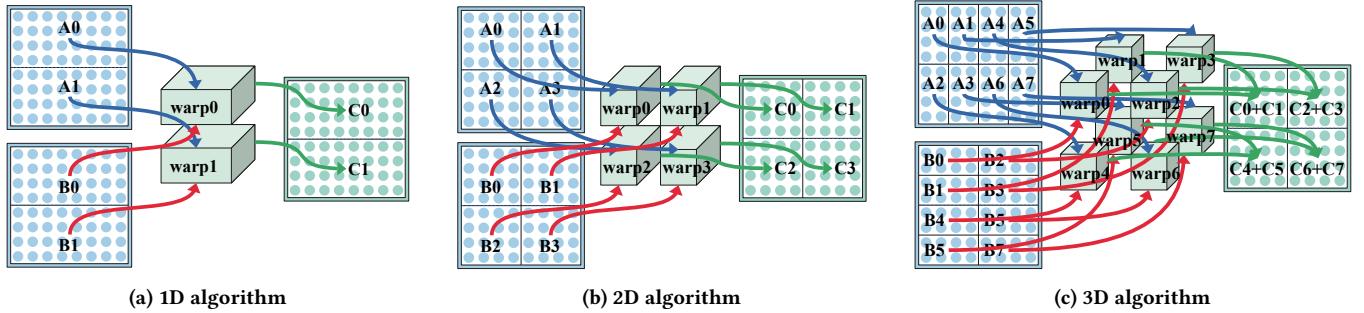


Figure 5: Matrix partitioning and memory hierarchy mapping before and after execution. Subfigures (a), (b), and (c) illustrate the data layout under the 1D ($p = 2$), 2D ($p = 4$), and 3D ($p = 8$) CA algorithms where p is the number of warps, respectively. In all algorithms, input matrices A and B are statically partitioned into p submatrices and initially reside in global or shared memory. The output matrix C is partitioned into p submatrices for 1D and 2D, and into $\sqrt[3]{p}$ submatrices for 3D. After computation, each warp holds $\frac{1}{p}$ of C in 1D and 2D, and $\frac{1}{\sqrt[3]{p}}$ in 3D aggregated from $\sqrt[3]{p}$ intermediate layers. This figure emphasizes the mapping between global memory and register files.

for process communication, in contrast to KAMI which utilizes on-chip shared memory for inter-warp data exchange; 4) Classic CA typically measures execution time, while KAMI adopts GPU clock cycles as a hardware-centric metric.

In KAMI, matrix multiplication is executed using multiple warps within a block, with each warp responsible for holding and processing a portion of the submatrix (Section 4.2). KAMI implements the CA algorithms in three fashions: 1D (Section 4.3), 2D (Section 4.4) and 3D (Section 4.5). Through cycle-grained modeling, we quantitatively characterize the computational and communication costs, enabling more accurate performance prediction across various hardware configurations. Beyond GEMM, we also consider sparsity to support both SpMM and SpGEMM (Section 4.6) on top of a Z-Morton ordered sparse block storage [43, 143]. We further introduce some key implementation details on NVIDIA tensor cores, AMD matrix cores and Intel Xe Matrix eXtensions (Section 4.7).

Table 2 provides the notation and definitions in this paper.

Symbol	Definition
$\mathbf{A}, \mathbf{B}, \mathbf{C}$	Matrices \mathbf{A} , \mathbf{B} and \mathbf{C} of size m -by- k , k -by- n , m -by- n
$A_i, \text{Sm}A_i$	Submatrix i of \mathbf{A} in registers, and in shared memory
$\mathbf{A}[:, i:j]$	The i^{th} column to $(j-1)^{th}$ column of matrix \mathbf{A}
s_e	Size of a single matrix element (bytes)
flops(\mathbf{A}, \mathbf{B})	Total arithmetic operations for multiplying \mathbf{A} and \mathbf{B}
O_{tc}	Arithmetic operations per cycle by each tensor core
ntc	Number of tensor cores per SM
p	Number of warps for parallel execution
L_{sm}	Latency from register to shared memory (cycles)
B_{sm}	Bandwidth of shared memory (bytes per cycle)
V_{cm}	Communication volume (bytes)
T_{cm}, T_{cp}	Number of communication, computation cycles (cycles)
T_{all}	All costs, sum of T_{cm} and T_{cp} (cycles)
θ_r, θ_w	Bank conflict factors of read and write, respectively $(0 \leq \theta \leq 1, \theta = 1$ means no conflicts)
$\text{AiSend}/\text{Recv}$	Submatrix i of \mathbf{A} to store/load between warps

Table 2: Notation and definitions.

4.2 Data Layout

In KAMI, matrices A, B, and C can be initially stored in global or shared memory. These matrices are partitioned into submatrices according to different CA algorithms during computation, as shown in Figures 5(a), (b) and (c).

In the 1D algorithm (Figure 5(a)), matrices A, B, and C are partitioned into p row-wise submatrices, where p is the number of warps. Each warp loads its corresponding submatrices of A (size $\frac{m}{p} \times k$) and B (size $\frac{k}{p} \times n$) from global or shared memory into registers and performs matrix multiplication. Since matrix B is shared among multiple warps, its submatrices are transferred through shared memory in a row-wise manner, conceptually similar to process communication. After multiplication, each warp writes its resulting submatrix of C (size $\frac{m}{p} \times n$) back to global or shared memory in row-wise.

In the 2D algorithm (Figure 5(b)), matrices A, B, and C are further partitioned into $\sqrt{p} \times \sqrt{p}$ two-dimensional submatrices. Each warp loads its corresponding submatrix of A (size $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$) and B (size $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$) from global or shared memory into registers and performs matrix multiplication. During computation, the 2D algorithm exchanges submatrices of A between warps in the same row and submatrices of B between warps in the same column via shared memory. After multiplication, each warp writes its resulting submatrix of C (size $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$) back to global or shared memory based on its two-dimensional position.

In the 3D algorithm (Figure 5(c)), matrices A and B are further subdivided into $\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$ three-dimensional submatrices, while matrix C maintains the $\sqrt[3]{p} \times \sqrt[3]{p}$ two-dimensional submatrices partitioning. Each warp loads its corresponding submatrix of A (size $\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}}$) and B (size $\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$) from global or shared memory into registers and performs matrix multiplication. Similar to the 2D algorithm, the 3D algorithm exchanges submatrices of A between warps in the same row and submatrices of B between warps in the same column via shared memory as a communication medium.

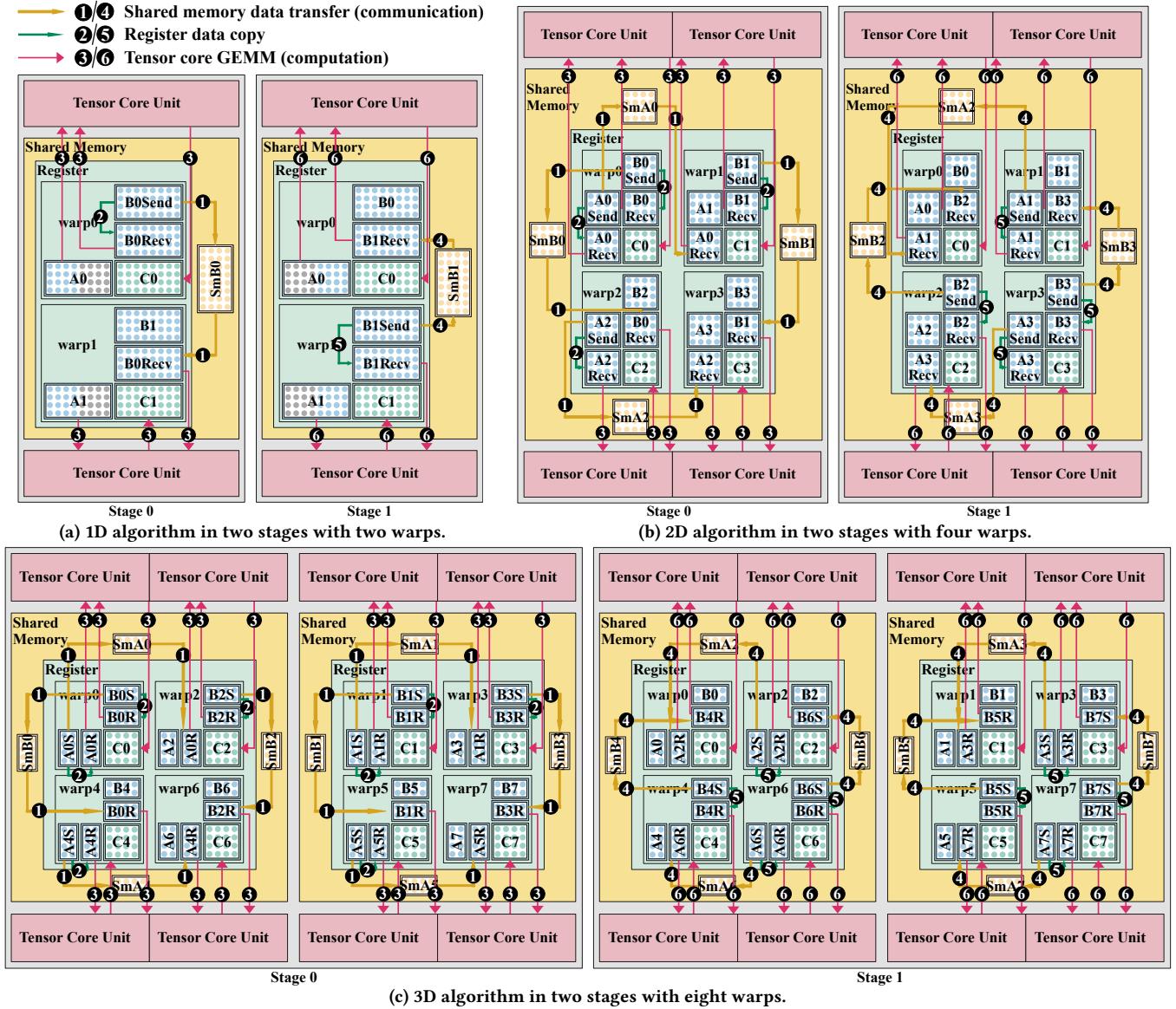


Figure 6: Examples of the 1D, 2D and 3D CA algorithms execution flow in KAMI. This figure depicts the interaction between shared memory and register files across stages. All algorithms follow a unified execution pattern: 1) inter-warp data transfers; 2) intra-warp register transfers facilitate submatrix alignment; and 3) submatrix multiplications are carried out using Tensor Cores. This figure captures the dynamic data residency and flow across the memory hierarchy during execution.

After multiplication, each warp accumulates results for the corresponding submatrix of C (size $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$) at the same position within the two-dimensional subblock before writing the final accumulated results back to global or shared memory.

4.3 1D Algorithm

In the 1D algorithm, p warps work for one multiplication operation, and each warp (denoted as warp i , where $0 \leq i < p$) holds submatrices A_i (size $\frac{m}{p} \times k$) and B_i (size $\frac{k}{p} \times n$). The GPU warps work in the SPMD (Single Program Multiple Data) model, meaning that each warp executes the same program concurrently.

The matrix multiplication task is then decomposed into p stages, each consisting of communication and computation phases. In the z^{th} stage ($0 \leq z < p$), the communication phase broadcasts the submatrix block $B_z\text{Send}$, which is the B_i held by the z^{th} warp, to the other warps.

Notably, in the 1D algorithm, communication occurs only for matrix B, and matrix A is not communicated. The communication runs in two steps: 1) The submatrix $B_z\text{Send}$ is loaded from registers into shared memory and stored as SmB_z ; 2) The other warps read SmB_z from shared memory into their registers and store it as $B_z\text{Recv}$ (steps ① and ④ in Figure 6(a), and lines 6 and 10 in Algorithm 1).

To reduce shared memory access pressure, after writing $\mathbf{B}_z\text{Send}$ to shared memory, the z^{th} warp also writes the same data into its local registers as $\mathbf{B}_z\text{Recv}$, (steps ② and ⑤ in Figure 6(a), and line 7 in Algorithm 1).

Algorithm 1 1D algorithm by p warps

```

1:  $i \leftarrow \text{warpID}$ 
2: GMem2Reg( $A_i \leftarrow A, B_i \leftarrow B, C_i \leftarrow C$ )
3:  $\_\text{syncthreads}()$ 
4: for  $z = 0$  to  $p$  do                                 $\triangleright$  The algorithm consists of  $p$  stages.
5:   if  $i = z$  then
6:     Reg2SMem( $\text{SmB} \leftarrow \text{BSend}$ )           $\triangleright$  Write  $\text{BSend}$  to shared memory.
7:     Reg2Reg( $\text{BRecv} \leftarrow \text{BSend}$ )            $\triangleright$  Copy  $\text{BSend}$  within registers.
8:    $\_\text{syncthreads}()$ 
9:   if  $i \neq z$  then
10:    SMem2Reg( $\text{BRecv} \leftarrow \text{SmB}$ )           $\triangleright$  Read  $\text{SmB}$  from shared memory.
11:     $\_\text{syncthreads}()$ 
12:     $C_i \leftarrow \text{TensorCoreGEMM}(A_i[:, :z \times \frac{k}{p}, (z+1) \times \frac{k}{p}], \text{BRecv})$        $\triangleright$  Part of  $A_i$  and  $\text{BRecv}$  multiplied by Tensor Core.
13: Reg2GMem( $C_i \leftarrow C$ )
  
```

Once all warps have their $\mathbf{B}_z\text{Recv}$, they begin the computation phase (steps ③ and ⑥ in Figure 6(a), and line 12 in Algorithm 1). The computation is multiplying the z^{th} portion of A_i (size $\frac{m}{p} \times \frac{k}{p}$) with the received $\mathbf{B}_z\text{Recv}$ (size $\frac{k}{p} \times n$) on tensor cores.

After completing the computation for the current stage, the algorithm proceeds to the next, repeating the procedure until all p stages are finished. Now each warp can save its computed C_i (size $\frac{m}{p} \times n$) in the registers to global or shared memory.

We now analyze the communication and computation time overheads. To simplify, we assume that the communication within the same warp is disregarded. Thus, the total communication volume consists of two components: writing B_i (size $\frac{k}{p} \times n$) to shared memory by one warp, and reading it from shared memory by $p - 1$ warps. Taking s_e as the byte size of a matrix element, the total communication volume V_{cm} is given by

$$V_{cm} = 1 \times \left(\frac{k}{p} \times n \right) \times s_e + (p-1) \times \left(\frac{k}{p} \times n \right) \times s_e = kn \times s_e. \quad (1)$$

Besides communication volume, we also consider shared memory access latency L_{sm} , bandwidth B_{sm} , and bank conflict factors θ_r and θ_w . Then, the communication cost T_{cm} can be expressed as

$$T_{cm} = L_{sm} + \frac{kn \times s_e}{\theta_w p B_{sm}} + \frac{(p-1)kn \times s_e}{\theta_r p B_{sm}}. \quad (2)$$

Next, we consider the algorithm's computational cost

$$T_{cp} = \frac{\text{flops}(A_i, B_z^{\text{Recv}})}{O_{tc}} = \frac{2 \times \frac{m}{p} \times \frac{k}{p} \times n}{O_{tc}} = \frac{2mnk}{p^2 O_{tc}}, \quad (3)$$

where O_{tc} represents the number of arithmetic operations per cycle by each tensor core.

The algorithm has p stages, each consisting of one communication phase followed by p concurrent computations. Therefore, the total execution cost T_{all} for the entire process is

$$\begin{aligned} T_{all} &= p \times (T_{cm} + \frac{p}{n_{tc}} \times T_{cp}) \\ &= L_{sm}p + \frac{kn \times s_e}{\theta_w B_{sm}} + \frac{(p-1)kn \times s_e}{\theta_r B_{sm}} + \frac{2mnk}{n_{tc} O_{tc}}. \end{aligned} \quad (4)$$

To provide a more concrete example, suppose in Figure 6(a), two warps ($p = 2$) multiply 8×8 matrices A and B ($m = n = k = 8$), and $s_e = 8$ in FP64. Through Formula 1, $V_{cm} = 512$ bytes.

Assuming that the shared memory latency $L_{sm} = 22$ cycles, the bank conflict factors $\theta_r = \theta_w = 1$, and the shared memory bandwidth $B_{sm} = 128$ bytes per cycle, bring in Formula 2, $T_{cm} = 26$ cycles.

If the tensor core performs 32 arithmetic operations per cycle and we have 4 tensor cores each SM ($O_{tc} = 32$ and $n_{tc} = 4$), bring in Formula 3, $T_{cp} = 8$ cycles. Thus, the total execution cost is $T_{all} = 60$ cycles as Formula 4.

4.4 2D Algorithm

In the 2D algorithm, p warps are divided into a $\sqrt{p} \times \sqrt{p}$ grid for one multiplication operation, and warp i works in the SPMD model and holds A_i (size $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$) and B_i (size $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$).

The multiplication task now has \sqrt{p} stages, each consisting of communication and computation phases. In the z^{th} stage ($0 \leq z < \sqrt{p}$), the communication phase broadcasts a submatrix block $A_j\text{Send}$, which is the A_i held by the z^{th} column of the warp grid, to the other warps in the same row, and $B_j\text{Send}$, i.e. B_i held by the z^{th} row of the warp grid, to the other warps in the same column.

The communication runs in two steps: 1) $A_j\text{Send}$ is copied from registers into shared memory and stored as SmA_j , and $B_j\text{Send}$ is from registers to shared memory as SmB_j . 2) The other warps in the same row read SmA_j from shared memory into their registers as $A_j\text{Recv}$ (steps ① and ④ in Figure 6(b), and lines 6 and 13 in Algorithm 2), and the other warps in the same column of the warp grid read SmB_j from shared memory into their registers as $B_j\text{Recv}$ (steps ① and ④ in Figure 6(b), and lines 9 and 15 in Algorithm 2). The data transfer within a warp is the same as in the 1D algorithm (steps ② and ⑤ in Figure 6(b), and lines 7 and 10 in Algorithm 2).

Once all warps have their $A_j\text{Recv}$ (size $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$) and $B_j\text{Recv}$ (size $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$), the two submatrices are multiplied on tensor cores (steps ③ and ⑥ in Figure 6(b), and line 17 in Algorithm 2).

Algorithm 2 2D algorithm by p warps

```

1:  $i \leftarrow \text{warpID}$ 
2: GMem2Reg( $A_i \leftarrow A, B_i \leftarrow B, C_i \leftarrow C$ )
3:  $\_\text{syncthreads}()$ 
4: for  $z = 0$  to  $\sqrt{p}$  do                                 $\triangleright$  The algorithm consists of  $\sqrt{p}$  stages.
5:   if  $i \% \sqrt{p} = z$  then
6:     Reg2SMem( $\text{SmA} \leftarrow \text{ASend}$ )           $\triangleright$  Write  $\text{ASend}$  to shared memory.
7:     Reg2Reg( $\text{ARecv} \leftarrow \text{ASend}$ )            $\triangleright$  Copy  $\text{ASend}$  between registers.
8:   if  $i / \sqrt{p} = z$  then
9:     Reg2SMem( $\text{SmB} \leftarrow \text{BSend}$ )           $\triangleright$  Write  $\text{BSend}$  to shared memory.
10:    Reg2Reg( $\text{BRecv} \leftarrow \text{BSend}$ )            $\triangleright$  Copy  $\text{BSend}$  between registers.
11:    $\_\text{syncthreads}()$ 
12:   if  $i \% \sqrt{p} \neq z$  then
13:     SMem2Reg( $\text{ARecv} \leftarrow \text{SmA}$ )           $\triangleright$  Read  $\text{SmA}$  from shared memory.
14:     if  $i / \sqrt{p} \neq z$  then
15:       SMem2Reg( $\text{BRecv} \leftarrow \text{SmB}$ )           $\triangleright$  Read  $\text{SmB}$  from shared memory.
16:      $\_\text{syncthreads}()$ 
17:      $C_i \leftarrow \text{TensorCoreGEMM}(\text{ARecv}, \text{BRecv})$        $\triangleright$   $\text{ARecv}$  and  $\text{BRecv}$  multiplied by Tensor Core.
18: Reg2GMem( $C_i \leftarrow C$ )
  
```

We now analyze the communication and computational costs. Same as the 1D algorithm, we also assume that the communication

overhead within the same warp can be ignored. Then the total communication volume consists of two components: writing A_i (size $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$ and B_i (size $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$) to shared memory by \sqrt{p} warps, and reading it from shared memory by $\sqrt{p} \times (\sqrt{p}-1)$ warps. Taking s_e as the size of an element, the total communication volume

$$\begin{aligned} V_{cm} &= \left(\sqrt{p} \times \left(\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}} \right) + \sqrt{p} \times (\sqrt{p}-1) \times \left(\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}} \right) \right) \times s_e \\ &\quad + \left(\sqrt{p} \times \left(\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \right) + \sqrt{p} \times (\sqrt{p}-1) \times \left(\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \right) \right) \times s_e \\ &= (mk + kn) \times s_e. \end{aligned} \quad (5)$$

Considering shared memory access latency L_{sm} , bandwidth B_{sm} , and bank conflict factors θ_r and θ_w , the communication cost

$$T_{cm} = L_{sm} + \frac{(mk + nk) \times s_e}{\theta_w \sqrt{p} B_{sm}} + \frac{(\sqrt{p}-1)(mk + nk) \times s_e}{\theta_r \sqrt{p} B_{sm}}. \quad (6)$$

When O_{tc} is the number of arithmetic operations per cycle by per tensor core, the algorithm's computational cost

$$T_{cp} = \frac{\text{flops}(A_j^{\text{Recv}}, B_j^{\text{Recv}})}{O_{tc}} = \frac{2 \times \frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}}{O_{tc}} = \frac{2mnk}{\sqrt[3]{p} O_{tc}}. \quad (7)$$

The algorithm has \sqrt{p} stages, each has one communication phase and p concurrent computations. Then the total execution cost

$$\begin{aligned} T_{all} &= \sqrt{p} \times (T_{cm} + \frac{p}{n_{tc}} \times T_{cp}) \\ &= L_{sm} \sqrt{p} + \frac{(mk + nk) \times s_e}{\theta_w B_{sm}} + \frac{(\sqrt{p}-1)(mk + nk) \times s_e}{\theta_r B_{sm}} + \frac{2mnk}{n_{tc} O_{tc}}. \end{aligned} \quad (8)$$

To provide a more concrete example, suppose in Figure 6(b), four warps ($p = 4$) multiply 8×8 matrices A and B , and $s_e = 8$ in FP64. Through Formula 5, $V_{cm} = 1024$ bytes.

Assuming that the shared memory latency $L_{sm} = 22$ cycles, the bank conflict factors $\theta_r = \theta_w = 1$, and the shared memory bandwidth $B_{sm} = 128$ bytes per cycle, bring in Formula 6, $T_{cm} = 30$ cycles. If the tensor core performs 32 arithmetic operations per cycle and we have 4 tensor cores each SM ($O_{tc} = 32$ and $n_{tc} = 4$), bring in formula 7, $T_{cp} = 4$ cycles. Thus, the total execution cost is $T_{all} = 68$ cycles as Formula 8.

4.5 3D Algorithm

In the 3D algorithm, p warps are divided into a $\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$ warp cube for one multiplication, and warp i holds submatrices A_i (size $\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}}$) and B_i (size $\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$). The warp cube can be viewed as $\sqrt[3]{p}$ warp grids of size $\sqrt[3]{p} \times \sqrt[3]{p}$, with A_i and B_i in the 2D algorithm divided along the k -dimension into $\sqrt[3]{p}$ submatrices accordingly. The warps also work in the SPM model.

The multiplication now has $\sqrt[3]{p}$ stages, each with communication and computation phases. In the z^{th} stage ($0 \leq z < \sqrt[3]{p}$), the communication phase broadcasts the submatrix block A_j^{Send} , which is the A_i held by the z^{th} column of the warp cube, to the other warps in the same row and same layer, and B_j^{Send} , which is the B_i held by the z^{th} row of the warp cube, to the other warps in the same column and same layer.

The communication has two steps: 1) The submatrices A_j^{Send} and B_j^{Send} are copied from registers to shared memory as SmA_j and SmB_j , respectively. 2) The other warps in the same row and layer of the warp cube read SmA_j from shared memory into their registers as A_j^{Recv} (steps ① and ④ in Figure 6(c), and lines 6 and 13 in Algorithm 3), and the other warps in the same column and layer read SmB_j from shared memory into their registers as B_j^{Recv} (steps ① and ④ in Figure 6(c), and lines 9 and 15 in Algorithm 3). The data transfer within a warp is also the same as in the 1D and 2D algorithms (steps ② and ⑤ in Figure 6(c), and lines 7 and 10 in Algorithm 3).

Now all warps have their A_j^{Recv} (size $\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}}$) and B_j^{Recv} (size $\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$) and multiply them on tensor cores (see steps ③ and ⑥ in Figure 6(c), and line 17 in Algorithm 3).

Algorithm 3 3D algorithm by p warps

```

1:  $i \leftarrow$  warpID
2: GMem2Reg( $A_i \leftarrow A$ ,  $B_i \leftarrow B$ ,  $C_i \leftarrow C$ )
3:  $\_\_\text{syncthreads}()$ 
4: for  $z = 0$  to  $\sqrt[3]{p}$  do ▷ The algorithm consists of  $\sqrt[3]{p}$  stages.
5:   if  $i / \sqrt[3]{p} / \sqrt[3]{p} = z$  then
6:     Reg2SMem( $\text{SmA} \leftarrow \text{ASend}$ ) ▷ Write ASend to shared memory.
7:     Reg2Reg(ARecv  $\leftarrow$  ASend) ▷ Copy ASend between registers.
8:   if  $i / \sqrt[3]{p} \% \sqrt[3]{p} = z$  then
9:     Reg2SMem( $\text{SmB} \leftarrow \text{BSend}$ ) ▷ Write BSend to shared memory.
10:    Reg2Reg(BRecv  $\leftarrow$  BSend) ▷ Copy BSend between registers.
11:    $\_\_\text{syncthreads}()$ 
12:   if  $i / \sqrt[3]{p} / \sqrt[3]{p} \neq z$  then
13:     Smem2Reg(ARecv  $\leftarrow \text{SmA}$ ) ▷ Read SmA from shared memory.
14:   if  $i / \sqrt[3]{p} \% \sqrt[3]{p} \neq z$  then
15:     Smem2Reg(BRecv  $\leftarrow \text{SmB}$ ) ▷ Read SmB from shared memory.
16:    $\_\_\text{syncthreads}()$ 
17:    $C_i \leftarrow \text{TensorCoreGEMM}(ARecv, BRecv)$  ▷ ARecv and BRecv multiplied by Tensor Core.
18:   Reg2GMem( $C_i \leftarrow C[i \% \sqrt[3]{p}]$ )
19:    $C \leftarrow C + C[i \% \sqrt[3]{p}]$ 

```

With the two components: writing A_i (size $\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}}$) and B_i (size $\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$) to shared memory by $\sqrt[3]{p}$ warps, and reading them by $\sqrt[3]{p} \times (\sqrt[3]{p}-1)$ warps, the total communication volume

$$\begin{aligned} V_{cm} &= \left(\sqrt[3]{p} \times \left(\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}} \right) + \sqrt[3]{p} \times (\sqrt[3]{p}-1) \times \left(\frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}} \right) \right) \times s_e \\ &\quad + \left(\sqrt[3]{p} \times \left(\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}} \right) + \sqrt[3]{p} \times (\sqrt[3]{p}-1) \times \left(\frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}} \right) \right) \times s_e \\ &= (mk + kn) \times s_e. \end{aligned} \quad (9)$$

Considering shared memory access latency L_{sm} , bandwidth B_{sm} , and bank conflict factors θ_r and θ_w , the communication cost

$$T_{cm} = L_{sm} + \frac{(mk + nk) \times s_e}{\theta_w \sqrt[3]{p} B_{sm}} + \frac{(\sqrt[3]{p}-1)(mk + nk) \times s_e}{\theta_r \sqrt[3]{p} B_{sm}}. \quad (10)$$

The algorithm's computational cost

$$T_{cp} = \frac{\text{flops}(A_j^{\text{Recv}}, B_j^{\text{Recv}})}{O_{tc}} = \frac{2 \times \frac{m}{\sqrt[3]{p}} \times \frac{k}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}}{O_{tc}} = \frac{2mnk}{\sqrt[3]{p} O_{tc}}. \quad (11)$$

The algorithm includes $\sqrt[3]{p}$ stages, each has one communication and p concurrent computation phases, and the total execution cost

$$\begin{aligned} T_{all} &= \sqrt[3]{p} \times (T_{cm} + \frac{p}{n_{tc}} \times T_{cp}) \\ &= L_{sm} \sqrt[3]{p} + \frac{(mk + nk) \times s_e}{\theta_w B_{sm}} + \frac{(\sqrt[3]{p} - 1)(mk + nk) \times s_e}{\theta_r B_{sm}} + \frac{2mnk}{n_{tc} O_{tc}} \end{aligned} \quad (12)$$

In Figure 6(c), four warps ($p = 8$) multiply 8×8 matrices A and B, and $s_e = 8$. Through Formula 9, V_{cm} is 1024 bytes. When $L_{sm} = 22$, $\theta_r = \theta_w = 1$, $B_{sm} = 128$, $n_{tc} = 4$ and $O_{tc} = 32$, we can compute $T_{cm} = 30$ cycles (Formula 10) and $T_{all} = 68$ cycles (Formula 12).

4.6 Sparse Extension: SpMM and SpGEMM

We extend KAMI to support two sparse matrix multiplication operations SpMM and SpGEMM. To well exploit the tensor cores, we save the sparse matrices in smaller dense blocks of user-configurable size, with a default of 16×16 selected to align with various tensor core shapes. Figure 7 illustrates the sparse storage using a 4×4 block as an example. For the 1D algorithm (Figure 7(a)), the blocks are saved row-by-row. For the 2D and 3D algorithms (Figure 7(b)), a multi-level Z-Morton order is implemented to facilitate efficient submatrix indexing, which is similar to the sparse formats proposed by Buluç et al. [43] and Yzelman et al. [219–221].

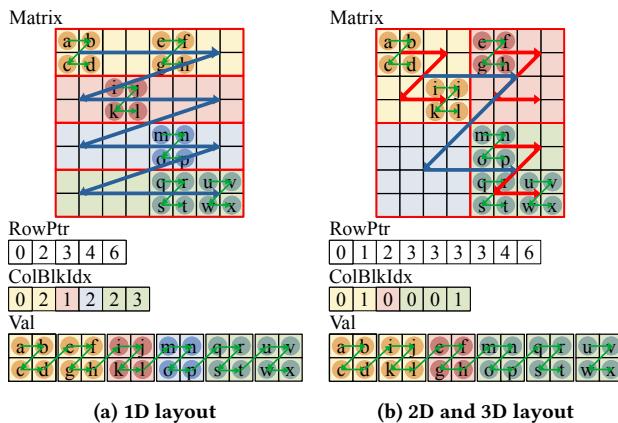


Figure 7: Sparse matrix storage in KAMI.

In the sparse form of KAMI, the organization of warps and the stages remain consistent with those in the dense schemes explained in Sections 4.3–4.5. The entire process also includes communication and computation phases. In the 1D algorithm, each warp processes a distinct sparse row block. In the 2D and 3D algorithms, both A and B are copied in the sparse warp grid or cube. During communication, besides transferring the Val array, it is necessary to transmit the index arrays RowPtr and ColBlkIdx that represent the sparse matrix structure. This data transfer requires allocating additional space in shared memory for supporting the sparsity.

In SpMM, B and C are dense. After all the submatrices are communicated in KAMI, we follow the same compute pattern proposed by Koanantakool et al. [112]. Specifically, the corresponding blocks in B_i are identified by every nonzero block in A_i by traversing the index arrays. Then the intermediate results are computed by tensor cores, and accumulated into the appropriate locations of C_i .

In SpGEMM, matrices A, B and C are all sparse, and a symbolic phase is needed before the numeric computation. The symbolic phase calls a separate kernel, before the CA numeric kernel, to calculate the number of nonzero blocks and allocate the necessary memory space. The symbolic kernel uses a classic sparse accumulator by Gilbert et al. [87]. On top of our 1D, 2D and 3D compute patterns in KAMI, the CA numeric kernel utilizes the indexing method proposed by Hong and Buluç [99] to accumulate the resulting blocks into C_i stored in registers.

4.7 Implementation Details

We elaborate on two core implementation details in KAMI. The first arises from the limited register and shared memory capacities. For example, storing three 128×128 matrices in FP64 (two 32-bit registers per element) with eight warps (i.e., 256 threads) requires $3 \times 128 \times 128 \times 2 \div 256 = 384$ registers per thread, exceeding the hardware limit of 255. To address this, KAMI slices the matrices along the k dimension, storing only a portion of A and B in registers, while offloading the inactive sub-matrices to shared memory. The slicing ratio—i.e., the fraction of data kept in registers versus shared memory—is a tunable parameter selected based on empirical performance and varies by matrix size. In our implementation, each k -slice has a dimension of 16 to align with the MMA unit granularity, thereby minimizing hardware fragmentation. This cooperation applies to KAMI-1D/2D/3D, though optimal ratios differ with data layouts. Slicing overhead is negligible; performance differences arise mainly from shared-memory latency. Section 5.2.5 evaluates different ratios and annotates when register demand exceeds hardware limits, motivating fallback to shared memory.

The second detail concerns the overlap of communication and computation, analogous to MPI_Isend() and MPI_Irecv(). KAMI does not enforce explicit overlap strategies, as the CUDA warp scheduling and underlying GPU hardware should be effective at interleaving data transfer and computation. Section 5.6.2 validates this by showing that actual clock cycles closely follow the theoretical model when considering communication-computation concurrency.

5 Experimental Results

5.1 Experimental Setup

KAMI is evaluated on four GPUs from NVIDIA, AMD, and Intel, with implementations using CUDA, HIP, and SYCL, respectively. The device specifications are shown in Table 3, and the programming methods are listed in Table 4.

Vendor	NVIDIA		AMD	Intel
Specifications	GH200	RTX 5090	7900 XTX	Max 1100
Boost clock (MHz)	1980	2655	2498	1550
#Banks \times bank width (Bytes)	32 \times 4	32 \times 4	32 \times 4	16 \times 4
#SMs \times #tensor cores/SM	132 \times 4	170 \times 4	96 \times 2	448 \times 1
Peak FP16 tensor (TFLOPS)	990	462	123	22
Peak FP64 tensor (TFLOPS)	67	N/A	N/A	N/A

Table 3: Four GPUs from NVIDIA, AMD and Intel.

We evaluate KAMI across a diverse set of workloads. For square GEMM, we compare against cuBLASDx v0.2.0 [156] and CUTLASS

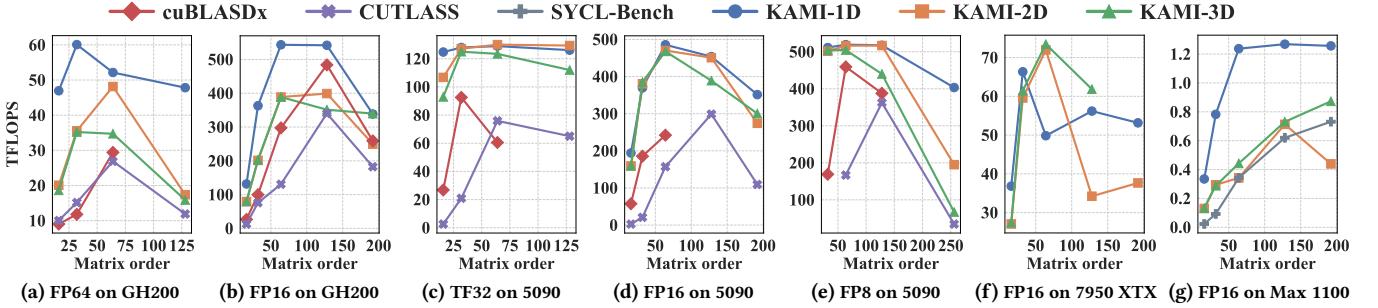


Figure 8: Block-Level GEMM Performance across GPU Architectures.

GPU Vendor	NVIDIA	AMD	Intel
Programming API	CUDA	HIP	SYCL
Local storage	Register	fragment	joint_matrix
Communication space	Shared memory	Shared memory	Local memory
Tensor core func.	mma	mma_sync	joint_matrix_mad
Instruction shape	m16n8k8 (FP64) m16n8k16 (FP16)	m16n16k16 (FP16)	m16n16k16 (FP16)

Table 4: Programming API supported by KAMI

v3.8.0 [157] on NVIDIA GH200 in FP64 and FP16, and 5090 in TF32, FP16 and FP8. KAMI’s FP16 performance on AMD 7900 XTX and Intel Max 1100 (vs. SYCL-Bench [120]) is also reported. Matrices with orders 16, 32, 64, and 128 are used for FP64, TF32, FP16 and FP8, with an additional 192 for FFP16 and 256 for FP8. We also analyze the effect of varying block sizes and shared memory temporary saving in FP16 on 5090.

For low-rank GEMM, we test KAMI, cuBLASDx and CUTLASS on GH200 in FP16, using $k = 16$ or 32 , with m and n aligning with the square GEMM orders.

Batched GEMM is evaluated on GH200 in FP64, compared with cuBLAS v12.8 [155] and MAGMA v2.9 [149]. Matrix orders follow the square GEMM setup, with batch sizes of 1000 and 10000.

Block-level KAMI is further evaluated on SpMM and SpGEMM on GH200 in FP16, using five sparse matrices (50% random sparsity) of the same order as the square GEMM test. All block-level results (square, low-rank, and sparse) are averaged over 1000 runs with 16,384 blocks launched simultaneously per run.

Finally, we provide a theoretical analysis of KAMI, including register usage and execution cycles.

5.2 Block-Level square GEMM

5.2.1 KAMI on NVIDIA GPU. We evaluate block-level GEMM performance of KAMI and cuBLASDx on GH200 and 5090 GPUs for square matrices. Figures 8(a), (b), and (d) present FP64/FP16 results on GH200 and FP16 on 5090.

For FP64 on GH200, KAMI-1D/2D/3D outperform cuBLASDx and CUTLASS by 4.02x/2.29x/2.08x and 3.65x/1.90x/1.70x on average (up to 5.20x/3.02x/2.99x and 4.68x/2.34x/2.32x). For FP16, KAMI-1D/2D/3D achieve 2.56x/1.62x/1.67x and 4.54x/2.88x/2.95x speedups on average (up to 4.93x/2.98x/2.98x and 10.31x/6.23x/6.23x) on GH200, and, 2.46x/2.25x/2.24x and 19.98x/17.25x/17.01x (up to 3.38x/2.77x/2.76x and 74.36x/60.99x/60.66x) on 5090. For TF32 on 5090, KAMI-1D/2D/3D outperform cuBLASDx and CUTLASS by 2.72x/2.50x/2.28x and 14.38x/12.66x/11.22x on average (up to 4.65x/3.98x/3.46x and 47.76x/40.87x/35.56x). For FP8 on

5090, KAMI-1D/2D/3D outperform cuBLASDx and CUTLASS by 1.83x/1.81x/1.74x and 5.40x/3.39x/2.06x on average (up to 3.03x/2.97x/2.98x and 11.67x/5.64x/3.02x). KAMI supports larger matrices with lightweight shared memory use compared with cuBLASDx.

KAMI-1D generally outperforms KAMI-2D/3D. On GH200, KAMI-3D can even underperform compared to cuBLASDx, probably due to more complex control flows, with additional branches, loops, and synchronizations. Profiling a 128×128 FP16 kernel shows KAMI-2D/3D incur 45.32%/152.38% more nop instructions than KAMI-1D, making KAMI-1D more suitable for current single-GPU use.

5.2.2 KAMI on AMD GPU. As no block-level library exists on AMD, we report only KAMI’s performance in Figure 8(f). When the matrix order exceeds 48, KAMI-1D’s performance drops, which occurs later for KAMI-2D/3D.

5.2.3 KAMI on Intel GPU. Figure 8(g) shows KAMI’s performance on the Intel Max 1100 GPU, compared with SYCL-Bench. KAMI-1D/2D/3D outperform SYCL-Bench by 4.97x/2.20x/2.00x on average, with peak speedups of 14.48x/5.63x/5.71x, respectively.

5.2.4 Block Size Effects. Figure 9 shows the GEMM performance of two 64×64 matrices by KAMI-1D, KAMI-2D, and KAMI-3D on 5090, with peak performances of 469.80, 470.57, and 449.07 TFLOPS.

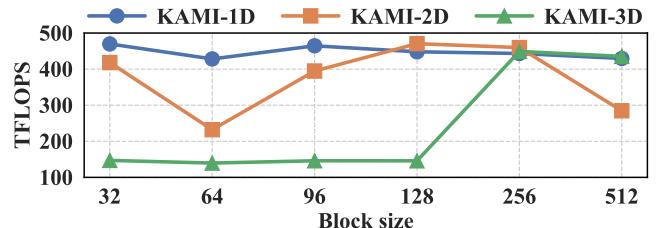


Figure 9: Impact of block size in FP16 on 5090.

KAMI-1D delivers consistently high performance across a wide range of block sizes. In contrast, KAMI-2D, with its 2D warp configuration, achieves only 54.22% of KAMI-1D’s performance at block size 64. KAMI-3D is even more sensitive, performing well only when the block size exceeds 256.

Thus, KAMI-1D is robust under tight block size constraints, while KAMI-2D/3D is preferable when larger block sizes are available. This also explains why KAMI-1D performs better than KAMI-2D/3D

under the current architectures with limited number of thread blocks.

5.2.5 Shared Memory and Register Cooperation. To validate the effectiveness of temporary saving in shared memory as mentioned in Section 4.7, we illustrate how the performance of GEMM (FP16) varies with shared memory usage in Figure 10.

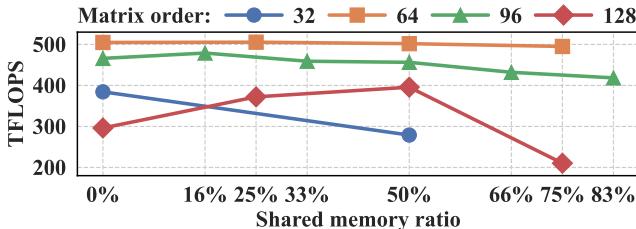


Figure 10: Impact of shared memory ratio on block-level.

For small matrices (32-64), registers alone suffice to store all necessary data, and using shared memory degrades performance. As matrix order increases, registers become insufficient. Temporary saving data in shared memory improves performance. At matrix order 128, performance peaks at 1.34x when 50% is temporarily saved in shared memory. However, excessive use of shared memory leads to a slowdown due to its higher cost. For example, performance drops to 0.71x when 75% of the data is in shared memory.

Results show that the optimal register–shared memory ratio is scale-dependent: registers suffice for small matrices, while moderate shared memory use benefits medium sizes; excessive use degrades performance. Accordingly, we preset ratios in our implementation and allow user tuning to balance generality and specialization.

5.3 Low-Rank GEMM

We compare KAMI and cuBLASDx on low-rank GEMM for $k = 16$ (Figure 11(a)) and $k = 32$ (Figure 11(b)) on GH200 in FP16.

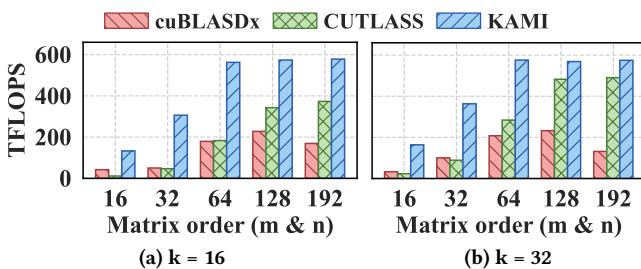


Figure 11: Low-rank GEMM in FP16 on GH200.

KAMI consistently outperforms cuBLASDx and CUTLASS, achieving average speedups of 3.66x, 4.89x (up to 6.11x, 11.61x) for $k = 16$ and 3.65x, 3.09x for $k = 32$ (up to 5.03x, 7.00x).

KAMI exhibits more pronounced advantages in low-rank GEMM than in square matrix GEMM, mainly due to differing memory access strategies. Traditional kernels, as in cuBLASDx/CUTLASS, load data into shared memory and then into registers, enhancing locality but offering limited benefit when k is small. In contrast, KAMI loads data directly into registers and uses shared memory for communication, better matching low-rank GEMM patterns.

5.4 Batched GEMM

KAMI’s batched interface is consistent with cuBLAS and MAGMA, and supports various matrix orders in a batch. We compare them in a uniform order to focus on the GEMM efficiency in Figure 12.

KAMI achieves significant speedups, with average speedups of 31.60x and 340.37x for batch sizes of 1000, and 10.23x and 96.17x for batch sizes of 10000, compared with MAGMA and cuBLAS. We attribute this to the limited optimization of small-scale GEMM operations in both MAGMA and cuBLAS.

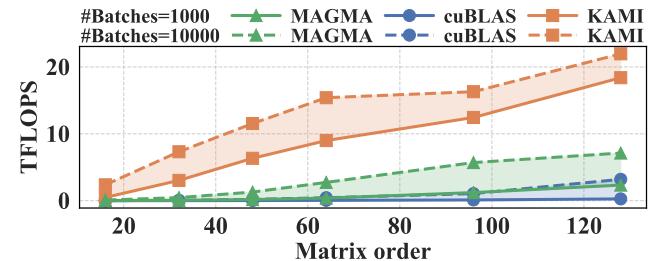


Figure 12: Comparison of batched GEMM in FP64 on GH200.

We also note that the absolute performance in batched GEMM is lower than that in the standalone GEMM case (Figure 8), which is expected. In the batched setting, each small matrix in the batch is loaded separately from global memory, incurring higher memory traffic per FLOP compared to monolithic GEMM where reuse and shared memory optimization are more effective. This memory-bound nature of batched GEMM constrains throughput despite kernel efficiency.

5.5 SpMM and SpGEMM

Figure 13 presents the performance of SpMM and SpGEMM in FP16 on the GH200 platform.

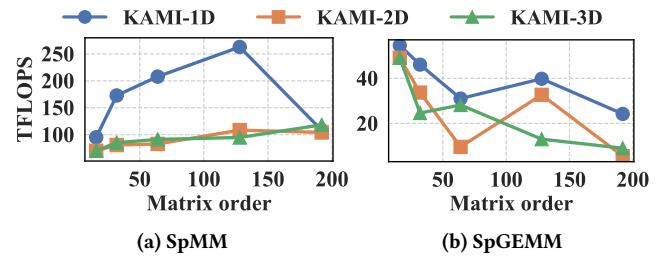


Figure 13: SpMM and SpGEMM in FP16 on GH200.

The performance trend of SpMM closely resembles that of GEMM, as the input matrices B and C are dense. In this case, the only sparsity lies in matrix A, which allows for highly regular computations and memory accesses within the dense blocks of B and C. As a result, SpMM benefits from efficient coalesced memory accesses, reduced indexing overhead, and a computational pattern similar to dense GEMM, which explains its relatively high performance.

In contrast, SpGEMM introduces significantly more complex indexing and results in less predictable memory access patterns due to different sparse structures in both input matrices. These irregularities lead to distinct performance behaviors and reduced throughput of SpGEMM.

5.6 Theoretical Analysis

5.6.1 Register Allocation. To validate the theoretical analysis presented in Section 4, we compare the theoretical register usage of KAMI with the actual allocation measured during compilation. We test KAMI-1D (4 warps), KAMI-2D (4 warps) and KAMI-3D (8 warps), with C fixed at 64×32 and A, B varying with k (Figure 14).

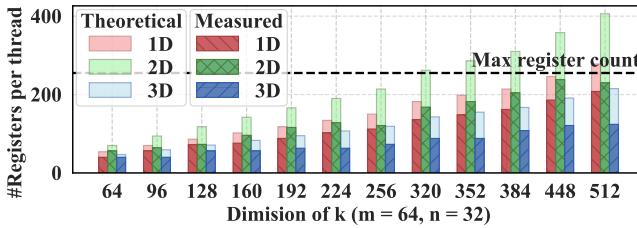


Figure 14: The register usage of KAMI in FP16.

Results show that actual register usage is lower than theoretical predictions, reaching 76.86% for KAMI-1D, 73.14% for KAMI-2D, and 65.67% for KAMI-3D. The deviation is likely primarily attributable to compiler optimizations, such as shortening variable lifetimes and optimizing register reuse.

We also compare the overall on-chip memory usage to cuBLASDx and CUTLASS. For a 64×64 GEMM in FP16, KAMI-1D/2D/3D use 62/80/55 registers per thread—between cuBLASDx’s 40 and CUTLASS’s 96—and only 2–8 KB of shared memory per block, significantly less than cuBLASDx’s 27 KB and CUTLASS’s 65 KB.

5.6.2 Cycles Breakdown. We break down execution cycles into communication and computation on GH200 and 5090, comparing results with the theoretical model in Section 4. Cycle counts are measured using the `clock()` function with a single CUDA thread block (4 warps of KAMI-1D/2D and 8 warps of KAMI-3D). Figure 15 shows the results.

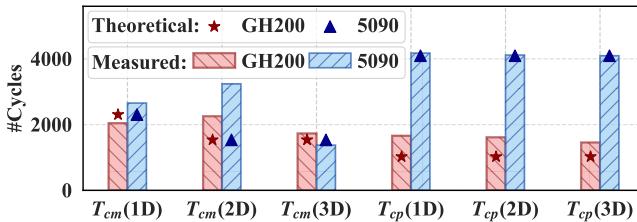


Figure 15: The theoretical and measured cycles in FP16.

Overall, the experimental results are largely consistent with the theoretical model, aside from some discrepancies in a few cases. For example, on GH200, the theoretical cycles of computation are consistently lower than the measured values. We attribute this to the tested 62% maximum MMA instruction execution efficiency on the Hopper architecture [136].

6 Related Work

GEMM has been accelerated on a variety of CPUs, such as x86 [139, 200] and ARM [195, 198, 199, 202, 208–210], GPUs [49, 79, 106, 118, 121, 124, 127, 148, 181, 193, 224], TPUs [100], DSPs [217], and distributed platforms [6, 24, 31, 119, 129]. Representative open-source libraries include ATLAS [58], GotoBLAS [88, 89, 137, 170],

OpenBLAS [223], BLIS [188–192, 206], LAPACK [63, 111], ScaLAPACK [56], MAGMA [149], SLATE [84], Iris [140], CUTLASS [157], and Ozaki [145–147, 159, 185]. Beyond dense linear algebra, GEMM has also been exploited in sparse computations, such as accelerating SpMV [134], BFS [153], and sparse LU factorization [82, 197].

Besides manually tuned kernels, code generation techniques [33, 37, 151, 196] can bring better performance portability by selecting methods [66, 107, 161] and block sizes [36, 50, 81, 116, 162]. Some other factors in parallel GEMM, such as scheduling [30, 32, 45], numerical stability [16, 34, 71], fault tolerant [38, 39, 152], low precision [1, 96, 128], energy efficiency [11, 52, 68], multiplying tall-and-skinny matrix [53], as well as tensor operations [123, 125], were considered as well.

Low-rank can be very useful in many dense [110, 133, 138] and sparse [8, 46] problems. Such scenarios highly require multiplying small matrices [98, 142, 169] and its batched implementations [10, 76, 86, 117, 225]. Also, numerous studies consider sparsity in matrix multiplication. Among them, SpMM [4, 7, 83, 95, 101, 126, 168, 184, 222] and SpGEMM [3, 55, 62, 99, 130, 131, 154, 163, 204, 205] have been the most extensively studied. In this work, our KAMI shows promising performance on low-rank and sparse matrix multiplications.

Communication is often the major bottleneck of distributed algorithms [5, 12, 41, 42, 44, 54, 64, 80, 115, 186, 187]. To address the problem, a series of CA algorithms were proposed [22, 85, 94, 175] and analyzed theoretically [6, 18, 105]. The CA methods are highly effective in linear algebra, particularly for dense problems including GEMM [67, 102, 119, 173] and its Strassen’s optimization [20, 23, 24, 33, 129], matrix factorization [9, 14, 15, 19, 19, 21, 27, 69, 70, 74, 75, 91–93, 104, 110, 176, 178], eigenvalue problems [25, 26, 171], and tensor operations [5, 177]. As for sparse linear algebra, the CA research mainly focused on sparse matrix multiplication [17, 28, 40, 72, 103, 109, 112, 167], sparse triangular solve [8, 164, 201], iterative solvers [29, 48, 73, 141, 179, 207] and direct solvers [108, 165, 166]. The CA methods have also been extended to write-avoiding methods [47], FFT [59, 114], AI operations [60, 144, 183, 212–216], graph processing [35, 61, 172, 174], stencil computation [203], as well as N-body problems [2, 77, 78, 113]. To our knowledge, KAMI for the first time optimizes and theoretically analyzes the CA GEMM within a single GPU.

7 Conclusion

In this paper, we have proposed KAMI, a set of 1D, 2D, and 3D CA GEMM algorithms within a single GPU. KAMI improved the utilization of high-speed registers for local storage and tensor cores for computation, and used shared memory for communication. A theoretical analysis in clock cycles was also provided. In the experiments, KAMI achieved significant speedups over existing work on GPUs from NVIDIA, AMD and Intel.

Acknowledgments

We greatly appreciate the invaluable comments of all reviewers. Weifeng Liu is the corresponding author of this paper. This work was partially supported by the National Natural Science Foundation of China (Grant No. U23A20301, No. 62372467, and No. 62402525).

References

- [1] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, et al. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 344–369.
- [2] Mustafa Abduljabbar, George S. Markomanolis, Huda Ibeid, Rio Yokota, and David Keyes. 2017. Communication Reducing Algorithms for Distributed Hierarchical N-Body Problems with Boundary Distributions. In *International Conference on High Performance Computing (ISC)*.
- [3] Kadir Akbudak and Cevdet Aykanat. 2017. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (2017), 2258–2271.
- [4] Hasan Metin Aktulgul, Aydin Buluc, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [5] Hussam Al Daas, Grey Ballard, Laura Grigori, Suraj Kumar, and Kathryn Rouse. 2024. Communication lower bounds and optimal algorithms for multiple tensor-times-matrix computation. *SIAM J. Matrix Anal. Appl.* 45, 1 (2024), 450–477.
- [6] Hussam Al Daas, Grey Ballard, Laura Grigori, Suraj, and Kathryn Rouse. 2023. Parallel Memory-Independent Communication Bounds for SYRK. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [7] José I Aliaga, Hartwig Anzt, Enrique S Quintana-Ortí, and Andrés E Tomás. 2023. Sparse matrix-vector and matrix-multivector products for the truncated SVD on graphics processors. *Concurrency and Computation: Practice and Experience* 35, 28 (2023), e7871.
- [8] Patrick Amestoy, Olivier Boiteau, Alfredo Buttari, Matthieu Gerest, Fabienne Jézquel, Jean-Yves L'Excellent, and Theo Mary. 2024. Communication avoiding block low-rank parallel multifrontal triangular solve with many right-hand sides. *SIAM J. Matrix Anal. Appl.* 45, 1 (2024), 148–166.
- [9] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. 2011. Communication-avoiding QR decomposition for GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [10] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S Quintana-Ortí. 2017. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. In *International Conference on Parallel Processing (ICPP)*.
- [11] Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2015. Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5096–5113.
- [12] Hartwig Anzt, Axel Huebl, and Xiaoye S. Li. 2024. Then and Now: Improving Software Portability, Productivity, and 100x Performance. *Computing in Science & Engineering* 26, 1 (2024), 61–70.
- [13] Mochamad Asri, Dhairyu Malhotra, Jiajun Wang, George Biros, Lizy K. John, and Andreas Gerstlauer. 2021. Hardware Accelerator Integration Tradeoffs for High-Performance Computing: A Case Study of GEMM Acceleration in N-Body Methods. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 2035–2048.
- [14] Marc Baboulin, Simplex Donfack, Jack Dongarra, Laura Grigori, Adrien Rémy, and Stanimire Tomov. 2012. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *Procedia Computer Science* 9 (2012), 17–26.
- [15] Grey Ballard, Dulcemeia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Itron Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. 2014. Communication-avoiding symmetric-indefinite factorization. *SIAM J. Matrix Anal. Appl.* 35, 4 (2014), 1364–1406.
- [16] Grey Ballard, Austin R Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. 2016. Improving the numerical stability of fast matrix multiplication. *SIAM J. Matrix Anal. Appl.* 37, 4 (2016), 1382–1418.
- [17] Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication optimal parallel multiplication of sparse random matrices. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [18] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* 23 (2014), 1–155.
- [19] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, and Nicholas Knight. 2018. A 3d parallel algorithm for qr decomposition. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [20] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [21] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2010. Communication-optimal Parallel and Sequential Cholesky Decomposition. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3495–3523.
- [22] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901.
- [23] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2012. Graph expansion and communication costs of fast matrix multiplication. *J. ACM* 59, 6 (2012), 1–23.
- [24] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2014. Communication costs of Strassen's matrix multiplication. *Commun. ACM* 57, 2 (2014), 107–114.
- [25] Grey Ballard, James Demmel, and Nicholas Knight. 2012. Communication avoiding successive band reduction. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [26] Grey Ballard, James Demmel, and Nicholas Knight. 2015. Avoiding communication in successive band reduction. *ACM Transactions on Parallel Computing* 1, 2 (2015), 1–37.
- [27] Grey Ballard, James Demmel, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. 2013. Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [28] Grey Ballard, Christopher Siebert, and Jonathan Hu. 2016. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing* 38, 3 (2016), C203–C231.
- [29] Protomou Basu, Anand Venkat, Mary Hall, Samuel Williams, Brian Van Straalen, and Leonid Oliker. 2013. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*.
- [30] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. 2001. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1033–1051.
- [31] Olivier Beaumont, Lionel Eyraud-Dubois, and Thomas Lambert. 2016. Cuboid Partitioning for Parallel Matrix Multiplication on Heterogeneous Platforms. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
- [32] Olivier Beaumont and Loris Marchal. 2014. Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.
- [33] Austin R Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [34] Jérémie Berthomieu, Stef Graillat, Dimitri Lesnoff, and Theo Mary. 2025. Multi-word matrix multiplication over large finite fields in floating-point arithmetic. *HAL preprint hal-04917201* (2025).
- [35] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefer, and Edgar Solomonik. 2020. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [36] Paolo Bientinesi, John A Gunnels, Margaret E Myers, Enrique S Quintana-Ortí, and Robert A van de Geijn. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Software* 31, 1 (2005), 1–26.
- [37] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ACM International Conference on Supercomputing (ICS)*.
- [38] Noam Birnbaum, Roy Nissim, and Oded Schwartz. 2020. Fault Tolerance with High Performance for Fast Matrix Multiplication. In *The SIAM Workshop on Combinatorial Scientific Computing (CSC)*.
- [39] Noam Birnbaum and Oded Schwartz. 2018. Fault tolerant resource efficient matrix multiplication. In *The SIAM Workshop on Combinatorial Scientific Computing (CSC)*.
- [40] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. 2024. Two-face: Combining collective and one-sided communication for efficient distributed spmm. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [41] Benjamin Brock, Aydin Buluç, and Katherine Yelick. 2024. RDMA-Based Algorithms for Sparse Matrix Multiplication on GPUs. In *ACM International Conference on Supercomputing (ICS)*.
- [42] Benjamin Brock, Robert Cohn, Suyash Bakshi, Tuomas Karna, Jeongnim Kim, Mateusz Nowak, Łukasz undefinedlusarczyk, Kacper Stefanski, and Timothy G. Mattson. 2024. Distributed Ranges: A Model for Distributed Data Structures, Algorithms, and Views. In *ACM International Conference on Supercomputing (ICS)*.
- [43] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [44] Kirk W Cameron and Rong Ge. 2004. Predicting and evaluating distributed communication performance. In *The ACM/IEEE Conference on Supercomputing (SC)*.

- [45] Qinglei Cao, Thomas Herault, Aurelien Bouteiller, Joseph Schuchart, and George Bosilca. 2024. Evaluating PaRSEC Through Matrix Computations in Scientific Applications. In *Asynchronous Many-Task Systems and Applications (WAMTA)*.
- [46] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. 2021. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [47] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. 2016. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [48] Erin Carson, Nicholas Knight, and James Demmel. 2013. Avoiding communication in nonsymmetric Lanczos-based Krylov subspace methods. *SIAM Journal on Scientific Computing* 35, 5 (2013), S42–S61.
- [49] Noel Chalmers, Jakub Kurzak, Damon McDougall, and Paul Bauman. 2023. Optimizing high-performance lmpack for exascale accelerated architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [50] Erni Chan, Field G Van Zee, Paolo Bientinesi, Enrique S Quintana-Orti, Gregorio Quintana-Orti, and Robert Van de Geijn. 2008. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [51] Lorenzo Chelini, Henrik Barthels, Paolo Bientinesi, Marcin Copik, Tobias Grosser, and Daniele G Spampinato. 2022. MOM: Matrix Operations in MLIR. *arXiv preprint arXiv:2208.10391* (2022).
- [52] Jieyang Chen, Li Tan, Panruo Wu, Dingwen Tao, Hongbo Li, Xin Liang, Si-huan Li, Rong Ge, Laxmi Bhuyan, and Zizhong Chen. 2016. GreenLA: Green Linear Algebra Software for GPU-accelerated Heterogeneous Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [53] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *ACM International Conference on Supercomputing (ICS)*.
- [54] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D Owens. 2022. Scalable irregular parallelism with GPUs: Getting CPUs out of the way. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [55] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *International Conference on Parallel Processing (ICPP)*.
- [56] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. 1996. ScALAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance. *Computer Physics Communications* 97, 1–2 (1996), 1–15.
- [57] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023), 9–17.
- [58] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35.
- [59] Kenneth Czechowski, Casey Battaglino, Chris McClanahan, Kartik Iyer, P-K Yeung, and Richard Vuduc. 2012. On the communication complexity of 3D FFTs and its implications for exascale. In *ACM International Conference on Supercomputing (ICS)*.
- [60] Swapnil Das, James Demmel, Karon Fountoulakis, Laura Grigori, Michael W Mahoney, and Shenghao Yang. 2021. Parallel and communication avoiding least angle regression. *SIAM Journal on Scientific Computing* 43, 2 (2021), C154–C176.
- [61] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.
- [62] Gunduz Vehbi Demirci and Cevdet Aykanat. 2020. Cartesian partitioning models for 2d and 3d parallel spgemm algorithms. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2763–2775.
- [63] James Demmel. 1991. LAPACK: A portable linear algebra library for high-performance computers. *Concurrency: Practice and Experience* 3, 6 (1991), 655–666.
- [64] Jim Demmel. 2011. Rethinking algorithms for future architectures: Communication-avoiding algorithms. In *IEEE Hot Chips Symposium (HCS)*.
- [65] Jim Demmel. 2012. Communication avoiding algorithms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [66] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R Clint Whaley, and Katherine Yelick. 2005. Self-adapting linear algebra algorithms and software. *Proc. IEEE* 93, 2 (2005), 293–312.
- [67] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [68] James Demmel, Andrew Gearhart, Benjamin Lipshitz, and Oded Schwartz. 2013. Perfect strong scaling using no additional energy. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [69] James Demmel, Laura Grigori, Ming Gu, and Hua Xiang. 2015. Communication avoiding rank revealing QR factorization with column pivoting. *SIAM J. Matrix Anal. Appl.* 36, 1 (2015), 55–89.
- [70] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239.
- [71] James Demmel and Nicholas J Higham. 1992. Stability of block algorithms with fast level-3 BLAS. *ACM Trans. Math. Software* 18, 3 (1992), 274–291.
- [72] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [73] Aditya Devarakonda, Karon Fountoulakis, James Demmel, and Michael W Mahoney. 2019. Avoiding communication in primal and dual block coordinate descent methods. *SIAM Journal on Scientific Computing* 41, 1 (2019), C1–C27.
- [74] Simplice Donfack, Laura Grigori, and Alok Kumar Gupta. 2010. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [75] Simplice Donfack, Laura Grigori, and Amal Khabou. 2012. Avoiding communication through a multilevel LU factorization. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
- [76] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science* 108 (2017), 495–504.
- [77] Michael Driscoll, Evangelos Georganas, Penporn Koanantakool, Edgar Solomonik, and Katherine Yelick. 2013. A communication-optimal n-body algorithm for direct interactions. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [78] Marquita Ellis, Aydin Buluç, and Katherine Yelick. 2021. Scaling generalized n-body problems, a case study from genomics. In *International Conference on Parallel Processing (ICPP)*.
- [79] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. 2021. Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [80] Oliver Fortmeier, H Martin Bücker, BO Fagginger Auer, and Rob H Bisseling. 2013. A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications. *Parallel Comput.* 39, 8 (2013), 319–335.
- [81] Jeremy D Frens and David S Wise. 1997. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [82] Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhai Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, Zhou Jin, and Weifeng Liu. 2023. PangluU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [83] Daichi Fujiki, Niladri Chatterjee, Donghyuk Lee, and Mike O'Connor. 2019. Near-memory data transformation for efficient sparse matrix multi-vector multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [84] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a modern distributed and accelerated linear algebra library. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [85] Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.
- [86] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluç. 2018. Integrated model, batch, and domain parallelism in training neural networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [87] John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse Matrices in MATLAB: Design and Implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.
- [88] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Software* 34, 3 (2008), 1–25.
- [89] Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Software* 35, 1 (2008), 1–14.
- [90] Laura Grigori, Sébastien Cayrols, and James W Demmel. 2018. Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting. *SIAM Journal on Scientific Computing* 40, 2 (2018), C181–C209.

- [91] Laura Grigori, James Demmel, and Hua Xiang. 2008. Communication avoiding Gaussian elimination. In *The ACM/IEEE Conference on Supercomputing (SC)*.
- [92] Laura Grigori, James Demmel, and Hua Xiang. 2011. CALU: a communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. Appl.* 32, 4 (2011), 1317–1350.
- [93] Laura Grigori, Matthias Jacquelin, and Amal Khabou. 2014. Performance predictions of multilevel communication optimal LU and QR factorizations on hierarchical platforms. In *International Conference on High Performance Computing (ISC)*.
- [94] Laura Grigori, Bernard Philippe, Ahmed H. Sameh, Damien Tromeur-Dervout, and Marián Vajtersic. 2008. Parallel matrix algorithms and applications. *Parallel Comput.* 34, 6–8 (2008), 293–295.
- [95] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [96] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [97] J-Fr Hake and Willi Homberg. 1990. The impact of memory organization on the performance of matrix multiplication. In *The ACM/IEEE conference on Supercomputing (SC)*.
- [98] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [99] Yuxi Hong and Aydin Buluç. 2024. A sparsity-aware distributed-memory algorithm for sparse-sparse matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [100] Kuan-Chieh Hsu and Hung-Wei Tseng. 2021. Accelerating applications using edge tensor processing units. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [101] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [102] Hua Huang and Edmond Chow. 2022. CA3DMM: a new algorithm based on a unified view of parallel matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [103] Md Taufique Hussain, Oguz Selvitopi, Aydin Buluç, and Ariful Azad. 2021. Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [104] Edward Hutter and Edgar Solomonik. 2019. Communication-avoiding Cholesky-QR2 for rectangular matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [105] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1017–1026.
- [106] Abhinav Jangda and Mohit Yadav. 2024. Fast kronecker matrix-matrix multiplication on gpus. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [107] Changhao Jiang and Marc Snir. 2005. Automatic tuning matrix multiplication performance on graphics hardware. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- [108] Oguz Kaya, Ramakrishnan Kannan, and Grey Ballard. 2018. Partitioning and communication strategies for sparse non-negative matrix factorization. In *International Conference on Parallel Processing (ICPP)*.
- [109] Enver Kayaaslan, Cevdet Aykanat, and Bora Uçar. 2018. 1.5 D parallel sparse matrix-vector multiply. *SIAM Journal on Scientific Computing* 40, 1 (2018), C25–C46.
- [110] Amal Khabou, James Demmel, Laura Grigori, and Ming Gu. 2013. LU factorization with panel rank revealing pivoting and its communication avoiding version. *SIAM J. Matrix Anal. Appl.* 34, 3 (2013), 1401–1429.
- [111] Kyungjoo Kim, Timothy B Costa, Mehmet Deveci, Andrew M Bradley, Simon D Hammond, Murat E Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. 2017. Designing vector-friendly compact BLAS and LAPACK kernels. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [112] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. 2016. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [113] Penporn Koanantakool and Katherine Yelick. 2014. A computation-and communication-optimal parallel direct 3-body algorithm. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [114] Thomas Koopman and Rob H Bisseling. 2023. Minimizing communication in the multidimensional FFT. *SIAM Journal on Scientific Computing* 45, 6 (2023), C330–C347.
- [115] Suraj Kumar, Lionel Eyraud-Dubois, and Sriram Krishnamoorthy. 2019. Performance Models for Data Transfers: A Case Study with Molecular Chemistry Kernels. In *International Conference on Parallel Processing (ICPP)*.
- [116] HT Kung, Vikas Natesh, and Andrew Sabot. 2021. Cake: matrix multiplication using constant-bandwidth blocks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [117] Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra. 2016. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2016), 2036–2048.
- [118] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2045–2057.
- [119] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Rafał Solcā, and Torsten Hoefer. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [120] Sohan Lal, Aksel Alpay, Philip Salzmann, Biagio Cosenza, Alexander Hirsch, Nicolai Stawinoja, Peter Thoman, Thomas Fahringer, and Vincent Heuveline. 2020. SYCL-bench: a versatile cross-platform benchmark suite for heterogeneous computing. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
- [121] E Scott Larsen and David McAllister. 2001. Fast matrix multiplies using graphics hardware. In *The ACM/IEEE Conference on Supercomputing (SC)*.
- [122] Ang Li, Weifeng Liu, Mads R.B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and Analyzing the Real Impact of Modern On-Package Memory on HPC Scientific Kernels. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [123] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [124] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In *ACM International Conference on Supercomputing (ICS)*.
- [125] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: hierarchical storage of sparse tensors. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*.
- [126] Shigang Li, Kazuki Osawa, and Torsten Hoefer. 2022. Efficient quantized sparse matrix operations on tensor cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [127] Xiuhong Li, Yun Liang, Shenghen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [128] Xiaoye S Li, James Demmel, David H Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y Kang, Anil Kapur, Michael C Martin, et al. 2002. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Software* 28, 2 (2002), 152–205.
- [129] Benjamin Lipschitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel Strassen: Implementation and performance. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.
- [130] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2018. Register-based implementation of the sparse general matrix-matrix multiplication on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [131] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85 (2015), 47–61.
- [132] Francisco López, Lars Karlsson, and Paolo Bientinesi. 2023. FLOPs as a Discriminant for Dense Linear Algebra Algorithms. In *International Conference on Parallel Processing (ICPP)*.
- [133] Hatem Ltaief, Jesse Cranney, Damien Gratadour, Yuxi Hong, Laurent Gatineau, and David Keyes. 2021. Meeting the real-time challenges of ground-based telescopes using low-rank matrix computations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [134] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [135] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. 2024. AmgT: Algebraic Multigrid Solver on Tensor Cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

- [136] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the nvidia hopper gpu architecture. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [137] Bryan Marker, Field G Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A Van De Geijn. 2007. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
- [138] Théo Mary, Ichitaro Yamazaki, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [139] John D McCalpin. 2018. HPL and DGEMM performance variability on the Xeon Platinum 8160 processor. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [140] Narasinga Rao Miniskar, Mohammad Alaul Haque Monil, Pedro Valero-Lara, Frank Liu, and Jeffrey S Vetter. 2022. Iris-blas: Towards a performance portable and heterogeneous blas library. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*.
- [141] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. 2009. Minimizing communication in sparse matrix solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [142] Yoav Moran and Oded Schwartz. 2023. Multiplying 2×2 Sub-Blocks Using 4 Multiplications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [143] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company.
- [144] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluç. 2024. Sparsity-Aware Communication for Distributed Graph Neural Network Training. In *International Conference on Parallel Processing (ICPP)*.
- [145] Daichi Mukunoki, Takeshi Ogita, and Katsuhisa Ozaki. 2019. Reproducible BLAS routines with tunable accuracy using ozaki scheme for many-core architectures. In *International Conference on Parallel Processing and Applied Mathematics (PPAM)*.
- [146] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Toshiyuki Imamura. 2020. DGEMM using tensor cores, and its accurate and reproducible versions. In *International Conference on High Performance Computing (ISC)*.
- [147] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Toshiyuki Imamura. 2021. Accurate matrix multiplication on binary128 format accelerated by ozaki scheme. In *International Conference on Parallel Processing (ICPP)*.
- [148] Rajib Nath, Stanimire Tomov, Tingxing "Tim" Dong, and Jack Dongarra. 2011. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [149] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An Improved Magma Gemm For Fermi Graphics Processing Units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.
- [150] Roy Nissim and Oded Schwartz. 2019. Revisiting the I/O-complexity of fast matrix multiplication with recomputations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [151] Roy Nissim and Oded Schwartz. 2023. Accelerating Distributed Matrix Multiplication with 4-Dimensional Polynomial Codes. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*.
- [152] Roy Nissim, Oded Schwartz, and Yuval Spiizer. 2024. Fault-tolerant parallel integer multiplication. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [153] Yuyao Niu and Marc Casas. 2025. BerryBees: Breadth first search by bit-tensor-cores. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [154] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [155] NVIDIA. 2025. cuBLAS: Basic Linear Algebra on NVIDIA GPUs. Retrieved April 7, 2025 from <https://developer.nvidia.com/cublas>
- [156] NVIDIA. 2025. cuBLASDX: The cuBLAS Device Extensions. Retrieved April 7, 2025 from <https://docs.nvidia.com/cuda/cublasdx/index.html>
- [157] NVIDIA. 2025. CUTLASS: CUDA Templates for Linear Algebra Subroutines. Retrieved April 7, 2025 from <https://github.com/NVIDIA/cutlass>
- [158] Hiryuki Ootomo, Hidetaka Manabe, Kenji Harada, and Rio Yokota. 2023. Quantum circuit simulation by sgemm emulation on tensor cores and automatic precision selection. In *International Conference on High Performance Computing (ISC)*.
- [159] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. 2024. DGEMM on integer matrix multiplication unit. *The International Journal of High Performance Computing Applications* 38, 4 (2024), 297–313.
- [160] Elmar Peise and Paolo Bientinesi. 2019. The ELAPS framework: Experimental Linear Algebra Performance Studies. *The International Journal of High Performance Computing Applications* 33, 2 (2019), 353–365.
- [161] Xinxin Qi, Jianbin Fang, Peng Zhang, Yonggang Che, Ruibo Wang, Kai Lu, Tao Tang, Chui Huang, and Jie Ren. 2025. Constraint-Driven Auto-Tuning of GEMM-like Operators for MT-3000 Many-core Processor. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [162] Gregorio Quintana-Ortí, Enrique S Quintana-Ortí, Robert A Van De Geijn, Field G Van Zee, and Ernie Chan. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Software* 36, 3 (2009), 1–26.
- [163] Isuru Ranawaka, Md Taufique Hussain, Charles Block, Gerasimos Gerogiannis, Josep Torrellas, and Ariful Azad. 2024. Distributed-Memory Parallel Algorithms for Sparse Matrix and Sparse Tall-and-Skinny Matrix Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [164] Piyush Sao, Ramakrishnan Kannan, Xiaoye Sherry Li, and Richard Vuduc. 2019. A communication-avoiding 3D sparse triangular solver. In *ACM International Conference on Supercomputing (ICS)*.
- [165] Piyush Sao, Xiaoye Sherry Li, and Richard Vuduc. 2018. A communication-avoiding 3D LU factorization algorithm for sparse matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [166] Piyush Sao, Xiaoye S Li, and Richard Vuduc. 2019. A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. *J. Parallel and Distrib. Comput.* 131 (2019), 218–234.
- [167] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydin Buluç. 2021. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *ACM International Conference on Supercomputing (ICS)*.
- [168] Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding, and Omer Khan. 2023. Mergepath-spmm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
- [169] Jaewook Shin, Mary W Hall, Jacqueline Chame, Chun Chen, and Paul D Hovland. 2010. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *Software Automatic Tuning: From Concepts to State-of-the-Art Results*. Springer New York.
- [170] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [171] Edgar Solomonik, Grey Ballard, James Demmel, and Torsten Hoefer. 2017. A communication-avoiding parallel algorithm for the symmetric eigenvalue problem. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [172] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefer. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [173] Edgar Solomonik, Abhinav Bhatele, and James Demmel. 2011. Improving communication performance in dense linear algebra via topology aware collectives. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [174] Edgar Solomonik, Aydin Buluç, and James Demmel. 2013. Minimizing communication in all-pairs shortest paths. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [175] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2014. Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
- [176] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
- [177] Edgar Solomonik, James Demmel, and Torsten Hoefer. 2021. Communication lower bounds of bilinear algorithms for symmetric tensor contractions. *SIAM Journal on Scientific Computing* 43, 5 (2021), A3328–A3356.
- [178] Fengguang Song, Hatem Ltaief, Bilel Hadri, and Jack Dongarra. 2010. Scalable tile communication avoiding QR factorization on multicore cluster systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [179] Saeed Soori, Aditya Devarakonda, Zachary Blanco, James Demmel, Mert Gurubuzbalaban, and Maryam Mehrab Dehnavi. 2018. Reducing communication in proximal Newton methods for sparse least squares problems. In *International Conference on Parallel Processing (ICPP)*.
- [180] Paul Springer and Paolo Bientinesi. 2018. Design of a High-Performance GEMM-like Tensor-Tensor Multiplication. *ACM Trans. Math. Software* 44, 3 (2018).
- [181] Guangming Tan, Linchuan Li, Sean Trieble, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In

- International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*
- [182] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5–6 (2010), 232–240.
- [183] Alok Tripathy, Katherine Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [184] James D. Trotter, Sinan Ekmekçibasi, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [185] Yuki Uchino, Katsuhisa Ozaki, and Toshiyuki Immura. 2025. Performance enhancement of the Ozaki Scheme on integer matrix multiplication unit. *The International Journal of High Performance Computing Applications* 39, 3 (2025), 462–476.
- [186] Yuichiro Ueno and Rio Yokota. 2019. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*.
- [187] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. 2017. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020.
- [188] Field G Van Zee. 2020. Implementing high-performance complex matrix multiplication via the 1m method. *SIAM Journal on Scientific Computing* 42, 5 (2020), C221–C244.
- [189] Field G Van Zee, Devangi N Parikh, and Robert A Van De Geijn. 2021. Supporting mixed-domain mixed-precision matrix multiplication within the BLIS framework. *ACM Trans. Math. Software* 47, 2 (2021), 1–26.
- [190] Field G Van Zee and Tyler M Smith. 2017. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Trans. Math. Software* 44, 1 (2017), 1–36.
- [191] Field G Van Zee, Tyler M Smith, Bryan Marker, Tze Meng Low, Robert A Van De Geijn, Francisco D Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, et al. 2016. The BLIS framework: Experiments in portability. *ACM Trans. Math. Software* 42, 2 (2016), 1–19.
- [192] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Software* 41, 3 (2015), 1–33.
- [193] Vasily Volkov and James Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *The ACM/IEEE conference on Supercomputing (SC)*.
- [194] Hemeng Wang, Wenqing Lin, Qingxiao Sun, and Weifeng Liu. 2025. vGNN: Non-Uniformly partitioned full-graph GNN training on mixed GPUs. *CCF Transactions on High Performance Computing* (2025), 1–18.
- [195] Pengyu Wang, Weiling Yang, Jianbin Fang, Dezun Dong, Chun Huang, Peng Zhang, Tao Tang, and Zheng Wang. 2023. Optimizing Direct Convolutions on ARM Multi-Cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [196] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [197] Tengcheng Wang, Wenhao Li, Haojie Pei, Yuying Sun, Zhou Jin, and Weifeng Liu. 2023. Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation. In *Design Automation Conference (DAC)*.
- [198] Cunyang Wei, Haipeng Jia, Yunquan Zhang, Kun Li, and Luhan Wang. 2022. LBBGEMM: A Load-balanced Batch GEMM Framework on ARM CPUs. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*.
- [199] Cunyang Wei, Haipeng Jia, Yunquan Zhang, Liusha Xu, and Ji Qi. 2022. Ifatf: An input-aware tuning framework for compact blas based on armv8 cpus. In *International Conference on Parallel Processing (ICPP)*.
- [200] Cunyang Wei, Haipeng Jia, Yunquan Zhang, Jianyu Yao, Chendi Li, and Wenxuan Cao. 2024. IrGEMM: An Input-Aware Tuning Framework for Irregular GEMM on ARM and X86 CPUs. *IEEE Transactions on Parallel and Distributed Systems* 35, 9 (2024), 1672–1689.
- [201] Tobias Wicky, Edgar Solomonik, and Torsten Hoefler. 2017. Communication-avoiding parallel algorithms for solving triangular systems of linear equations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [202] Du Wu, Jintao Meng, Wenxi Zhu, Minwen Deng, Xiao Wang, Tao Luo, Mohamed Wahib, and Yanjie Wei. 2024. autoGEMM: Pushing the Limits of Irregular Matrix Multiplication on Arm Architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [203] Junmin Xiao, Shigang Li, Baodong Wu, He Zhang, Kun Li, Erlin Yao, Yunquan Zhang, and Guangming Tan. 2018. Communication-avoiding for dynamical core of atmospheric general circulation model. In *International Conference on Parallel Processing (ICPP)*.
- Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *ACM International Conference on Supercomputing (ICS)*.
- [205] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2021. A pattern-based spgemm library for multi-core and many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 159–175.
- [206] RuQing G Xu, Field G Van Zee, and Robert A van de Geijn. 2023. Towards a Unified Implementation of GEMM in BLIS. In *ACM International Conference on Supercomputing (ICS)*.
- [207] Ichitaro Yamazaki, Sivasankaran Rajamanickam, Erik G Boman, Mark Hoemmen, Michael A Heroux, and Stanimire Tomov. 2014. Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [208] Weiling Yang, Jianbin Fang, and Dezun Dong. 2021. Characterizing small-scale matrix multiplications on ARMv8-based many-core architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [209] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [210] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2024. Optimizing Full-Spectrum Matrix Multiplications on ARMv8 Multi-Core CPUs. *IEEE Transactions on Parallel and Distributed Systems* 35, 3 (2024), 439–454.
- [211] Jianyu Yao, Boqian Shi, Chunyang Xiang, Haipeng Jia, Chendi Li, Hang Cao, and Yunquan Zhang. 2021. Iaat: a input-aware adaptive tuning framework for small gemm. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.
- [212] Yang You, Aydin Buluç, and James Demmel. 2017. Scaling deep learning on GPU and knights landing clusters. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [213] Yang You and James Demmel. 2017. Runtime data layout scheduling for machine learning dataset. In *International Conference on Parallel Processing (ICPP)*.
- [214] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. 2015. CA-SVM: Communication-avoiding support vector machines on distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [215] Yang You, James Demmel, Kent Czechowski, Le Song, and Rich Vuduc. 2016. Design and implementation of a communication-optimal classifier for distributed kernel support vector machines. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2016), 974–988.
- [216] Yang You, Jonathan Hsieh, Chris Ying, James Demmel, Kurt Keutzer, and Chou-Jui Hsieh. 2019. Large-batch training for LSTM and beyond. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
- [217] Kainan Yu, Xinxin Qi, Peng Zhang, Jianbin Fang, Dezun Dong, Ruibo Wang, Tao Tang, Chun Huang, Yonggang Che, and Zheng Wang. 2024. Optimizing General Matrix Multiplications on Modern Multi-core DSPs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
- [218] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.
- [219] Albert-Jan Nicholas Yzelman and Rob H Bisseling. 2009. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing* 31, 4 (2009), 3128–3154.
- [220] Albert-Jan Nicholas Yzelman and Rob H Bisseling. 2012. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In *European Consortium of Mathematics in Industry (ECMI)*. Springer.
- [221] Albert-Jan Nicholas Yzelman and Dirk Roose. 2013. High-level strategies for parallel shared-memory sparse matrix–vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2013), 116–125.
- [222] Kaige Zhang, Xiaoyan Liu, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Jigsaw: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core. In *International Conference on Parallel Processing (ICPP)*.
- [223] Xianyi Zhang. 2016. OpenBLAS: An optimized BLAS library. Retrieved April 7, 2025 from <http://www.openmathlib.org/OpenBLAS/>
- [224] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [225] Keren Zhou, Karthik Ganapathi Subramanian, Po-Hsun Lin, Matthias Fey, Bin-qian Yin, and Jiajia Li. 2024. FASTEN: Fast GPU-accelerated Segmented Matrix Multiplication for Heterogenous Graph Neural Networks. In *ACM International Conference on Supercomputing (ICS)*.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

- C₁ We propose KAMI to extend CA algorithms within a single GPU to accelerate small-scale matrix multiplication.
- C₂ We present a new theoretical analysis scheme for communication and computation in GPU clock cycles.
- C₃ We exploit sparsity and block-wise Z-Morton storage for supporting SpMM and SpGEMM in our CA methods.
- C₄ We implement KAMI on NVIDIA, AMD and Intel GPUs, and show obviously faster performance over SOTA works.

A.2 Computational Artifacts

A₁ <https://doi.org/10.5281/zenodo.16947669>

Artifact ID	Contributions Supported	Related Paper Elements
A ₁	C ₁	Figure 8-15
	C ₂	Figure 15
	C ₃	Figure 13
	C ₄	Figure 8,11,12

B Artifact Identification

B.1 Computational Artifact A₁

Relation To Contributions

This artifact is named KAMI, a set of 1D, 2D, and 3D GEMM algorithms that extend the theory of communication-avoiding (CA) techniques within a single GPU. (C₁) We provide a theoretical analysis of CA performance from the perspective of GPU clock cycles, rather than the traditional execution time. (C₂) Also, we implement sparse-dense matrix-matrix multiplication (SpMM) and sparse general matrix-matrix multiplication (SpGEMM) with this compute-communication pattern. (C₃) Experimental results for general, low-rank, batched, and sparse multiplication on NVIDIA, AMD, and Intel GPUs show significant performance improvements over existing libraries cuBLAS, cuBLASDx, CUTLASS, MAGMA, and SYCL-Bench. (C₄)

Expected Results

This artifact contains KAMI in double and half precision, as well as cuBLAS, cuBLASDx, CUTLASS, MAGMA, and SYCL-Bench. In all test cases, KAMI should be faster than cuBLAS, cuBLASDx, CUTLASS, MAGMA, and SYCL-Bench.

Expected Reproduction Time (in Minutes)

The expected reproduction time consists of three phases: **Setup** (approximately 10 minutes), **Execution** (ranging from 20 to 100 minutes depending on the GPU: 100 minutes on NVIDIA GH200, 30 minutes on RTX 5090, 20 minutes on AMD 7900 XTX, and 40 minutes on Intel Max 1100), and **Analysis** (approximately 10 minutes).

Artifact Setup (incl. Inputs)

Hardware. KAMI is evaluated on four GPUs: NVIDIA GH200, NVIDIA RTX 5090, AMD 7900 XTX and Intel Max 1100.

Software. The NVIDIA GH200 is installed with Ubuntu 22.04 using GCC v11.4 and NVCC v12.8. The NVIDIA RTX 5090 is installed with Ubuntu 24.04 using GCC v11.4 and NVCC v12.8. The AMD 7900 XTX is installed with Ubuntu 24.04 using GCC v11.4 and ROCm 6.10. The Intel Max 1100 is using intel® Tiber™ AI Cloud.

Datasets / Inputs. All input datasets are generated in the code.

Installation and Deployment. The artifact includes ready-to-use shell scripts that automatically compile and prepare the executable before testing. No manual configuration of compilers or environment variables is needed. Users can simply run the provided Bash scripts to complete the installation and compilation process without any additional setup tools.

Artifact Execution

Since the datasets will be automatically generated, the artifact mainly consists of five tasks.

- T₁ Reproduce the original data of the paper and clean it to csv on NVIDIA GH200. Shell scripts `all_GH200.sh` in the `scripts` directory are used to do this.
- T₂ Reproduce the original data of the paper and clean it to csv on NVIDIA RTX 5090. Shell scripts `all_5090.sh` in the `scripts` directory are used to do this.
- T₃ Reproduce the original data of the paper and clean it to csv on AMD 7900 XTX. Shell scripts `all_AMD.sh` in the `scripts` directory are used to do this.
- T₄ Reproduce the original data of the paper and clean it to csv on Intel Max 1100. Shell scripts `all_intel.sh` in the `scripts` directory are used to do this.
- T₅ Reproduce all the plots used in the paper. Python scripts in the `plots` directory are used to do this. It can be done by `plots_all.sh` in the `plots` directory.

Dependencies: $T_1, T_2, T_3, T_4 \rightarrow T_5$.

Artifact Analysis (incl. Outputs)

The artifact contains four folders: `src`, `scripts`, `logs`, and `plots`. The `src` folder contains the source code of *KAMI*. The `scripts` folder contains the shell scripts to reproduce the experiments in the paper. The `log` file contains all the script outputs and will be saved in `logs`. And `log` file will be clean to `csv` file in `logs`. The `plots` contains Python scripts, which will reproduce all the plots used in the paper.

Figure 8 shows the performance of KAMI on NVIDIA, AMD and Intel GPUs. Figure 11 shows the low-rank GEMM of KAMI and cuBLASDx results on GH200 in FP16. Figure 12 compares the performance of batched GEMM of KAMI, cuBLAS and MAGMA in FP64 on GH200. Figure 13 presents the performance of SpMM and SpGEMM in FP16 on the GH200 platform. In Figure 15, we compare the theoretical register and cycle usage of KAMI with the actual measured and theoretical values.

Artifact Evaluation (AE)

C.1 Computational Artifact A₁

Artifact Setup (incl. Inputs)

Target Platform. This artifact supports execution on recent GPUs from NVIDIA (SM90 or later), AMD, and Intel. We provide platform-specific implementations and compilation scripts for each vendor. The artifact has been tested on the following platforms:

- NVIDIA GH200: Ubuntu 22.04, GCC 11.4, CUDA 12.8
- NVIDIA RTX 5090: Ubuntu 24.04, GCC 11.4, CUDA 12.8
- AMD 7900 XTX: Ubuntu 24.04, GCC 11.4, ROCm 6.10
- Intel Max 1100: Intel® Tiber™ AI Cloud with oneAPI 2025.0.1

Python 3 with NumPy, Matplotlib, Seaborn, and Pandas is used for plotting and analyzing results across all platforms.

Installation Instructions.

- (1) Clone the repository:

```
git clone https://github.com/ForADAE/SC25-pap926
cd SC25-pap926
```

- (2) (Optional) Set up Python environment for plotting:

```
pip3 install numpy matplotlib seaborn pandas
```

Input Data. All inputs are synthetic and automatically generated at runtime. No dataset download is required.

Artifact Execution

Workflow Overview. Since the datasets are automatically generated, the artifact consists of five main tasks:

- T₁** Reproduce the original experimental data and convert it to CSV format on NVIDIA GH200. This is done using the script `all_GH200.sh` in the `scripts` directory.
- T₂** Reproduce the original experimental data and convert it to CSV format on NVIDIA RTX 5090. This is done using the script `all_5090.sh` in the `scripts` directory.
- T₃** Reproduce the original experimental data and convert it to CSV format on AMD 7900 XTX. This is done using the script `all_AMD.sh` in the `scripts` directory.
- T₄** Reproduce the original experimental data and convert it to CSV format on Intel Max 1100. This is done using the script `all_intel.sh` in the `scripts` directory.
- T₅** Reproduce all plots used in the paper. Python scripts in the `plots` directory are used for this purpose. A single script `plots_all.sh` can be used to regenerate all figures.

Although the full reproduction requires access to multiple GPU platforms, we provide all the original output logs collected from our experimental environment under the `logs` directory. As a result, reproducing all plots and verifying the results is possible even without access to the complete set of hardware platforms.

Execution Steps.

- (1) Run benchmarks on NVIDIA GH200 (including square GEMM, low-rank GEMM, SpMM/SpGEMM, batched GEMM, and cycle cost evaluation):

```
cd scripts
bash all_GH200.sh
```

- (2) Run benchmarks on NVIDIA RTX 5090 (including square GEMM, block count vs. TFLOPS correlation, register usage, cycle cost, and register/shared memory analysis):

```
bash all_5090.sh
```

- (3) Run benchmarks on AMD 7900 XTX (including square GEMM):

```
bash all_AMD.sh
```

- (4) Run benchmarks on Intel Max 1100 (including square GEMM):

```
bash all_intel.sh
```

- (5) Generate all figures:

```
cd ../plots
bash plots_all.sh
```

Output Locations.

- `logs/`: Contains raw logs and cleaned CSV results.
- `plots/`: Contains regenerated plots in PDF formats.

Expected Runtime. The overall runtime is divided by platform as follows:

- NVIDIA GH200: ~100 minutes
- NVIDIA RTX 5090: ~30 minutes
- AMD 7900 XTX: ~20 minutes
- Intel Max 1100: ~40 minutes

Artifact Analysis (incl. Outputs)

Artifact Analysis. The expected results of this artifact include the numerical performance data collected from four different GPU platforms and the complete set of visualizations (figures) reproduced from the paper. These figures include performance comparisons of KAMI under different matrix shapes (square, low-rank, sparse, and batched), as well as hardware-level analysis such as register usage, cycle costs, and block scheduling.

All key experimental results presented in the paper are reproducible via this artifact. The corresponding logs and cleaned CSVs are stored in the `logs` directory, and the plots are regenerated and stored in `plots`. These plots correspond directly to the figures included in the main paper.

The evaluation methodology follows the five-step workflow defined in the Execution section (T_1-T_5). The Python scripts used for analysis ensure consistency with the data transformation and visualization pipelines used in the paper. Users can inspect the intermediate and final outputs to verify whether the reproduced figures match the original conclusions.

Pre-collected logs for all platforms allow users to regenerate all plots and verify the paper's claims, even without access to all hardware.

Overall, the artifact faithfully reflects the paper's contributions, and the outputs it produces are designed to be directly comparable to the published results.