A Tile-Based BLAS Library for Tenstorrent Dataflow Architectures

Calebe P. Bianchini[1,3], Evaldo Costa[1], Felipe Portella[2], Gabriel Freytag[1], Jeferson S. Rocha[2], Leon Barboza[1], Maicon M. Alves[2], Rodrigo Machado[1], Vinícius Ventura[1]

[1] CESAR Innovation Center, Recife, PE, Brazil

[2]Petróleo Brasileiro S.A. (PETROBRAS), Rio de Janeiro, RJ, Brazil

[3]Mackenzie Presbyterian University, São Paulo, SP, Brazil

{cpb,ebc,gf2,lmub,rcm,vlv}@cesar.org.br

{felipeportella,jsrocha,maicon.melo}@petrobras.com.br

calebe.bianchini@mackenzie.br

Abstract. *Emerging AI accelerators increasingly adopt spatial and dataflow execution models based on distributed compute units, local memories, and explicit data movement. Although primarily designed for machine learning, these architectures are also being explored for scientific computing. This paper presents a tile-based BLAS library for Tenstorrent dataflow architectures, mapping representative Level-1, Level-2, and Level-3 operations onto Tensix cores through tiled layouts, circular buffers, explicit DRAM–SRAM movement, and local data reuse. We describe the main design decisions required by this execution model, including data layout, kernel decomposition, synchronization, data-format constraints, and workload distribution. The library is evaluated on a dual-ASIC Wormhole™ n300 accelerator using device-side profiling in single-chip and dual-chip configurations. AXPY and GER show sublinear multichip scaling, whereas GEMM benefits more strongly from the second ASIC, reaching a maximum measured throughput of* 5.79 *TFLOP/s and an observed maximum speedup of* $2.07\times$*. The results show that tile organization, data movement, arithmetic intensity, and workload distribution strongly influence BLAS performance and multichip scaling on Tenstorrent accelerators.*

1. Introduction

High Performance Computing (HPC) has historically evolved alongside architectures exposing new forms of parallelism, locality, and data movement. Recent interest in RISC-V and dataflow-oriented accelerators has renewed the discussion about how emerging architectures can support scientific workloads beyond their original application domains [Brown 2024, Brown 2025, Lee et al. 2023, Diehl et al. 2024]. Although many applications can be functionally ported to these systems, achieving competitive performance still depends heavily on architecture-aware numerical software, memory organization, synchronization, and data movement.

Tenstorrent accelerators follow a spatial dataflow model built around Tensix cores with compute engines and software-managed per-core L1 SRAM. Computation uses

tiles, typically 32×32 blocks of values on Wormhole, transferred explicitly between device DRAM and local SRAM. Circular buffers in SRAM connect reader, compute, and writer stages, which load, process, and store tiles, respectively [Tenstorrent 2025d, Tenstorrent 2025a]. Unlike cache-oriented CPU execution or conventional GPU thread hierarchies, this model makes data movement and synchronization explicit.

Basic Linear Algebra Subprograms (BLAS) provide a compact framework for studying these characteristics, spanning bandwidth-sensitive vector operations, matrix-vector computations, and compute-intensive matrix-matrix kernels [Lawson et al. 1979, Dongarra et al. 1990, Blackford et al. 2002]. On CPUs and GPUs, optimized BLAS implementations rely on blocking, vectorization, and data reuse [Goto and van de Geijn 2008, Van Zee and van de Geijn 2015].

This paper presents a reusable tile-based BLAS library implementing five routines: AXPY (scaled vector addition) and SCAL (vector scaling) at Level 1, GEMV (matrix-vector product) and GER (rank-one matrix update) at Level 2, and GEMM (matrix-matrix product) at Level 3. The experimental study selects AXPY, GER, and GEMM to contrast vector, rank-one update, and matrix-matrix workloads across the three BLAS levels. These routines share tiled layouts, explicit DRAM–SRAM movement, circular-buffer synchronization, kernel orchestration, and workload distribution, while compute kernels implement operation-specific arithmetic.

The main contributions of this work are summarized as follows:

- A reusable tile-based BLAS software layer for Tenstorrent dataflow architectures, implementing representative Level-1, Level-2, and Level-3 operations through shared data-layout, data-movement, synchronization, and workload-distribution mechanisms.
- A data-layout and kernel orchestration strategy based on tiles, circular buffers, and explicit movement between memory levels.
- A discussion of the software-visible constraints introduced by the Tensix execution model, including synchronization, tiling, padding, and tile reuse.
- A single- and dual-ASIC experimental characterization of representative BLAS routines on a Wormhole n300 accelerator.

By focusing on BLAS, this work studies Tenstorrent from a scientific-computing perspective. The library is not intended to replace mature CPU or GPU BLAS implementations, but to provide a foundation for studying numerical kernels and reusable abstractions on spatial dataflow architectures.

The remainder of the paper covers related work, Tensix background, the BLAS implementation, experimental results, and conclusions in Sections 2–6.

2. Related Work

Studies of RISC-V and dataflow-oriented accelerators have examined vectorization, runtime systems, portability, and distributed execution for HPC, emphasizing software maturity, memory hierarchy management, and architecture-specific optimization [Brown 2024, Brown 2025, Lee et al. 2023, Diehl et al. 2024].

Tenstorrent HPC studies have mainly focused on individual kernels. Stencil implementations [Brown and Barton 2024, Piarulli and Sensi 2026] show sensitiv-

ity to alignment, tiling, local SRAM use, and synchronization; avoiding unnecessary tiling and tensor reorganization can improve efficiency. FFTs and N-body simulations [Brown et al. 2025, Almerol et al. 2025] likewise emphasize data movement and workload decomposition. Other work characterizes Wormhole performance in terms of locality, SRAM reuse, communication overlap, tiling, and matrix layout [Taylor et al. 2026, Pizzini Cavagna et al. 2025]. Together, these studies identify key Tensix performance factors but remain centered on isolated kernels.

Architecture-aware BLAS libraries traditionally rely on blocking, vectorization, cache reuse, and specialized microkernels [Goto and van de Geijn 2008, Van Zee and van de Geijn 2015]. RISC-V studies further show that BLAS interfaces and GEMM implementations may require redesign for new memory hierarchies, operand layouts, and reuse mechanisms [Pirova et al. 2026, Krebs et al. 2026]. To the best of our knowledge, prior Tenstorrent work has not investigated a reusable BLAS-oriented layer sharing tile layout, data movement, synchronization, and workload distribution across multiple BLAS levels.

3. Background

Tenstorrent uses a spatial execution model with Tensix cores connected by a Network-on-Chip (NoC). Tiles move between device DRAM, per-core L1 SRAM, and compute engines through explicit, software-controlled transfers [Tenstorrent 2025d, Tenstorrent 2025a].

3.1. Tensix Core

Each Tensix core integrates RISC-V control processors, specialized compute engines, per-core L1 SRAM, and NoC interfaces. RISC-V processors coordinate data movement and synchronization, whereas arithmetic operations execute on specialized engines. The L1 SRAM is software-managed rather than a hardware cache. Figure 1 illustrates a high-level Tensix core based on the Tensix Neo architecture.

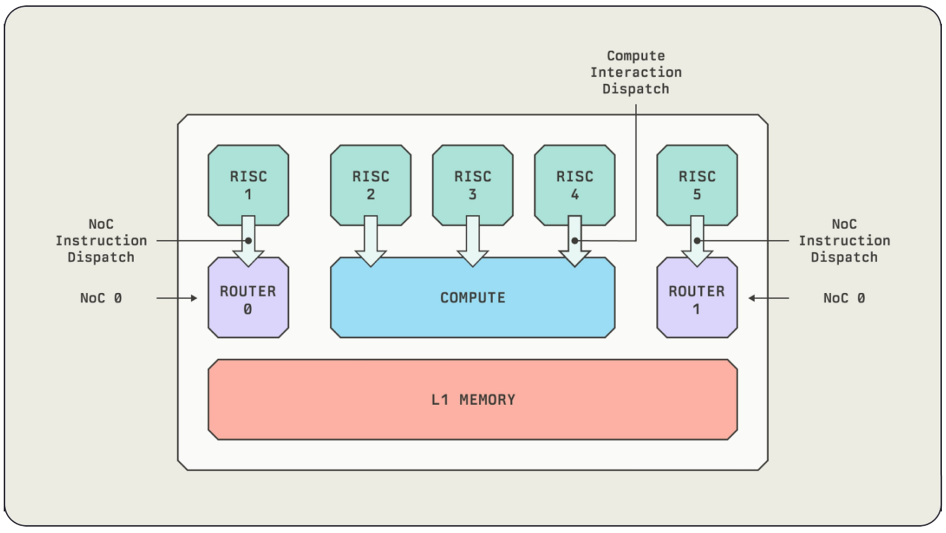

Figure 1. High-level organization of a Tensix core based on the Tensix Neo architecture [Tenstorrent 2025b].

The capacity of per-core L1 SRAM limits tile dimensions, circular-buffer depth, and kernel granularity. Level-1 routines typically offer limited data reuse; Level-2 routines may require padding and vector replication, whereas Level-3 routines can reuse matrix tiles across multiple arithmetic operations. NoC interfaces support asynchronous tile transfers between DRAM, cores, and local buffers.

3.2. Tensix Execution Model

The TT-Metal and LLK software stack organizes a program into reader, compute, and writer kernels [Tenstorrent 2025c, Tenstorrent 2025d, Tenstorrent 2025a]. Figure 2 depicts this pipeline, with circular buffers in per-core L1 SRAM connecting its stages. Readers load tiles from DRAM, compute kernels process them, and writers store results.

Inside compute kernels, *unpack* prepares operands for the compute engines, *math* performs arithmetic, and *pack* formats result tiles. Circular buffers implement producer–consumer synchronization through reserve, push, wait, and release operations, allowing communication and computation to overlap when scheduled appropriately.

Host-side tilization converts conventional row-major data into tiled layouts, while untilization reconstructs outputs. Their costs are relevant to end-to-end execution but are not captured by device-only timing.

4. Porting BLAS Operations

Mapping BLAS routines onto Tenstorrent architectures requires more than directly translating conventional CPU or GPU implementations into device kernels. Classical BLAS interfaces assume dense vector and matrix operands organized in contiguous layouts, while Tensix execution is fundamentally structured around fixed-size tensor tiles, explicit movement between memory levels, and software-managed synchronization. As a result, BLAS operations must be reformulated in terms of tile-compatible data layouts, circular buffers, and streaming execution stages. The following sections describe the main design decisions adopted in our library, including data-layout transformations, kernel organization, and work-distribution strategies used to adapt representative BLAS routines to the Tensix execution model.

4.1. Data Layout and Tiling Strategy

Tenstorrent kernels operate on fixed-size tensor tiles rather than arbitrary matrix dimensions [Tenstorrent 2025e]. Consequently, BLAS operands must be transformed into tile-aligned representations before execution. In current Wormhole implementations, tiles commonly use dimensions $T_h = T_w = 32$, requiring matrix dimensions to be aligned to tile boundaries. When dimensions do not naturally satisfy these constraints, padding must be introduced explicitly on the host side before data is transferred to the device.

This requirement affects vector-oriented BLAS routines more significantly than matrix-matrix operations. Dense matrices can usually be partitioned directly into tiles after boundary padding, while vectors frequently require additional transformations to become compatible with tensor-oriented execution. In practice, vectors cannot always be consumed directly by Tensix compute kernels and therefore must be reorganized into tiled matrix-like layouts.

For simple vector operations, padding consists primarily of extending the input dimensions with zeros until the data fits the tile boundaries. This transformation preserves the original semantics of the operation while allowing the resulting tensors to be processed by tile-oriented kernels.

For instance, in Level-2 BLAS routines, the padded vector layout must also account for how vector operands participate in matrix-style tensor operations internally.

The $GER(x, y)$ operation illustrates this requirement. Because GER forms a matrix-shaped outer product from two vectors, the padded representation must specify how each vector is expanded and replicated across tile-aligned matrix regions. Let

$$M_t = \left\lceil \frac{M}{T_h} \right\rceil, \qquad N_t = \left\lceil \frac{N}{T_w} \right\rceil$$

represent the number of tiles along each matrix dimension. The vector $x \in \mathbb{R}^M$ is partitioned into blocks

$$x^{(i)} \in \mathbb{R}^{T_h \times 1},$$

while the vector $y \in \mathbb{R}^N$ is partitioned into

$$y^{(j)} \in \mathbb{R}^{1 \times T_w}.$$

The resulting padded representations of x and y replicate vector elements across tile-aligned regions and are given by:

$$x_{padded} = \begin{bmatrix} [x^{(1)} & \bar{0}] & \cdots & [x^{(1)} & \bar{0}] \\ \vdots & & \ddots & \vdots & \\ [x^{(M_t)} & \bar{0}] & \cdots & [x^{(M_t)} & \bar{0}] \end{bmatrix}, y_{padded} = \begin{bmatrix} \begin{bmatrix} y^{(1)} \\ \bar{0} \end{bmatrix} & \cdots & \begin{bmatrix} y^{(N_t)} \\ \bar{0} \end{bmatrix} \\ \vdots & \ddots & \vdots \\ \begin{bmatrix} y^{(1)} \\ \bar{0} \end{bmatrix} & \cdots & \begin{bmatrix} y^{(N_t)} \\ \bar{0} \end{bmatrix} \end{bmatrix}.$$

These transformations allow vector operands to be consumed by tile-based tensor kernels while preserving the semantics of the original BLAS operation. A similar strategy is employed for other BLAS operations, such as GEMV, where vectors are expanded into tile-oriented layouts compatible with the matrix execution pipeline.

These transformations introduce additional preprocessing overheads that are largely absent from conventional CPU BLAS implementations. However, they are necessary to expose locality and tile reuse to the Tensix execution model, allowing operands to be streamed efficiently through the unpack, compute, and pack stages described in Section 3.

4.2. Kernel Design

The execution pipeline of the proposed BLAS library follows the Tensix dataflow model by decomposing each operation into reader, compute, and writer kernels. This organization allows communication and computation to progress concurrently while exposing synchronization and movement explicitly at the software level. As illustrated in Figure 2, tiles are streamed through circular buffers between movement and compute stages.

Figure 2. Tile movement and execution flow across reader, compute, and writer kernels.

Reader and writer kernels are structurally similar across most BLAS operations because their primary role is data movement. The host provides each core with the base addresses of the tensors, the starting tile index, and the number of tiles assigned to that core. Reader kernels then generate NoC addresses and asynchronously transfer tiles from DRAM into per-core L1 SRAM through circular buffers.

Synchronization between stages is coordinated through explicit producer-consumer semantics. Reader kernels reserve space in the destination buffer through `cb_reserve_back`, transfer tiles asynchronously with `noc_async_read_tile`, and publish completed transfers through `cb_push_back`. Writer kernels implement the complementary behavior by waiting for processed tiles with `cb_wait_front`, writing results back to DRAM, and releasing SRAM space through `cb_pop_front`.

The compute kernels contain the operation-specific logic of the BLAS implementation. Arithmetic stages are mapped to the Tensix compute engines: the matrix engine (FPU), used for tile-level matrix operations, and the vector engine (SFPU), used for vectorized arithmetic and special-function operations. While circular buffers provide the streaming interface between kernels, SFPU operations can consume operands already available in the register set and produce intermediate results that are reused by subsequent operations without being written back to circular buffers. This organization reduces SRAM traffic, synchronization overheads, and unnecessary movement between execution stages.

These optimizations become particularly important for Level-2 and Level-3 BLAS routines, where accumulation and tensor transformations can otherwise generate substantial communication overheads. Consequently, performance depends not only on arithmetic throughput, but also on synchronization frequency, buffer occupancy, movement scheduling, and tile reuse across the execution pipeline. For GEMM, each core processes a linear range of output tiles. The reader fetches the corresponding A and B tiles for each of the K_t reduction steps, synchronizing each NoC read. The Matrix Engine accumulates tile products in destination registers, then passes the result through an intermediate circular buffer to the SFPU for $\alpha AB + \beta C$.

4.3. Parallelization Strategy

To achieve high-performance execution across the accelerator, we implement a static workload partitioning model. Because the Tensix architecture follows a Single Program, Multiple Data (SPMD) execution paradigm, the host is responsible for explicitly calculating and distributing workload indices to each core prior to kernel launch.

Our strategy leverages the `split_work_to_cores` utility from the TT-Metalium framework to map the total tile space onto the available compute grid. The framework partitions the workload into two primary groups to handle non-uniform distributions; the first group contains cores assigned the base workload, while the second group

handles the remainder when the total tile count is not perfectly divisible by the number of active cores. Once the workload is partitioned, the host iterates through the core grid and uses the `SetRuntimeArgs` mechanism to provide each core with its specific execution context. This context includes the base addresses of the input and output DRAM buffers, the starting tile index, and the local tile count.

For multi-chip execution, the tile space is first partitioned between the two ASICs using TT-Metalium primitives. The per-device tile count is then passed to `split_work_to_cores` to distribute each partition across the local compute grid. Runtime arguments identify the tile range assigned to each core, and the reader kernel computes the corresponding offsets so that each ASIC streams the correct data partition from DRAM into per-core L1 SRAM.

After the reader stage has populated the local buffers, each compute kernel executes independently over the tile range defined by its runtime arguments, using the Tensix compute engines described in Section 4.2. The writer kernel completes the pipeline by consuming the final tiles from per-core L1 SRAM and writing the corresponding results back to DRAM.

5. Experiments and Results

5.1. Experimental Setup

The performance evaluation of our tile-based BLAS library was conducted on the Koyeb cloud platform using a Tenstorrent Wormhole™ n300 accelerator, made available through an experimental grant provided by Tenstorrent. The platform uses a dual-ASIC configuration, with 64 Tensix cores per ASIC and 128 programmable compute units in total, with a nominal AI clock frequency of 1 GHz. The accelerator provides 192 MB of aggregate per-core L1 SRAM (96 MB per ASIC) and 24 GB of GDDR6 global memory. At 1 GHz, the theoretical HiFi4 matrix-compute peak is approximately 65.54 TFLOP/s per ASIC, or 131.07 TFLOP/s for both ASICs combined [Tenstorrent 2026].

5.2. Measurement Methodology

We initially employed a subtractive profiling methodology based on host-side C++ timing instrumentation. End-to-end execution was decomposed into host processing, PCIe data transfers, and device execution, allowing non-device components to be subtracted from the total execution time.

However, because the experiments were conducted in a cloud environment with a virtualized host CPU, host-side measurements were subject to virtualization overhead and scheduling variability. In addition, run-to-run variation in the effective device clock frequency could distort comparisons between execution configurations. We therefore adopted device-side measurements using Tenstorrent's built-in Device Program Profiler, which isolates timing from host virtualization and scheduling effects while also reporting the effective device clock frequency for each run.

Two configurations were evaluated. The *single-chip* configuration uses only the host-facing ASIC, while the *dual-chip* configuration executes concurrently on both ASICs.

Device execution time was measured using scope-based profiler zones placed in the reader and writer paths. The measurement interval begins with the reader-side transfer feeding the computation and ends when the writer-side reading of the computed data is completed. Since data movement and computation overlap in the Wormhole dataflow pipeline, this interval includes the computation and any overlapping transfers and is therefore treated as a conservative, worst-case estimate of kernel execution time. For dual-chip execution, the interval is measured independently on each ASIC, and the maximum of the two durations is reported as the total execution time.

Across runs, non-negligible latency variation was observed. Because the experiments were conducted on the Koyeb cloud platform, the platform-level operating conditions were outside our control. The profiler-reported effective device clock frequency showed that some runs operated below the nominal frequency of 1 GHz. Such frequency variation can distort comparisons between execution configurations, including apparent superlinear speedups. To reduce the influence of these uncontrolled conditions, we select the 10 iterations with the highest profiler-reported effective device clock frequencies among the 150 iterations of each experiment. The reported measurements therefore characterize runs in which the accelerator operates closest to its nominal frequency, rather than the overall performance distribution of the cloud environment. The reported values are the mean of these ten selected iterations. Problem size is reported as the total number of 32×32 tiles.

5.3. Precision and Data Format

All experiments in this evaluation were conducted using the `bfloat16` data format, with GEMM using HiFi4 math fidelity. Although Tenstorrent hardware provides support for single-precision floating-point (`float32`), it is not fully IEEE 754-compliant and requires additional software configuration at both the host and kernel levels.

Architecturally, full 32-bit floating-point arithmetic is restricted to the SFPU vector engine. The main Matrix Engine (FPU) accepts 16-bit inputs and supports 32-bit precision only for intermediate accumulation in destination registers. Consequently, `bfloat16` represents the native data format for evaluating the hardware's primary compute pipeline. Level-1 BLAS operations can, however, be executed entirely on the SFPU using 32-bit floating-point arithmetic, as these operations do not require matrix multiplication, which is not supported by the SFPU.

5.4. Evaluation Metrics

We evaluate performance using three metrics:

1. **Device Execution Time (Latency):** The profiler-zone interval described above, characterizing execution within the reader–writer bounds.
2. **Compute Throughput (GFLOP/s):**

$$\text{GFLOP/s} = \frac{N_{\text{FLOPs}}}{T \cdot 10^9}, \tag{1}$$

where N_{FLOPs} is the useful floating-point operation count and T is the measured device interval in seconds. For the evaluated workloads, the counts are $2N$ for AXPY (vector length N), $2MN$ for GER (matrix size $M \times N$), and $2n^3$ for square

GEMM (dimension n), excluding extra work caused by padding or operand replication.

3. **Arithmetic Intensity (FLOPs/element):** For GEMM,

$$\text{Arithmetic Intensity} = \frac{N_{\text{FLOPs}}}{N_{\text{elems}}}, \tag{2}$$

where N_{elems} is the minimum theoretical number of element accesses required by the operation. Unlike conventional intensity in FLOPs/byte, this metric does not represent measured memory traffic or accesses introduced by padding and replication.

5.5. Benchmark Analysis: Low-Arithmetic-Intensity Workloads

AXPY is a low-arithmetic-intensity Level-1 BLAS operation, performing only a small amount of computation for each element moved through the memory hierarchy. Its performance is therefore expected to be particularly sensitive to data movement, limiting the extent to which the additional computational resources provided by the second chip can be exploited. The measured execution times and throughput are reported in Table 1.

As shown in Figure 3, the multichip configuration consistently outperforms the single-chip configuration across the entire evaluated range. At 2^8 tiles, performance increases from 32.50 GFLOP/s on a single chip to 43.00 GFLOP/s on two chips, corresponding to a $1.32\times$ speedup. As the problem size increases, the performance advantage remains relatively stable, reaching 137.48 GFLOP/s versus 100.58 GFLOP/s at 2^{16} tiles, or a $1.37\times$ speedup. The corresponding execution time decreases from 0.0161 to 0.0122 ms at 2^8 tiles and from 1.334 to 0.976 ms at 2^{16} tiles.

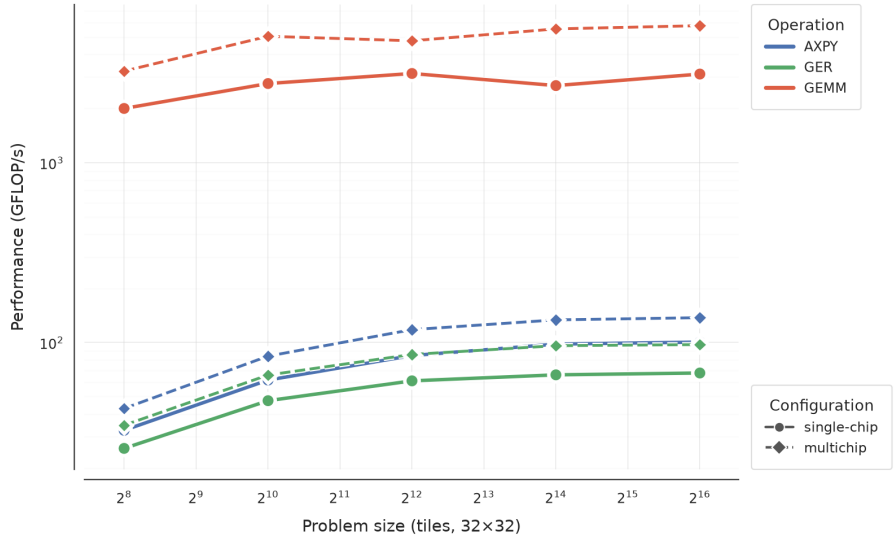

Figure 3. Compute throughput for AXPY, GER, and GEMM.

Although a second chip provides additional compute and memory resources, AXPY exposes limited computation per element moved. The observed sublinear scaling is therefore consistent with a workload for which data movement has a stronger relative influence on device execution than in higher-arithmetic-intensity operations.

GER presents a similar low-arithmetic-intensity behavior. As shown in Figure 3, the multichip configuration consistently improves performance over the single-chip configuration. At 2^8 tiles, throughput increases from 25.78 to 34.69 GFLOP/s, corresponding to a $1.35\times$ speedup. The advantage grows gradually with problem size, reaching 97.42 GFLOP/s compared with 67.69 GFLOP/s at 2^{16} tiles, yielding a $1.44\times$ speedup. Execution time follows the same trend, decreasing from 0.02034 to 0.01512 ms at 2^8 tiles and from 1.983 to 1.378 ms at 2^{16} tiles.

The somewhat larger scaling observed for GER compared with AXPY may be related to differences in workload structure, without necessarily indicating higher effective arithmetic intensity. As discussed in Section 4.1, GER requires vector replication and padding into tile-aligned layouts, potentially increasing device-side data movement without adding useful FLOPs. Nevertheless, scaling remains below the ideal $2\times$ improvement, consistent with a stronger influence of data movement than in GEMM. The increasing throughput with problem size also suggests improved hardware utilization. For the single-chip configuration, performance increases from 25.78 GFLOP/s at 2^8 tiles to 67.69 GFLOP/s at 2^{16} tiles, while multichip performance increases from 34.69 to 97.42 GFLOP/s.

5.6. Benchmark Analysis: High-Arithmetic-Intensity Workloads

GEMM exhibits substantially higher arithmetic intensity than AXPY and GER, making it well suited for evaluating the computational scaling of the Matrix Engine. As shown in Figure 4, the arithmetic intensity increases from approximately 341 FLOPs/element at 2^8 tiles to $5{,}461$ FLOPs/element at 2^{16} tiles. These values express the ratio between the floating-point work performed and the minimum theoretical number of element accesses required by the operation. At the largest problem size, this ratio is approximately sixteen times higher than at 2^8 tiles, characterizing an increasingly compute-intensive workload. Since this theoretical metric is independent of the number of active ASICs, the single- and dual-chip curves overlap exactly in the figure.

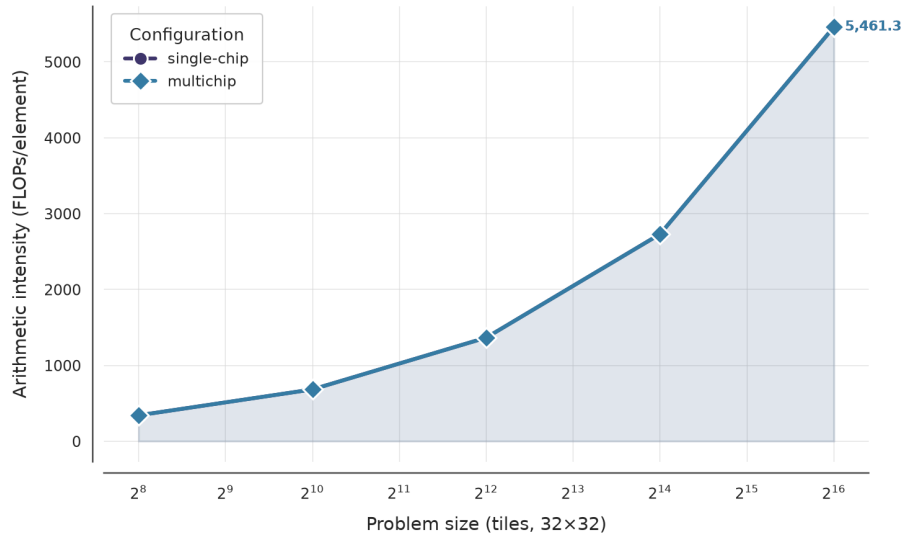

Figure 4. GEMM arithmetic intensity in FLOPs/element.

The measured execution times and throughput are reported in Table 1. The multi-

chip configuration consistently outperforms the single-chip configuration across all evaluated problem sizes. At 2^8 tiles, performance increases from $2{,}003.68$ GFLOP/s on a single chip to $3{,}233.32$ GFLOP/s in the multichip configuration, corresponding to a $1.61\times$ speedup. At 2^{10} tiles, throughput increases from $2{,}756.94$ to $5{,}049.54$ GFLOP/s, yielding a $1.83\times$ speedup. For 2^{12} tiles, the measured performance is $3{,}135.84$ GFLOP/s for the single-chip configuration and $4{,}765.85$ GFLOP/s for the multichip configuration, corresponding to a $1.52\times$ speedup.

The strongest scaling is observed at 2^{14} tiles. As reported in Table 1, multichip execution reaches $5{,}550.85$ GFLOP/s compared with $2{,}685.38$ GFLOP/s for the single-chip configuration, resulting in a $2.07\times$ speedup.

At 2^{16} tiles, the multichip configuration reaches its highest measured throughput of $5{,}785.57$ GFLOP/s, compared with $3{,}112.84$ GFLOP/s for the single-chip configuration. This corresponds to a $1.86\times$ speedup. The decrease in single-chip throughput at 2^{14} tiles, followed by a recovery at 2^{16}, also indicates that achieved performance is not determined solely by problem size. Factors such as workload distribution, device utilization, and execution variability can affect the realized throughput even when the arithmetic intensity continues to increase.

At 2^{16} tiles, the single- and dual-chip GEMM throughputs represent approximately 4.75% and 4.41% of their respective theoretical HiFi4 peaks. Thus, the second ASIC increases absolute throughput without increasing the fraction of the nominal compute peak achieved. These ratios do not isolate the effects of computation, data movement, and synchronization on device execution time.

Table 1. Device execution time and throughput for AXPY, GER, and GEMM using `bfloat16`.

		AXPY		GER		GEMM	
Configuration	Size (Tiles)	Device (ms)	GFLOP/s	Device (ms)	GFLOP/s	Device (ms)	GFLOP/s
single-chip	2^8	0.016133	32.50	0.020340	25.78	0.133971	2003.68
multichip	2^8	0.012194	43.00	0.015115	34.69	0.083022	3233.32
single-chip	2^{10}	0.033914	61.84	0.044163	47.49	0.778938	2756.94
multichip	2^{10}	0.024961	84.02	0.031799	65.95	0.425283	5049.54
single-chip	2^{12}	0.099102	84.65	0.137026	61.22	5.478553	3135.84
multichip	2^{12}	0.071113	117.96	0.097701	85.86	3.604789	4765.85
single-chip	2^{14}	0.342386	98.00	0.507481	66.12	51.180468	2685.38
multichip	2^{14}	0.251015	133.67	0.349402	96.03	24.759995	5550.85
single-chip	2^{16}	1.334422	100.58	1.982711	67.69	353.218389	3112.84
multichip	2^{16}	0.976275	137.48	1.377717	97.42	190.043723	5785.57

6. Conclusions and Future Work

This paper introduced a tile-based BLAS library for Tenstorrent dataflow architectures and examined how representative linear algebra operations map to the Tensix execution model. The implementation uses tiled data layouts, circular-buffer synchronization, explicit memory movement, and static workload partitioning to provide a reusable software structure across BLAS levels.

The experiments show that multichip scaling depends strongly on workload characteristics. For the lower-arithmetic-intensity AXPY and GER kernels, dual-chip execution achieved speedups of approximately 1.32–$1.37\times$ and 1.35–$1.44\times$, respectively. GEMM benefited more strongly from the second ASIC. It reached a maximum measured throughput of 5.79 TFLOP/s at 2^{16} tiles, while the largest observed speedup was $2.07\times$ at 2^{14} tiles. Because the effective device frequency varied across the cloud experiments, the $2.07\times$ value should be interpreted as an observed measurement under the selected near-nominal-frequency runs, rather than as evidence of algorithmic superlinear scaling.

The results also reinforce that performance on Tensix depends on more than arithmetic throughput. Tile organization, SRAM reuse, synchronization, device-side data movement, and workload distribution all affect the realized performance, particularly for operations with limited computation per element moved. Device-side profiling was important because host virtualization and scheduling made host-side timing unsuitable, while the profiler-reported effective device frequency exposed clock variation across runs.

The proposed library provides a foundation for investigating conventional HPC numerical kernels on Tenstorrent dataflow accelerators whose Tensix cores integrate embedded RISC-V processors and specialized arithmetic engines. Future work will investigate more effective overlap of communication and computation, improved tile-reuse and multichip data-distribution strategies, and support for additional data types and more complex BLAS routines. These directions can further clarify how spatial dataflow accelerators can be used for high-performance scientific computing beyond their primary AI workloads.

Use of Artificial Intelligence

The authors used AI tools to assist in restructuring and condensing the manuscript and to generate preliminary English-language formulations for selected passages throughout the article, particularly in the Abstract, table formatting, and the Conclusion. The tool also assisted with LaTeX phrasing and consistency checking. It was not used to collect or fabricate data, perform experiments, calculate the metrics reported in the evaluation, or independently conduct the evaluation. The paper's bibliographic references were independently selected and verified by the authors and were not generated by the tool. All AI-assisted content, factual claims, citations, and conclusions were critically reviewed, revised, and verified by the authors, who take full responsibility for the final content.

References

Almerol, J. L., Boella, E., Spera, M., and Gregori, D. (2025). Accelerating gravitational n-body simulations using the risc-v-based tenstorrent wormhole.

Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Różańska, G., and Wasniewski, J. (2002). An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151.

Brown, N. (2024). RISC-V for HPC: Where We Are and Where We Need to Go.

Brown, N. (2025). RISC-V for HPC: An Update of Where We Are and Main Action Points.

Brown, N. and Barton, R. (2024). Accelerating stencils on the tenstorrent grayskull risc-v accelerator.

Brown, N., Davies, J., and Clair, F. L. (2025). Exploring fast fourier transforms on the tenstorrent wormhole.

Diehl, P., Syskakis, P., Daiß, G., Brandt, S. R., Kheirkhahan, A., Singanaboina, S. Y., Marcello, D., Taylor, C., Leidel, J., and Kaiser, H. (2024). Preparing for HPC on RISC-V: Examining Vectorization and Distributed Performance of an Astrophysics Application with HPX and Kokkos. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1656–1665. IEEE Computer Society.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17.

Goto, K. and van de Geijn, R. A. (2008). Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25.

Krebs, C. P., Felzmann, I., Moreira, J., Rigo, S., Araujo, G., and Wanner, L. (2026). Isa-based gemm acceleration: Maximizing computational intensity for scalable performance. *IEEE Access*, 14:64596–64607.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323.

Lee, J. K. L., Jamieson, M., Brown, N., and Jesus, R. (2023). Test-Driving RISC-V Vector Hardware for HPC.

Piarulli, L. and Sensi, D. D. (2026). Stencil computations on tenstorrent wormhole.

Pirova, A., Vodeneeva, A., Kovalev, K., Ustinov, A., Kozinov, E., Liniov, A., Volokitin, V., and Meyerov, I. (2026). Performance Optimization of BLAS Algorithms with Band Matrices for RISC-V Processors. *Future Generation Computer Systems*, 174:107936.

Pizzini Cavagna, H., Cesarini, D., and Bartolini, A. (2025). Assessing tenstorrent's risc-v matmul acceleration capabilities.

Taylor, M., Pearson, C., Berger-Vergiat, L., Long, G., and Ciesko, J. (2026). Numerical kernels on a spatial accelerator: A study of tenstorrent wormhole.

Tenstorrent (2025a). LLK Top-Level Overview. `https://github.com/tenstorrent/tt-llk/blob/main/docs/llk/l2/top_level_overview.md`. Accessed: 2026-05-21.

Tenstorrent (2025b). Tensix Neo. `https://tenstorrent.com/ip/tensix-neo`. Accessed: 2026-05-21.

Tenstorrent (2025c). TT-Metal. `https://github.com/tenstorrent/tt-metal`. Accessed: 2026-05-21.

Tenstorrent (2025d). TT-Metalium Guide. `https://github.com/tenstorrent/tt-metal/blob/main/METALIUM_GUIDE.md`. Accessed: 2026-05-21.

Tenstorrent (2025e). TT-Metalium Tiles Documentation. https://docs.tenstorrent.com/tt-metal/latest/tt-metalium/tt_metal/advanced_topics/tiles.html. Accessed: 2026-05-21.

Tenstorrent (2026). Matrix Engine Technical Report. https://github.com/tenstorrent/tt-metal/blob/main/tech_reports/matrix_engine/matrix_engine.md. Accessed: 2026-10-19.

Van Zee, F. G. and van de Geijn, R. A. (2015). BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33.