Characterizing Matrix Multiplication Units across General Parallel Patterns in Scientific Computing

Yuechen Lu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yuechen.lu@cup.edu.cn

Hongwei Zeng
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
hongwei@student.cup.edu.cn

Marc Casas
Barcelona Supercomputing Center
Barcelona, Spain
Universitat Politècnica de Catalunya
Barcelona, Spain
marc.casas@bsc.es

Weifeng Liu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

Abstract

Matrix multiplication units (MMUs) in modern parallel processors enable efficient execution of tiled matrix multiplications at varying precisions. While their effectiveness in AI workloads has been well demonstrated, their utility in scientific computing lacks systematic analysis. In this work, we characterize MMUs across a broad range of scientific computing patterns by evaluating performance, power consumption, numerical precision, and memory access behavior. To support this analysis, we develop Cubie, a comprehensive benchmark suite comprising ten MMU-optimized kernels of key parallel patterns. We also categorize MMU utilization patterns into four quadrants and identify the MMU limitations that arise in scientific computing. Through detailed comparisons with vector units, we provide nine key observations on the behavior and implications of MMUs in general scientific workloads, offering valuable insights for architecture, algorithm, and application researchers.

CCS Concepts: • **General and reference** → **Performance**; **Evaluation**; • **Computing methodologies** → **Parallel programming languages**.

Keywords: Matrix multiplication unit, Parallel pattern, Benchmark suite

ACM Reference Format:
Yuechen Lu, Hongwei Zeng, Marc Casas, and Weifeng Liu. 2026. Characterizing Matrix Multiplication Units across General Parallel Patterns in Scientific Computing. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3774934.3786456

PPoPP '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786456

1 Introduction

As computational demands have steadily increased over time, the evolution of compute units, ranging from scalar [16] to vector [49] architectures, has continuously advanced. Recently, matrix multiplication units (MMUs) processing general matrix-matrix multiplication (GEMM) have emerged as a key in significantly enhancing performance in modern processors. Representative MMUs include NVIDIA's Tensor Core [15], AMD's Matrix Core [78], Intel's XMX [34] and AMX [58], as well as ARM's SME [94] and Google TPU's systolic matrix multipliers [36]. The latest Top500 list [1] also reveals that the top ten supercomputers are all equipped with MMUs from NVIDIA, AMD, and Intel GPUs.

MMUs significantly outperform vector units when multiplying matrices in a variety of precisions [60–63], and its low-precision computation has been proven to be highly effective in deep learning [9, 24, 29, 30, 33, 35, 40, 42, 84, 90, 91]. But, although offering 2× double precision peak performance over vector units, their effectiveness in scientific computing is not yet well understood. This mainly arises from the diverse computational patterns in scientific workloads, which make MMU utilization more complex than in deep learning. Fortunately, recent efforts leveraged NVIDIA's tensor cores for stencil [11, 28, 43, 48, 102], FFT [23, 41, 76], reduction and scan [17], particle in cell [57], dense matrix factorization [39, 88, 101], general sparse matrix multiplication [44, 51, 53, 73, 80, 99], and breadth-first search [59], demonstrating improved performance of scientific kernels.

However, existing studies examine MMUs mainly in machine learning or isolated GEMM settings [21, 54, 78], and current GPU benchmark suites [4, 8, 18, 20, 85, 92] such as Rodinia [8] and SHOC [18] remain designed for vector based

execution without support for evaluating MMU based operations. There lacks a systematic way to analyze MMUs in general scientific workloads, which limits hardware architecture researchers, parallel algorithm researchers, and HPC application researchers in building a comprehensive understanding of MMU behavior.

For hardware architecture researchers, a key focus is designing future MMUs. This requires understanding (1) the compute pattern change of target applications to align with matrix multiplication, (2) the appropriate memory bandwidth to avoid underuse or waste, and (3) the impact on power consumption before and after adopting MMUs.

For parallel algorithm researchers, it is crucial to design MMU-optimized methods for non-GEMM computations. This requires understanding (1) the utilization patterns of MMUs that can be more performant, (2) whether MMU algorithms ensure the performance portability across GPU generations, and (3) the impact of numerical error from MMUs.

For HPC application researchers, the primary concern is whether MMUs deliver reliable benefits in real-world deployments. This requires understanding (1) whether applications can be performance portable, (2) the power and energy efficiency of MMUs in practical simulations, and (3) the numerical stability of MMUs to meet precision requirements.

Table 1. Mapping research questions and observations to architecture, algorithm, and application researchers.

Concerns	Arch.	Alg.	App.	Observation
Compute Patterns	✓	✓		O1, O2
Performance Portability		✓	✓	O3
Necessity of MMUs	✓	✓		O4, O5
Power and Energy	✓		✓	O6
Numerical Precision	✓	✓	✓	O7
Memory	✓	✓		O8
Workload Diversity	✓		✓	O9

To address these questions, we firstly propose Cubie, a benchmark suite composed of ten MMU-optimized workloads that align with the Berkeley Dwarfs [2, 3] and Exascale Computing Project application motifs [81]. Then, based on the input and output matrix integrity and reuse in MMU, we categorize the MMU utilization patterns in scientific computing kernels into four quadrants. For analyzing the MMU behavior, for each workload, we implement three to four variants, including versions based on vector units and MMUs. By comparing these implementations, we assess the performance effectiveness and portability, analyze their impact on energy efficiency, quantify numerical errors at FP64 precision, and examine bandwidth utilization.

We employ NVIDIA tensor cores as a representative MMU and perform our evaluation on NVIDIA Ampere A100, Hopper H200, and Blackwell B200 GPUs. This setting is chosen because NVIDIA provides a well-defined and widely used MMU programming interface [72], and tensor cores have served as the basis for most prior work [17, 41, 51, 53, 102] on MMU-accelerated scientific computing. Moreover, despite vendor-specific details, different MMUs across architectures share a consistent MMA-based abstraction and cooperative execution model; therefore, the characterization and observations derived from tensor cores are able to generalize to MMU-style matrix engines.

Our evaluation yields nine key observations on the challenges and opportunities of using MMUs in scientific computing. We find that MMU-accelerated kernels often deliver higher performance and lower energy consumption than their vector-based counterparts across a wide range of workloads and GPU architectures. These gains, however, come at the cost of data structure and algorithm changes, which may in turn introduce redundant computations and numerical error. Nevertheless, by systematically analyzing performance, power and energy, numerical accuracy, and memory behavior, we offer valuable insights for researchers in architecture, algorithm design, and application development.

This work makes the following contributions:

- We propose Cubie, a benchmark suite of ten open-source MMU workloads in scientific computing, offering high diversity and open accessibility (Section 3).
- We categorize the MMU utilization patterns in scientific computing kernels into four quadrants based on the input/output matrix integrity and reuse (Section 4).
- We characterize MMUs across general parallel patterns in scientific computing, quantifying performance, power, precision, and memory access (Section 5-9).
- We obtain nine key observations of using MMUs, providing valuable insights for architecture, algorithm, and application researchers (Section 11).

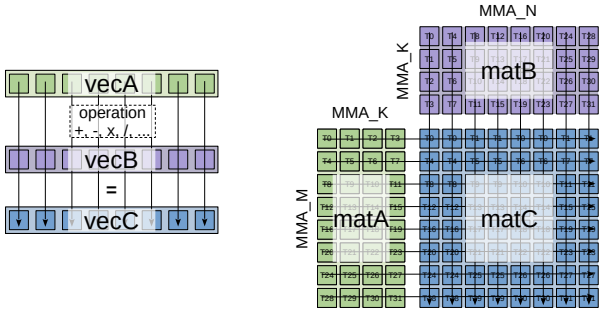
2 Background of MMU

Over the past decades, processor architectures have been moving from relying solely on scalar [16] and vector [49] execution toward treating matrix computation as an explicit hardware target. Figure 1 contrasts the execution pattern of a conventional vector unit in Figure 1a with the matrix-level computation layout of an MMU in Figure 1b.

From the programming viewpoint, MMUs expose a common abstraction as matrix multiply–accumulate (MMA) operations on fixed matrices. Algorithm 1 sketches a warp-level GEMM that calls an MMA instruction, showing how loads (line 6), MMA execution (line 7), and stores (line 8) are structured around this interface. Each MMA instruction defines the matrix shape, fragment layout, and cooperative execution group that holds operand and accumulator fragments in registers. Correspondingly, Figure 1b illustrates an FP64 MMA instruction, where a warp collectively owns the A, B and C matrices across its 32 threads.

Table 2. Workloads in the Cubie benchmark suite with their basic information, test cases, and comparison baselines.

Kernel	Ref.	Five Test Cases	Baseline
GEMM	cudaSample [68]	M*N*K: 256*256*256, 512*512*512, 1K*1K*1K, 2K*2K*2K, 4K*4K*4K	cudaSample [68] matrixMul v12.8
PiC	PiCTC [57]	N: 64K, 128K, 256K, 512K, 1M	-
FFT	tcFFT [41]	Sizes: 256*256, 256*512, 256*1K, 512*256, 512*512; Batch: 2K	cuFFT [66] v12.8
Stencil	LoRaStencil [102]	star2d1r: 1K*1K, 5K*5K, 10K*10K; star3d1r: 512*512, 1K*1K	DRStencil [98]
Scan	TCU-Scan [17]	Size: 64, 128, 256, 512, 1024	CUB [64] BlockScan v2.7.0
Reduction	TCU-Reduction [17]	Size: 64, 128, 256, 512, 1024	CUB [64] BlockReduce v2.7.0
BFS	BerryBees [59]	Five real-world graphs from SuiteSparse [19], see Table 3	Gunrock [89]
GEMV	-	M*N: 4K*16, 4K*32, 11K*16, 32K*16, 40K*16	cuBLAS [65] GEMV v12.8
SpMV	DASP [51]	Five real-world sparse matrices from SuiteSparse [19], see Table 4	cuSPARSE [67] SpMV v12.8
SpGEMM	AmgT-SpGEMM [53]	Five real-world sparse matrices from SuiteSparse [19], see Table 4	cuSPARSE [67] SpGEMM v12.8



(a) Vector unit computation layout with SIMD-style operations across vector lanes.

(b) MMU computation layout with tile-level MMA and cooperative fragment mapping.

Figure 1. Computation layouts of vector unit and MMU.

Algorithm 1 Warp-level GEMM using `FP64_m8n8k4_mma`

Input: Matrices $A \in \mathbb{R}^{8\times4}$ and $B \in \mathbb{R}^{4\times8}$
Output: Matrix $C \in \mathbb{R}^{8\times8}$
$t \leftarrow lane\_id$ ▷ Define the thread index in the warp
double $a, b, c[2]$ ▷ Allocate registers $a, b, c[2]$
$c[0] \leftarrow 0, c[1] \leftarrow 0$ ▷ Init register $c[2]$
$a, b \leftarrow$ LoadMatrixElements(A, B, t) ▷ Load A and B from GMEM/SMEM
FP64_M8N8K4_MMA(c, a, b) ▷ Call an MMA instruction
StoreMatrixElements(C, t, c) ▷ Store C to GMEM/SMEM

At the architectural level, MMUs have become increasingly capable [74, 79, 86] and widely deployed as dedicated matrix execution paths in modern processors. NVIDIA tensor cores illustrate this progression across GPU generations (*e.g.*, Volta [12], Turing [7], Ampere [14], Hopper [13], and Blackwell [87]) through expanded precision support and richer instruction interfaces. AMD provides matrix cores [78, 83] exposed through wavefront-level MFMA instructions, while Intel offers matrix extensions on both CPUs (AMX [58]) and GPUs (XMX [34]). Domain-specific tensor accelerators such as TPU [36] and NPU [10] also expose MMA-style primitives as first-class operators. Beyond these general MMUs, specialized designs [32, 45, 46, 77, 93] target sparse and irregular computation with sparsity-aware representations and dataflows, reflecting a trend toward more flexible MMUs.

3 The Cubie Benchmark Suite

To systematically characterize MMUs, we propose the Cubie benchmark suite. Cubie comprises ten open-source MMU-accelerated kernels selected for scientific computing, including GEMV, GEMM [68], SpMV [51], SpGEMM [53], FFT [41], stencil computations [102], reduction [17], scan [17], BFS [59], and PiC [57]. Detailed information on these workloads is provided in Table 2. Except for BFS, all kernels in Cubie perform floating-point computations using tensor core 64-bit MMA instructions.

General Matrix Multiplication (GEMM) computes the product of two dense matrices, resulting in another dense matrix. Cubie incorporates the routine `dmmaTensorCoreGEMM` from CUDA Samples [68], where each thread block processes a 64-by-64 tile using the FP64 `wmma_m8n8k4` instruction.

Particle in Cell (PiC) simulates the behavior of charged particles in a plasma or electromagnetic field. In Cubie, we adapt the FP16 PiCTC [57] to an FP64 version, employing the Boris push method [6] to simplify computation and mapping data into small blocks of size 8-by-4 and 4-by-8 for tensor core acceleration.

Fast Fourier Transform (FFT) converts time-domain signals into frequency-domain. Cubie adapts tcFFT [41] by converting FP16 to FP64, mapping data into 8-by-4 and 4-by-8 blocks, and leveraging tensor cores to perform complex matrix multiplications and element-wise computations.

Stencil Computation updates values of a grid using their neighbors on structured grid. In Cubie, the stencil computation follows LoRAStencil [102] in FP64. It decomposes stencil weight matrices into small components to use tensor cores, enabling memory-efficient data gathering and reducing computation. This transformation underlies our Observation 1.

Scan computes the prefix sum of an array. Cubie adapts the FP16 segmented scan of Dakkak et al. [17] to FP64. This method represents the input array as 8-by-8 blocks and multiplies the input with three different constant matrices to get

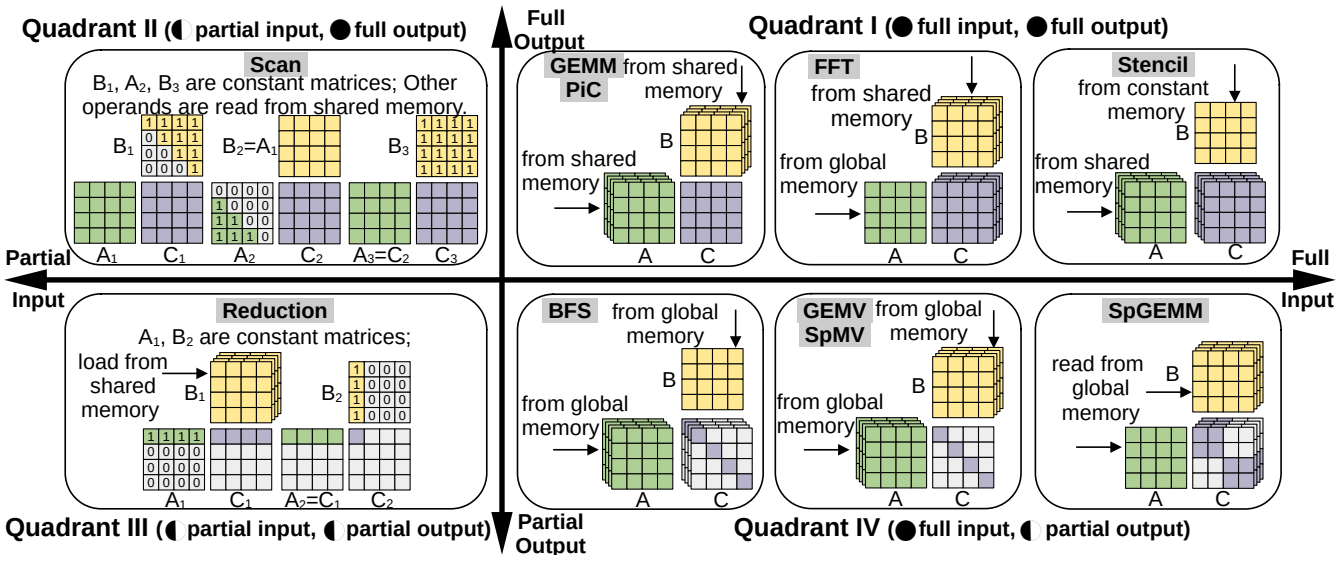


Figure 2. A categorization of workloads into four quadrants based on their utilization of tensor cores' input and output patterns. Quadrant I represents workloads with full input and output, *e.g.*, GEMM, PiC, FFT, and Stencil. Quadrant II includes workloads with partial input but full output, *e.g.*, Scan. Quadrant III consists of workloads with partial input and partial output, *e.g.*, Reduction. Quadrant IV covers workloads with full input but partial output, *e.g.*, BFS, GEMV, SpMV, and SpGEMM.

the row-wise prefix sums, column-wise prefix sums, and the final result. This transformation leads to Observation 1.

Reduction calculates the sum of all values in an array. Cubie incorporates the segmented reduction proposed by Dakkak et al. [17] and reproduces it from FP16 computations to FP64 computations. This approach stores the input array into several blocks of size 8-by-8, performs multiple multiplications with two constant matrices, and obtains the final sum, which also leads to the Observation 1.

Table 3. The Graphs evaluated in BFS.

Graph	#Vertices	#Edges	Group
wikipedia-20070206	3,566,907	90,043,704	Gleich
mycielskian17	98,303	100,245,742	Mycielski
wb-edu	9,845,725	112,468,163	SNAP
kron_g500-logn21	2,097,152	182,082,942	DIMACS10
com-Orkut	3,072,441	234,370,166	SNAP

Breadth-First Search (BFS) explores reachable vertices from a given source in a graph and is widely applied in combinatorial scientific computations [37, 56]. Cubie integrates BerryBees [59], which represents a graph in an 8-by-128 bitmap block slice-set format and executes bit operations using the single-bit `mma_m8n8k128` instruction of tensor cores. The tailored structure supports our Observation 1.

General Matrix-Vector Multiplication (GEMV) multiplies a dense matrix A with a dense vector x to produce a vector y. Our implementation partitions matrix A into blocks, broadcasts the corresponding vector x into blocks, calls the FP64 `mma_m8n8k4` instruction to perform matrix multiplication on tensor cores, and extracts the diagonal elements from the output matrix.

Table 4. The matrices evaluated in SpMV and SpGEMM.

Matrix	#Rows	#Nonzeros	Group
spmsrtls	29,995	229,947	GHS_indef
Chevron1	37,365	330,633	Chevron
raefsky3	21,200	1,488,768	Simon
conf5_4-8x8-10	49,152	1,916,928	QCD
bcsstk39	46,772	2,089,294	Boeing

Sparse Matrix-Vector Multiplication (SpMV) multiplies a sparse matrix with a dense vector, producing a vector. Cubie implements this kernel using DASP [51] and adopts its FP64 version. DASP groups the rows of the input matrix into three categories and organizes them into small blocks of size 8-by-4 to use tensor cores. This restructuring for MMU usage underlies our Observation 1.

Sparse General Matrix-Matrix Multiplication (SpGEMM) multiplies two sparse matrices to generate a sparse matrix. Cubie uses the FP64 SpGEMM kernel from AmgT [53], which partitions sparse matrices in the mBSR format, forming 4-by-4 blocks and combining them into blocks of size 8-by-4 to leverage tensor cores, which also motivates the Observation 1.

Key Observation 1: To exploit MMUs, non-GEMM algorithms in scientific computing often have to modify data structures and reorganize algorithms.

4 Categorization of MMU Utilization Patterns

We introduce a systematic categorization to characterize how different workloads leverage the MMA pattern. To properly run on MMUs, algorithms must undergo data preprocessing or algorithm reorganizing to transform non-GEMM operations into computational patterns compatible with MMA instructions, leading to a variety of MMU utilization patterns. We classify workloads along two dimensions: input matrix utilization and output matrix utilization, each categorized as either full or partial. The four groups are illustrated in the coordinate system in Figure 2.

Cubie Quadrant I (●[1], ●) includes workloads such as Stencil, FFT, GEMM, and PiC, which fully utilize both input and output matrices. The key distinction among these workloads lies in which component is reused. As shown in Quadrant I in Figure 2, GEMM and PiC repeatedly load inputs to accumulate into one result matrix, Stencil loads matrix B only once from constant memory for reuse, and FFT loads matrix A only once from global memory for multiple uses and products multiple resulting matrices.

Cubie Quadrant II (◐[2], ●), containing the Scan kernel, which uses the constant matrix consisting of values zero and one as one of the input matrices and fully utilizes all elements of the output matrix. Specifically, the Scan is completed using three consecutive types of MMA operations, where one of the operands in each MMA is a constant matrix: (1) an upper triangular matrix of ones, (2) a lower triangular matrix of ones, or (3) a matrix entirely filled with ones, *i.e.*, B_1, A_2, and B_3 in Figure 2 Quadrant II. The constant matrices do not require loading from global memory.

Cubie Quadrant III (◐, ◐) contains the Reduction kernel, which exhibits partial utilization of both input and output matrices. Similar to the Scan, this kernel also uses constant matrices as one of the operands. These constant matrices typically have a single row or column filled with ones, while the remaining elements are zeros, *i.e.*, A_1 and B_2 in Figure 2 Quadrant III. Unlike the Scan, this kernel utilizes only a small portion of the output matrix, specifically a single row or even a single element, for the final result.

Cubie Quadrant IV (●, ◐), including four kernels BFS, GEMV, SpMV and SpGEMM, takes full input A and B but only partial output C. Among the four kernels shown in Figure 2 Quadrant IV, (1) BFS reuses one B, loads multiple A, and extracts the diagonal elements of multiple C, (2) GEMV and SpMV repeatedly load A and B and accumulate results also in the diagonal elements of one C, and (3) SpGEMM reuses A, loads multiple B, and accumulates results into the diagonal tiles of C, achieving slightly higher utilization.

Key Observation 2: Scientific kernels may not fully utilize the dense input and output matrices of MMUs, exhibiting distinct utilization patterns in four quadrants characterized by varying levels of density.

The utilization patterns in this section are derived from MMU adapted kernels and characterize MMU behavior in the transformed code space. A deeper question is whether MMU accelerability can be inferred from the original algorithm or a CUDA core implementation before such transformations. Addressing this question requires linking algorithmic structure to MMU execution semantics, likely with compiler assistance [25, 95, 97]. Our categorization provides a first step toward the algorithm level reasoning about MMU suitability.

5 Experimental Design

We take the tensor core as a representative MMU to conduct our evaluation. This section presents the experimental setup and the algorithmic variants for all workloads, which serve as the basis for analyzing MMU behaviors in later sections.

5.1 Experimental Setup

We evaluate Cubie on NVIDIA A100 (Ampere), H200 (in the GH200 platform, Hopper), and B200 (Blackwell) GPUs, see Table 5 for details. The CUDA version we used is v12.8. Prior work has shown that GPUs of the same type may exhibit non-negligible performance variation due to manufacturing effects and process variation [82]. To avoid such device-level variability obscuring the performance trends, all measurements in this study are conducted on a single physical GPU for each GPU type. This design choice allows us to more clearly attribute observed performance differences to hardware features and execution characteristics, rather than cross-device variation.

Table 5. The specifications of the three GPUs tested.

NVIDIA GPUs	FP64 Units	Peak Performance
A100 (Ampere) PCIe 40 GB, 1.55 TB/s	Tensor Core	19.5 TFLOPs
	CUDA Core	9.7 TFLOPs
H200 (Hopper) SXM 96 GB, 4 TB/s	Tensor Core	66.9 TFLOPs
	CUDA Core	33.5 TFLOPs
B200 (Blackwell) SXM 180 GB, 8 TB/s	Tensor Core	40.0 TFLOPs
	CUDA Core	40.0 TFLOPs

[1]Symbol ●indicates the algorithm utilizes the entire input or output matrix.
[2]Symbol ◐signifies the algorithm only utilizes a portion of the input or output matrix, such as when the input is a zero matrix or only the diagonal of the output is meaningful.

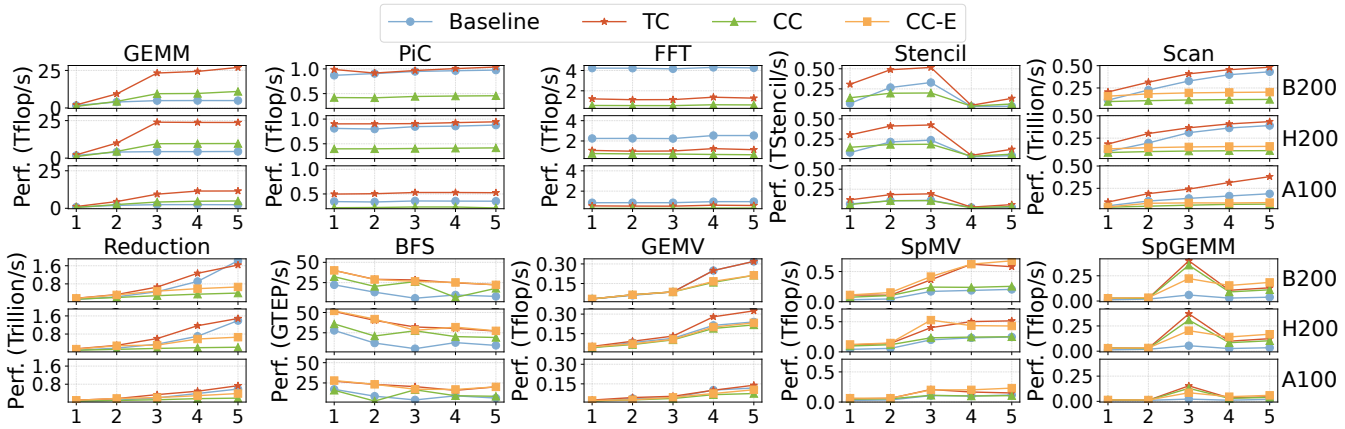


Figure 3. Performance comparison of baselines, TC, CC, and CC-E implementations for all workloads on the three GPUs.

Test Cases: In Cubie, each workload is evaluated using five test cases, as shown in Table 2. These cases span small to large problem scales and cover the major GPU performance regimes. A more detailed representativeness analysis will be presented in Section 10.

Baselines: For each workload, the TC version is compared against a baseline from CUDA official libraries or the methods presented in the original studies. The CUDA libraries include cuBLAS [65] for GEMV, cudaSample [68] for GEMM, cuSPARSE [67] for SpMV and SpGEMM, cuFFT [66] for FFT, and CUB [64] for Reduction and Scan. Gunrock [89] and DRStencil [98] are used for BFS and Stencil, respectively.

5.2 Algorithmic Implementation Variants

To analyze the changes in program performance, energy efficiency, and computational characteristics, and to determine whether these changes are driven by the use of MMUs or the design of the algorithm, we consider three variants.

Tensor Core Version (TC): This algorithmic variant consists of programs that perform floating-point computations mainly using tensor core MMA 64-bit instructions. The specific implementation techniques for each workload are described in Section 3. This algorithmic variant serves as a reference for assessing the efficiency of tensor core-based computation across various workloads.

CUDA Core MMA Replacement (CC): This variant replaces tensor core MMA operations with CUDA core-based computations while maintaining identical data structures and algorithmic settings, *i.e.*, this version implements the exact same algorithm as TC but using CUDA core instructions instead of tensor core MMAs. Specifically, in an FP64 tensor core MMA instruction `mma_m8n8k4`, each thread within a warp processes a specific subset of matrix elements. Our CC implementation preserves same thread responsibilities and data layouts, which enables a direct comparison between tensor core and CUDA core executions.

CUDA Core Essential Replacement (CC-E): This version eliminates redundant or useless operations introduced by tensor core MMAs, preserving only the essential computations using CUDA cores. For workloads that do not fully use the whole MMA pattern, expressing them in MMA patterns introduces redundant computation. In such cases, we replace tensor core MMA instructions with CUDA core instructions that execute only the mathematically necessary operations to ensure correctness. For example, in the case of the GEMV kernel $y = A \cdot x$, the CC variant actually computes $A \cdot [x, ..., x]$, where the second operand is a matrix generated by replicating the x vector several times and takes the output diagonal to get y, since CC must express GEMV in an MMA-like operation. In contrast, the CC-E version of GEMV indeed computes $y = A \cdot x$ without any redundant operation, and the similar principle applies to BFS, SpMV, and SpGEMM. Comparing CC-E with TC makes it possible to observe changes in various characteristics, distinguishing the effects of tensor core-specific optimizations from the underlying hardware. For the GEMM, PiC, FFT, and Stencil kernels, the CC-E version is equivalent to CC.

6 Performance of MMUs

In this section, we conduct three performance evaluations to analyze the suitability of tensor core acceleration for the scientific workloads that Section 3 describes. For each workload, we consider the three variants described in Section 5.2 and its corresponding baseline. Section 6.1 evaluates the performance improvements of TC versions over their corresponding baselines, focusing on whether these advantages hold consistently across different GPU architectures. Section 6.2 compares the performance of the TC and CC implementations to evaluate the performance acceleration of MMUs over vector units when using identical data structures and algorithms. Section 6.3 evaluates the CC-E versions to investigate whether the redundant computations introduced

by transformations for MMU utilization are worthwhile for scientific computing kernels.

Figure 3 shows several subplots indicating in their y-axis the performance for all workloads across their four implementations: baseline, TC, CC, and CC-E. The subplots represent in their x-axis the five test cases per workload that Table 2 specifies. For each workload, the three subplots from bottom to top are the performance on A100, H200, and B200 GPUs, respectively. For stability, most benchmarks start with 100 warm-up runs followed by 1000 timed executions, with the arithmetic average performance reported.

6.1 Comparison of Baseline Against TC

To evaluate the performance benefits of using tensor core acceleration compared to standard CUDA libraries, we consider the TC versions and compare them against their corresponding baselines, which are listed in the fourth column of Table 2. Figure 3 represents the performance of the baselines and TC versions in terms of blue 'circle' and red 'star' markers, respectively. Figure 4 summarizes the speedups of TC versions over baselines for each workload, where each value is averaged across the five representative test cases.

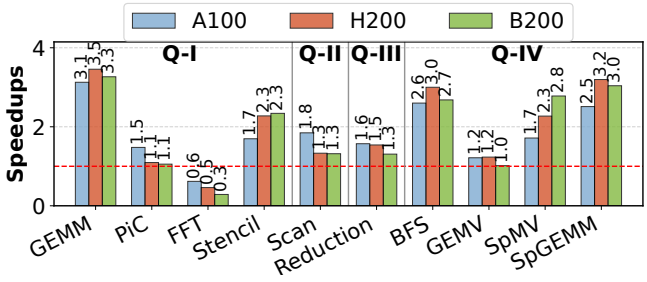


Figure 4. Speedups of TC implementations compared to their baselines on the three GPUs across all workloads, grouped by utilization patterns (Quadrants I-IV).

For workloads in **Quadrant I**, which fully use both the input and output registers of MMA instructions, TC versions are expected to deliver consistent and portable performance gains across hardware generations. GEMM and Stencil experience strong acceleration when using tensor cores for computation. In contrast, PiC and FFT show reduced benefits, with FFT in particular performs worse than the cuFFT baseline. The poor performance of FFT is explained by the difficulty to express the FFT-specific butterfly computation patterns [100] in terms of MMA instructions.

The **Quadrant II** workload, Scan, performs MMA instructions using constant matrices as one operand, which reduces data transfer overhead and significantly improves tensor core utilization. Therefore, its TC version outperforms the baseline across all the three GPUs, achieving speedups of 1.8×, 1.3×, and 1.3× on A100, H200, and B200, respectively.

The **Quadrant III** kernel, Reduction, uses constant matrices, similarly as Scan, but only accesses a single row or element from each 8-by-8 output tile. While tensor core acceleration benefits from using constant operands, as it increases register reuse, the low arithmetic intensity of the Reduction workload mitigates the performance benefits of using tensor cores. As a result, the TC version achieves 1.3-1.6× speedups over the baseline on the three GPUs.

The **Quadrant IV** kernels are all memory-bound and, thus, they strongly benefit from high memory bandwidth, such as H200 and B200. The considered BFS algorithm [59] judiciously leverages tensor core bit-wise operations and efficient data structures with low memory footprint to achieve 2.6×, 3.0×, and 2.7× speedups over the baseline on A100, H200, and B200, respectively. SpGEMM successfully leverages half of the 8-by-8 output tiles of MMA, resulting in 2.5-3.2× speedups over cuSPARSE on the three GPUs. While B200's FP64 tensor core throughput is lower than H200's, its superior 8 TB/s memory bandwidth enables competitive or even better performance for memory-bound workloads.

> **Key Observation 3:** MMU-accelerated workloads consistently outperform vector baselines in most cases, and exhibit performance portability across the Ampere, Hopper, and Blackwell architectures.

6.2 Comparison of CC Against TC

This section conducts an ablation study by comparing the CC and TC implementations to evaluate the performance benefit of tensor core acceleration under constant data structures and algorithms. To isolate the effect of the compute unit, each MMA instruction in the TC versions is replaced with a semantically equivalent CUDA core instruction in the CC variants. Figure 3 represents the performance of the CC versions with green 'triangle' markers. Figure 5 summarizes the average speedups of CC over TC for each workload on A100, H200, and B200. Specifically:

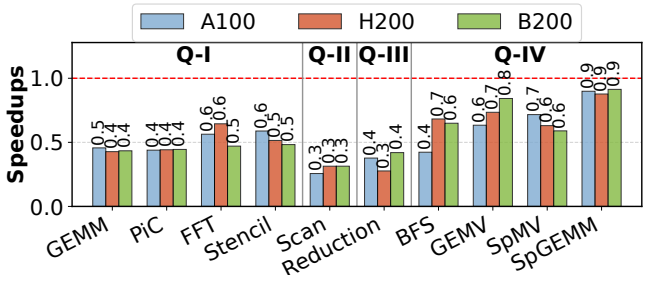


Figure 5. Speedups of CC replacements over TC versions on the three GPUs across kernels in Quadrants I-IV.

The **Quadrant I** workloads have high MMA computation density and fully use the MMA pattern, as Figure 2 illustrates. The performance of the CC versions generally drops around 50% of that achieved by the TC counterparts, which aligns

with our expectations. As Figure 2 illustrates, GEMM and PiC benefit from more efficient data movement than FFT and Stencil, and achieve higher tensor core utilization. Consequently, their CC versions experience larger slowdowns than FFT and Stencil. For instance, the PiC CC implementation only achieves a 0.4× speedup of its TC version. FFT suffers the smallest degradation within Quadrant I since its TC version does not exploit the tensor core performance, as Section 6.1 explains.

For workloads in **Quadrants II** and **III**, including Scan and Reduction, their CC versions perform noticeably worse than the TC counterparts. Specifically, the CC versions of Scan and Reduction deliver less than 40% of the performance achieved by their TC counterparts. This gap exceeds the ratio between the peak performances of the tensor and CUDA cores. Besides the lower floating-point performance of CUDA cores, the additional degradation comes from the fact that the TC versions of Scan and Reduction benefit from using constant matrices as operands, and CUDA cores do not leverage these constant operands as much as tensor cores.

In **Quadrant IV**, although the kernels are memory-bound, the CC variants still perform worse than the TC versions, with relatively small performance gaps. For example, in the case of SpMV, the CC versions retain 60-70% of the TC performance on the three GPUs. Consistent with prior studies [5, 96] showing that effective use of vector units improves SpMV performance, MMUs can provide further improvements for memory-bound kernels.

> **Key Observation 4:** Removing the impact of data structures and algorithms (replacing MMU instructions with equivalent vector unit operations), MMUs account for 10% to 200% of the performance gains.

6.3 Comparison of CC-E Against TC

To use MMUs, scientific kernels in Quadrants II-IV are transformed into GEMM-like forms at the cost of redundant computations. For example, in SpMV, the input is divided into small blocks, and full MMA operations are performed among them, but only the diagonal elements of the outputs are ultimately used. To evaluate whether such redundant computations are worthwhile, we compare CC-E variants (recall the description in Section 5.2), which retain only essential computations on CUDA cores, against TC versions. Figure 6 summarizes the average speedups of CC-E (recall the yellow 'square' markers in Figure 3) over TC across the three GPUs. The CC-E versions of workloads belonging to Quadrant I are equivalent to the CC variants, so Figure 6 only lists results of Quadrants II-IV.

In **Quadrants II** and **III**, the CC-E versions of Scan and Reduction consistently underperform their TC counterparts, with speedups ranging from 0.34–0.45× and 0.66–0.79× across A100, H200, and B200, respectively. This is because, when

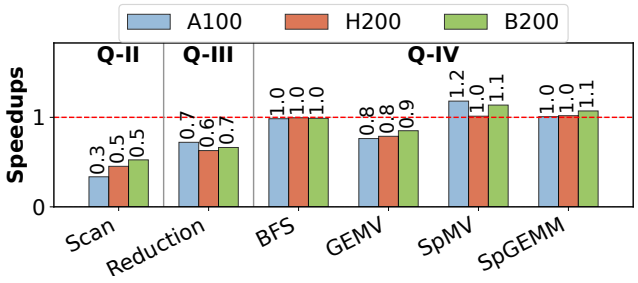


Figure 6. Speedups of CC-E replacements over TC versions on the three GPUs across kernels in Quadrants II-IV.

processing small blocks, partial and irregular computations are less efficient than tensor cores' full and regular computation patterns. Despite the redundant computations introduced by MMU use, the TC versions still outperform both the CC-E and baselines, meaning that the overhead of redundant computations is worthwhile in these two kernels.

In **Quadrant IV**, the performance of CC-E shows variation across workloads. For SpMV, the CC-E versions outperform the TC by 1.0-1.2× speedups on the three GPUs, considering that TC is faster than the baseline by a factor of 1.7-2.8× (recall Section 6.1), removing redundant computations introduced for MMU further improves performance over baseline. This suggests that while using MMUs incurs redundant computations for SpMV, the changes of data structures and algorithmic workflows made to enable MMU use remain beneficial. In contrast, GEMV's CC-E is slightly slower than its TC version, and CC-E and TC of SpGEMM and BFS exhibit similar performance, showing that removing the redundant computations in general does not bring performance gains.

> **Key Observation 5:** Generally, the redundant computations introduced to enable MMU-friendly matrix computing patterns should not be removed. The only exception is SpMV, where avoiding the redundancy yields up to 20% higher performance.

7 Power and Energy Efficiency of MMUs

This section evaluates the impact of tensor core usage on the GPU power and energy efficiency. We monitor the workloads power using the NVIDIA Management Library [71] (`nvmlDeviceGetPowerUsage()`). During each test, a monitoring process logs both timestamps and power values from kernel launch to completion. For brevity, we show the power and energy efficiency results only for the H200 GPU, which has a thermal design power of 750W and the highest theoretical FP64 tensor core throughput among the three GPUs.

Figure 8 illustrates the power consumption (y-axis) over runtime (x-axis) for each workload, while the area under

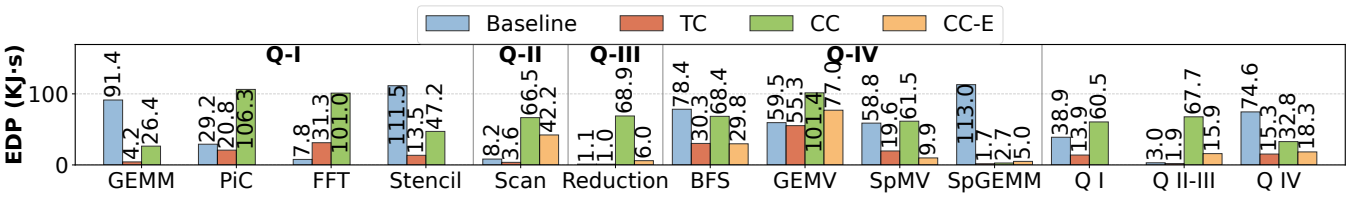


Figure 7. The EDP comparison of baselines, TC, CC, and CC-E implementations for all workloads and the geomean within each quadrant on H200. Each of the ten workloads is executed 500, 60, 400, 5K, 25K, 50K, 2K, 6M, 1M, and 5K times, respectively.

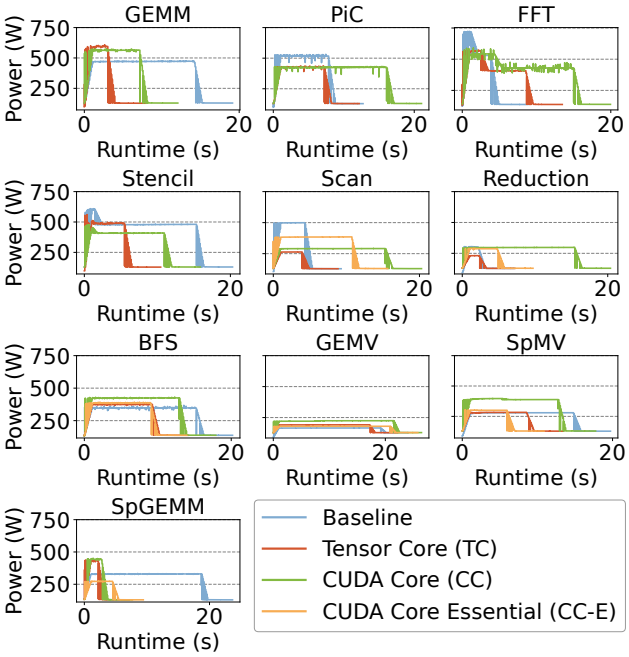


Figure 8. Power consumption over time of baselines, TC, CC, and CC-E implementations for all workloads on H200.

each power-runtime curve reflects the total energy consumption per workload. In addition, to measure the balance between energy efficiency and performance, we compute the energy-delay product (EDP) [26, 27] for each test case, as $EDP = Average\ Power \times Execution\ Time^2$. The EDP results are summarized in Figure 7.

The **Quadrant I** workloads, GEMM, FFT, Stencil, and PiC, fully utilize the register operands of MMA instructions, reflecting efficient use of tensor cores. Therefore, the TC implementations typically exhibit high instantaneous power consumption, often exceeding 400W. However, their much shorter execution times lead to lower overall energy usage and better EDP. For example, in Stencil, the TC version completes execution in 5.5s at an average power of 450W, while the baseline takes 15s at 470W. This leads to a 65% reduction in energy and an 88% EDP reduction. Based on the geomean EDP within Quadrant I, the TC version reduces EDP by approximately 64% compared to the baseline.

The **Quadrants II** and **III** workloads, Scan and Reduction, are composed of lightweight and regular computations, which drive stable power curves throughout their execution. For both kernels, the TC implementations consistently achieve lower power consumption and the shortest execution times, resulting in the lowest overall EDP. For example, in Scan, the TC version runs in 3.8 seconds at an average power of 244W, leading to an EDP of 3.63 kJ·s. In contrast, the baseline consumes over 300W and reaches an EDP of 8.24 kJ·s, meaning that the TC reduces EDP by over 55%. For Quadrants II and III, the TC implementation achieves a 36% reduction in geomean EDP relative to the baseline.

The **Quadrant IV** kernels, including BFS, GEMV, SpMV, and SpGEMM, are primarily memory-bound. Due to frequent memory accesses, their baselines show low CUDA core utilization, and have relatively low power consumption but longer execution times. In contrast, the TC and its variants improve memory accessing through regularized data layouts, leading to higher core utilization. These implementations consume similar power but complete execution much faster, resulting in lower overall energy usage and EDP. For example, in BFS, the TC and CC-E versions consume around 375W, compared to 340W for the baseline. However, both reduce execution time by about 40%, leading to over 60% lower EDP. For Quadrant IV, the TC version lowers the geomean EDP by nearly 80% compared to the baseline.

Key Observation 6: MMUs exhibit similar power consumption to vector units but complete computations significantly faster, resulting in 30% to 80% lower geomean EDP across all workloads.

8 Floating-point Accuracy of MMUs

This section compares the FP64 accuracy of tensor cores with CUDA cores. Specifically, we compare outputs obtained on H200 and B200 GPUs using both kinds of cores against a naive CPU serial implementation (*e.g.*, CSR-based SpMV), which serves as the ground truth in terms of floating-point accuracy. For input data initialization, SpMV and SpGEMM load matrix from the original dataset files and generate vector inputs using pseudo-random values; FFT and Stencil follow the initialization schemes in their original benchmark codes; and

Table 6. FP64 numerical errors of different implementations for all workloads on H200 and B200 GPUs, obtained against CPU serial computing results. Bold numbers indicate the lowest average error for each workload.

Workload	Errors on H200 GPU						Errors on B200 GPU					
	Baseline		TC/CC		CC-E		Baseline		TC/CC		CC-E	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
GEMV	5.19E-16	3.55E-15	**0**	0	4.69E-16	3.55E-15	6.30E-16	3.55E-15	**4.92E-16**	5.33E-15	6.07E-16	3.55E-15
GEMM	**4.36E-14**	3.69E-13	3.12E-13	1.82E-12	-	-	**5.22E-15**	4.97E-14	7.40E-15	1.14E-13	-	-
SpMV	2.15E-08	9.54E-07	**7.11E-10**	2.38E-07	2.02E-08	1.07E-06	2.10E-08	9.54E-07	**8.92E-09**	4.77E-07	2.09E-08	1.07E-06
SpGEMM	7.10E-16	7.11E-14	**6.30E-16**	8.53E-14	**6.30E-16**	8.53E-14	6.78E-16	7.11E-14	**6.55E-16**	8.53E-14	**6.55E-16**	8.53E-14
FFT	**4.83E-18**	1.22E-15	7.50E-17	2.77E-14	-	-	**5.00E-18**	1.22E-15	7.49E-17	2.77E-14	-	-
Stencil	**1.05E-16**	6.66E-16	8.77E-15	5.68E-14	-	-	**1.05E-16**	6.66E-16	5.84E-15	4.26E-14	-	-
Reduction	**1.82E-14**	5.68E-14	2.91E-14	8.53E-14	2.13E-14	5.33E-14	**1.82E-14**	5.68E-14	2.91E-14	8.53E-14	2.13E-14	5.33E-14
Scan	**9.53E-15**	5.68E-14	1.11E-14	8.17E-14	1.11E-14	8.17E-14	**9.53E-15**	5.68E-14	1.11E-14	8.17E-14	1.11E-14	8.17E-14
PiC	**0**	0	**0**	0	-	-	**2.52E-16**	2.22E-15	**2.52E-16**	2.22E-15	-	-

the remaining workloads use pseudo-random values. We generate pseudo-random values distributed within (-2, 2) using a linear congruential generator method [38], following the LINPACK benchmark [22]. The average and maximum errors are computed as $Average\_Error = \frac{1}{n}\sum_{i=1}^{n}\left|result_{gpu,i} - result_{cpu,i}\right|$ and $Max\_Error = \left|result_{gpu,i} - result_{cpu,i}\right|_{max}$, respectively. The n denotes the number of input samples in this comparison, which ranges from 1K to 100M depending on kernels.

Table 6 presents the FP64 numerical error results for all versions. BFS is excluded since it does not perform floating-point computations. We observe that **(i)** for each workload, the TC and CC versions produce identical errors. Since these two variants use the same data structures and algorithms and differ only in the compute unit, tensor core and CUDA core provide equivalent numerical accuracy for FP64 operations. **(ii)** For workloads in Quadrant IV, including GEMV, SpMV, and SpGEMM, the TC/CC versions produce small numeric errors, while CC-E can introduce deviations up to an order of magnitude larger (*e.g.*, SpMV). These differences suggest that the performance-driven optimizations of the CC-E versions may impact the floating-point computations and complicate the reproducibility of scientific results.

For workloads in other quadrants, the lowest deviations are typically achieved by the baselines. For example, in FFT, the baseline yields an average deviation of 4.83E-18 on H200 and 5.00E-18 on B200, while the TC/CC versions show an error of 7.50E-17 and 7.49E-17 on two GPUs, nearly an order of magnitude higher. These differences arise from variations in algorithmic structure and accumulation order, which affect how rounding errors propagate during computation.

Key Observation 7: MMUs and vector units provide comparable numerical accuracy, but algorithmic transformations for MMU utilization can induce significant numerical deviations that undermine the reproducibility of scientific results.

9 Performance Model

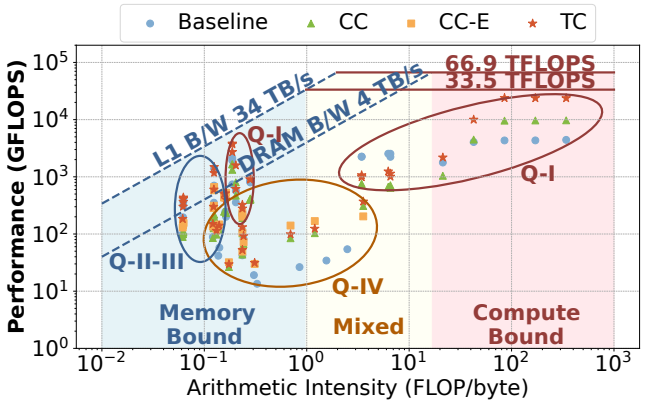


Figure 9. The cache-aware roofline model for Cubie, illustrating the performance characteristics of different implementations. The L1 cache bandwidth is computed as $BW_{L1} = N_{SM} \times N_{LSU} \times W_{access} \times f_{clock}$, and the DRAM bandwidth is derived from the whitepaper [63] of H200 GPU.

To better understand the interaction between computation and memory efficiency in Cubie workloads, we construct a cache-aware roofline model [31, 50, 55], as shown in Figure 9. This model incorporates ceilings for DRAM and L1 cache bandwidth, as well as FP64 compute throughput defined by the peak performance of tensor cores and CUDA cores. The plot includes all workloads and their implementations, except for BFS, which relies on bit-wise operations.

The workloads in **Quadrant I** exhibit varying levels of arithmetic intensity. GEMM and FFT have high intensity, while Stencil and PiC are lower. Although GEMM falls in the compute-bound region, its performance does not reach the tensor core peak of 66.9 TFLOPS. This is due to the absence of advanced optimizations such as those in cuBLAS [65] or CUTLASS [69], which are excluded from Cubie for simplicity. Nonetheless, the TC versions of these kernels still achieve

a clear performance advantage over the CC, demonstrating the effectiveness of tensor core acceleration.

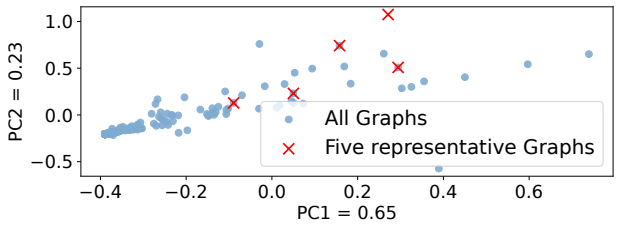
For workloads in **Quadrants II** and **III**, Reduction and Scan have low arithmetic intensities around 10^{-1}. With segmented processing and improved data locality, they are relatively cache-friendly. Thus, the TC versions exceed the DRAM bandwidth ceiling and achieve improved performance.

In **Quadrant IV**, workloads span arithmetic intensities from 10^{-1} to 3, and are considered memory-bound kernels. However, the performance of baselines does not approximate the bandwidth limit. In contrast, the TC versions approach the bandwidth limit more closely, and in some cases, the CC-E versions do as well, indicating that the adaptations for tensor core usage lead to more efficient memory access.

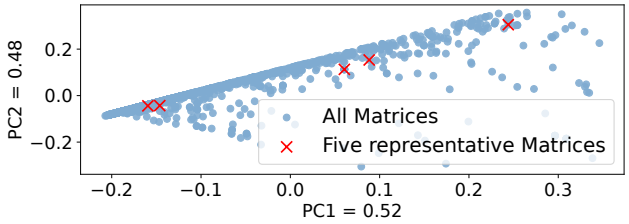
Key Observation 8: Adapting data layouts and algorithms for MMUs fundamentally alters memory access patterns, often yielding more regular access and significant performance gains.

10 Analysis of Benchmark Suite Coverage

To characterize the coverage of Cubie benchmark suite, we analyze both its input cases and workload composition.



(a) The PCA results of the 499 graphs in SuiteSparse and the five representative graphs used in BFS.



(b) The PCA results of the 2893 matrices in SuiteSparse and the five representative matrices used in SpMV and SpGEMM.

Figure 10. The PCA visualizations of the graphs and matrices considered in our experiments.

We standardize the structural features, including sparsity, row and column degree statistics, and block structures, and then apply principal component analysis (PCA) to capture the dominant variation patterns of matrices and graphs in the SuiteSparse Matrix Collection [19]. As shown in Figures 10a and 10b, the five selected matrices exhibit a dispersion of 0.18 compared to 0.05 among their nearest neighbors, and the five selected graphs cover 81–96% of the structural value ranges with 94.6% of all graphs lying close to at least one representative, demonstrating that both sets effectively span the major structural variations in their respective domains. These matrices and graphs are also widely used in prior graph algorithms [59] and sparse matrix studies [47, 51, 96].

Table 7. Comparison of Cubie with two existing benchmark suites (Rodinia and SHOC) based on the number of the Berkeley Dwarf covered and the features analyzed in each suite.

Dwarf / Feature	Rodinia [8]	SHOC [18]	Cubie (this work)
Dense linear algebra	3	2	2
Sparse linear algebra	-	-	2
Spectral methods	-	1	1
N-Body	-	1	1
Structured grids	4	1	1
Unstructured grids	2	-	-
MapReduce	-	3	2
Graph traversal	2	-	1
Dynamic programming	1	-	-
Parallelization pattern	✓		✓
Performance	✓	✓	✓
Power and energy	✓	✓	✓
Precision			✓
Memory bandwidth		✓	✓
CPU-GPU data transfer	✓	✓	

Additionally, we compare Cubie with Rodinia [8] and SHOC [18] in terms of the number of the Berkeley Dwarf [2, 3] computation patterns covered and the evaluated features, as summarized in Table 7. These two suites both cover five dwarfs and evaluate three or four key features. Compared with them, Cubie covers seven dwarfs by including one or two representative workloads for each dwarf and evaluates five key features, offering both broad pattern coverage and comprehensive feature assessment.

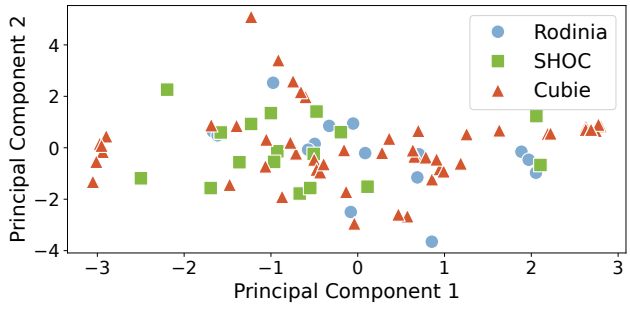


Figure 11. The PCA results comparing Rodinia, SHOC, and Cubie.

To further compare Cubie with existing benchmark suites, we perform PCA on key architectural metrics (memory efficiency, compute throughput, and instruction pipeline usage for FMA and tensor operations) collected using NCU [70], which provides a comprehensive description of the workload behavior. We execute kernels and applications from Rodinia, SHOC, and Cubie, using the datasets specified in the original papers [8, 18]. For each application, performance metrics are collected across the complete kernel execution to ensure an exhaustive coverage of the execution stages. Then, using the Python module Scikit-learn [75], the data is standardized, followed by applying PCA by computing the covariance matrix and extracting the two top principal components representing highest variance in workload behavior.

As shown in Figure 11, Cubie workloads span a wider area in the principal component space, reflecting a greater diversity in execution behavior than both Rodinia and SHOC. This broader dispersion demonstrates Cubie's ability to represent a wide range of patterns relevant to modern processor.

Key Observation 9: Originally developed with the primary goal of evaluating MMUs, the Cubie benchmark suite encompasses a wide range of behaviors in scientific programs, positioning it as an effective tool for assessing modern processors.

11 Conclusions

This paper has developed Cubie, a benchmark suite comprising optimized scientific kernels tailored for MMUs, and used it to characterize MMUs across scientific computing patterns in performance, power, precision, and memory layout. With its wide coverage of parallel patterns and computing characteristics, valuable insights for researchers in architecture, algorithm, and application have been provided.

For architecture design researchers, we confirmed the value of MMUs in scientific computing, in terms of performance (**O4**, **O5**) and energy (**O6**) advantages for most kernels. However, as some patterns only use part of the two input and one output matrices of MMUs (**O1**, **O2**), enabling architectural support for more flexible compute patterns will improve MMU applicability to a broader class of workloads.

For parallel algorithm researchers, MMUs indeed offer strong and portable performance across architectures (**O3**, **O4**, **O5**), making them more beneficial for scientific kernels. By changing data layouts and applying programming-level transformations to align with the four utilization patterns, non-GEMM kernels can also effectively use MMUs (**O1**, **O2**). Optimizing for higher arithmetic intensity and lower memory overhead will further enhance these gains (**O8**).

For application researchers, MMUs generally provide better and portable performance (**O3**), along with improved energy efficiency across diverse workloads (**O6**). While MMUs themselves do not introduce more numerical errors than the same implementations on vector units, the algorithmic choices made to exploit MMUs can affect precision, requiring users to exercise caution in their selection (**O7**).

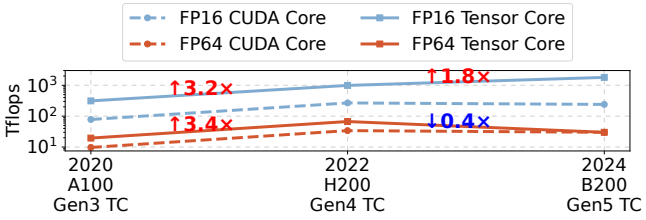


Figure 12. Peak throughput of NVIDIA's three latest GPU architectures (Ampere, Hopper, Blackwell), comparing FP16 and FP64 performance on CUDA cores and Tensor Cores.

We highlight an unexpected and concerning divergence in tensor core evolution, as shown in Figure 12: FP16 tensor core peak throughput continues to scale across generations, rising from 312 TFLOPS on Ampere to 989.5 TFLOPS on Hopper and 1800 TFLOPS on Blackwell, whereas FP64 tensor core peak throughput increases from 19.5 TFLOPS on Ampere to 67 TFLOPS on Hopper but, instead of sustaining this growth, falls to 30 TFLOPS in the latest Blackwell, which is less than half of the Hopper. This regression may directly undermine FP64 MMU adoption for scientific computing and should be viewed as a step backward for HPC capability. While the reduction may reflect a vendor perception that FP64 MMU support is of limited practical value, our results demonstrate that FP64 MMU acceleration can largely benefit most scientific workloads. Given our observations, future GPU roadmaps should preserve and materially strengthen FP64 MMU capability rather than treating it as a secondary feature, so that architectural gains translate into sustained progress for scientific computing applications.

Data Availability Statement

The artifact [52] associated with this paper is publicly available under DOI: https://doi.org/10.5281/zenodo.17725527.

Acknowledgments

We appreciate the valuable comments of all reviewers and the shepherd's guidance. Weifeng Liu is the corresponding author of this paper. This work has been partially supported by the National Natural Science Foundation of China (U23A20301 and 62372467). This work has also received funding from 'Future of Computing, a Barcelona Supercomputing Center and IBM initiative' (2023) and has been partially supported by the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU /AEI/10.13039/501100011033 and EU ERDF. We are also grateful to Yuyao Niu for help in the poster design.

References

[1] 2024. The top500 list. https://top500.org/

[2] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, and Samuel Webb Williams. 2006. The landscape of parallel computing research: A view from berkeley. (2006). https://escholarship.org/uc/item/1z50m2xt

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (2009). doi:10.1145/1562764.1562783

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks—summary and preliminary results. In *SC '91*. doi:10.1145/125826.125925

[5] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09*. doi:10.1145/1654059.1654078

[6] Jay P Boris. 1970. Relativistic plasma simulation-optimization of a hybrid code. In *CNSP '70*.

[7] John Burgess. 2019. Rtx on—the nvidia turing gpu. In *HCS '19*. doi:10.1109/HOTCHIPS.2019.8875651

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*. doi:10.1109/IISWC.2009.5306797

[9] Jou-An Chen, Hsin-Hsuan Sung, Ruifeng Zhang, Ang Li, and Xipeng Shen. 2025. Accelerating GNNs on GPU Sparse Tensor Cores through N: M Sparsity-Oriented Graph Reordering. In *PPoPP '25*. doi:10.1145/3710848.3710881

[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS '14*. doi:10.1145/2541940.2541967

[11] Yuetao Chen, Kun Li, Yuhao Wang, Donglin Bai, Lei Wang, Lingxiao Ma, Liang Yuan, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. ConvStencil: Transform stencil computation to matrix multiplication on tensor cores. In *PPoPP '24*. doi:10.1145/3627535.3638476

[12] Jack Choquette. 2017. Nvidia's volta gpu: Programmability and performance for gpu computing. In *HCS '17*.

[13] Jack Choquette. 2022. Nvidia hopper gpu: Scaling performance. In *HCS '22*. doi:10.1109/HCS55958.2022.9895592

[14] Jack Choquette and Wish Gandhi. 2020. Nvidia a100 gpu: Performance & innovation for gpu computing. In *HCS '20*. doi:10.1109/HCS49909.2020.9220622

[15] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021). doi:10.1109/MM.2021.3061394

[16] John Cocke and Victoria Markstein. 1990. The evolution of RISC technology at IBM. *IBM Journal of research and development* 34, 1 (1990). doi:10.1147/rd.341.0004

[17] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *ICS '19*. doi:10.1145/3330345.3331057

[18] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU '10*. doi:10.1145/1735688.1735702

[19] Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011). doi:10.1145/2049662.2049663

[20] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. 2017. Bringing parallel patterns out of the corner: The p^{3}arsec benchmark suite. *ACM Transactions on Architecture and Code Optimization* 14, 4 (2017). doi:10.1145/3132710

[21] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng ChenT, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed WahibT, and Satoshi Matsuoka. 2021. Matrix engines for high performance computing: A paragon of performance or grasping at straws?. In *IPDPS '21*. doi:10.1109/IPDPS49936.2021.00114

[22] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003). doi:10.1002/cpe.728

[23] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. 2021. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *PACT '21*. doi:10.1109/PACT52795.2021.00032

[24] 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 *SC '21*. doi:10.1145/3458817.3476157

[25] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *ASPLOS '23*. doi:10.1145/3575693.3576933

[26] Xixhou Feng, Rong Ge, and Kirk W Cameron. 2005. Power and energy profiling of scientific applications on distributed systems. In *IPDPS '05*. doi:10.1109/IPDPS.2005.346

[27] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. 2010. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems* 21, 5 (2010). doi:10.1109/TPDS.2009.76

[28] Haozhi Han, Kun Li, Wei Cui, Donglin Bai, Yiwei Zhang, Liang Yuan, Yifeng Chen, Yunquan Zhang, Ting Cao, and Mao Yang. 2025. FlashFFTStencil: Bridging fast fourier transforms to memory-efficient stencil computations on tensor core units. In *PPoPP '25*. doi:10.1145/3710848.3710897

[29] Nhut-Minh Ho and Weng-Fai Wong. 2022. Tensorox: Accelerating GPU applications via neural approximation on unused tensor cores. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022). doi:10.1109/TPDS.2021.3093239

[30] Guyue Huang, Haoran Li, Minghai Qin, Fei Sun, Yufei Ding, and Yuan Xie. 2022. Shfl-BW: Accelerating deep neural network inference with tensor-core aware weight pruning. In *DAC '22*. doi:10.1145/3489517.3530588

[31] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2016. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Trans. Comput.* 66, 1 (2016). doi:10.1109/TC.2016.2582151

[32] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2023. Vegeta: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus. In *HPCA '23*. doi:10.1109/HPCA56546.2023.10071058

[33] Zhuoran Ji and Cho-Li Wang. 2022. Efficient exact k-nearest neighbor graph construction for billion-scale datasets using GPUs with tensor cores. In *ICS '22*. doi:10.1145/3524059.3532368

[34] Hong Jiang. 2022. Intel's Ponte Vecchio GPU : Architecture, Systems & Software. In *HCS '22*. doi:10.1109/HCS55958.2022.9895631

[35] Peng Jiang, Lihan Hu, and Shihui Song. 2022. Exposing and exploiting fine-grained block structures for fast and accurate sparse training. In *NeurIPS '22*. https://proceedings.neurips.cc/paper_files/paper/2022/hash/fa69e968b7319fd42524febd41475fb3-Abstract-Conference.html

[36] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA '17*. doi:10.1145/3079856.3080246

[37] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS. In *HPEC '16*. doi:10.1109/HPEC.2016.7761646

[38] D.H. Lehmer. 1951. Mathematical methods in large-scale computing units. *The Annals of' the Computation Laboratory of Harvard University* 26 (1951).

[39] Yuhan Leng, Gaoyuan Zou, Hansheng Wang, Panruo Wu, and Shaoshuai Zhang. 2025. High Performance Householder QR Factorization on Emerging GPU Architectures Using Tensor Cores. *IEEE Transactions on Parallel and Distributed Systems* 36, 3 (2025). doi:10.1109/TPDS.2024.3522776

[40] Ang Li and Simon Su. 2021. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021). doi:10.1109/TPDS.2020.3045828

[41] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcfft: A fast half-precision fft library for nvidia tensor cores. In *CLUSTER '21*. doi:10.1109/Cluster48925.2021.00035

[42] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *CGO '21*. doi:10.1109/CGO51591.2021.9370335

[43] Qi Li, Kun Li, Haozhi Han, Liang Yuan, Junshi Chen, Yunquan Zhang, Yifeng Chen, Hong An, Ting Cao, and Mao Yang. 2025. SparStencil: Retargeting Sparse Tensor Cores to Scientific Stencil Computations via Structured Sparsity Transformation. In *SC '25*. doi:10.1145/3712285.3759820

[44] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient quantized sparse matrix operations on tensor cores. In *SC '22*. doi:10.1109/SC41404.2022.00042

[45] Haocheng Lian, Qiyue Zhang, Xinran Zhao, Meichen Dong, Yijie Nie, Zhengyi Zhao, Junzhong Shen, Wei Guo, Chun Huang, Bingcai Sui, and Weifeng Liu. 2026. Uni-STC: Unified Sparse Tensor Core. In *HPCA '26*.

[46] Jun Liu, Guohao Dai, Hao Xia, Lidong Guo, Xiangsheng Shi, Jiaming Xu, Huazhong Yang, and Yu Wang. 2023. Tstc: Two-level sparsity tensor core enabling both algorithm flexibility and hardware efficiency. In *ICCAD '23*. doi:10.1109/ICCAD57390.2023.10323775

[47] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS '15*. doi:10.1145/2751205.2751209

[48] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *ICS '22*. doi:10.1145/3524059.3532392

[49] Yiwei Liu and Olin Johnson. 1988. Optimal scheduling policies for mixed scalar-vector multiprocessor supercomputers. In *SC '88*. doi:10.1109/SUPERC.1988.44661

[50] André Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. 2017. Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling. In *ISPASS '17*. doi:10.1109/ISPASS.2017.7975297

[51] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *SC '23*. doi:10.1145/3581784.3607051

[52] Yuechen Lu, Hongwei Zeng, Marc Casas, and Weifeng Liu. 2025. Cubie. doi:10.5281/zenodo.17725527

[53] 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 *SC '24*. doi:10.1109/SC41406.2024.00058

[54] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *IPDPSW '18*. doi:DOI:10.1109/IPDPSW.2018.00091

[55] Diogo Marques, Aleksandar Ilic, Zakhar A Matveev, and Leonel Sousa. 2020. Application-driven cache-aware roofline model. *Future Generation Computer Systems* 107 (2020). doi:10.1016/j.future.2020.01.044

[56] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. 2017. GraphBLAS C API: Ideas for future versions of the specification. In *HPEC '17*. doi:10.1109/HPEC.2017.8091095

[57] Vishal Mehta. 2019. Particle in Cell using Tensor Core. https://github.com/vishalmehta1991/pictc.

[58] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *ISSCC '22*. doi:10.1109/ISSCC42614.2022.9731107

[59] Yuyao Niu and Marc Casas. 2025. BerryBees: Breadth first search by bit-tensor-cores. In *PPoPP '25*. doi:10.1145/3710848.3710859

[60] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[61] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

[62] NVIDIA. 2024. NVIDIA Blackwell Architecture Technical Brief. https://resources.nvidia.com/en-us-blackwell-architecture

[63] NVIDIA. 2024. NVIDIA GH200 Grace Hopper Superchip Architecture. https://nvdam.widen.net/s/c9lts6msjj/nvidia-grace-hopper-superchip-architecture-whitepaper

[64] NVIDIA. 2025. The API reference for CUB. https://docs.nvidia.com/cuda/cub/

[65] NVIDIA. 2025. The API Reference guide for cuBLAS, the CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/

[66] NVIDIA. 2025. The API reference guide for cuFFT, the CUDA Fast Fourier Transform library. https://docs.nvidia.com/cuda/cufft/

[67] NVIDIA. 2025. The API reference guide for cuSPARSE, the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/

[68] NVIDIA. 2025. CUDA Samples. https://docs.nvidia.com/cuda/cuda-samples/

[69] NVIDIA. 2025. CUDA Templates for Linear Algebra Subroutines and Solvers. https://nvidia.github.io/cutlass/

[70] NVIDIA. 2025. Nsight Compute. https://docs.nvidia.com/nsight-compute/

[71] NVIDIA. 2025. NVIDIA Management Library. https://developer.nvidia.com/management-library-nvml

[72] NVIDIA. 2025. NVIDIA Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/

[73] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High Performance Unstructured SpMM Computation Using Tensor Cores. In *SC '24*. doi:10.1109/SC41406.2024.00060

[74] 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). doi:10.1177/10943420241239588

[75] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011). https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

[76] Louis Pisha and Łukasz Ligowski. 2021. Accelerating non-power-of-2 size Fourier transforms with GPU tensor cores. In *IPDPS '21*. doi:10.1109/IPDPS49936.2021.00059

[77] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *HPCA '20*. doi:10.1109/HPCA47549.2020.00015

[78] Gabin Schieffer, Daniel Araújo De Medeiros, Jennifer Faj, Aniruddha Marathe, and Ivy Peng. 2024. On the rise of amd matrix cores: Performance, power efficiency, and programmability. In *ISPASS '24*. doi:10.1109/ISPASS61541.2024.00022

[79] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. 2024. Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper. In *ICPP '24*. doi:10.1145/3673038.3673110

[80] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. 2025. Flashsparse: Minimizing computation redundancy for fast sparse matrix multiplications on tensor cores. In *PPoPP '25*. doi:10.1145/3710848.3710858

[81] Andrew Siegel, Erik W Draeger, Jack Deslippe, Thomas Evans, Marianne M Francois, Timothy C Germann, Daniel F Martin, and William Hart. 2021. *Map applications to target exascale architecture with machine-specific performance analysis, including challenges and projections.* Technical Report. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). doi:10.2172/1838979

[82] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. 2022. Not all gpus are created equal: characterizing variability in large-scale, accelerator-rich systems. In *SC '22*. doi:10.1109/SC41404.2022.00070

[83] Alan Smith and Norman James. 2022. AMD Instinct™ MI200 Series Accelerator and Node Architectures. In *HCS '22*. doi:10.1109/HCS55958.2022.9895477

[84] Zhuoran Song, Jianfei Wang, Tianjian Li, Li Jiang, Jing Ke, Xiaoyao Liang, and Naifeng Jing. 2020. GPNPU: Enabling Efficient Hardware-Based Direct Convolution with Multi-Precision Support in GPU Tensor Cores. In *DAC '20*. doi:10.1109/DAC18072.2020.9218566

[85] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127, 7.2 (2012). http://impact.crhc.illinois.edu/Shared/Report/impact-12-01.parboil.pdf

[86] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023). doi:10.1109/TPDS.2022.3217824

[87] Ajay Tirumala and Raymond Wong. 2024. Nvidia blackwell platform: Advancing generative ai and accelerated computing. In *HCS '24*. doi:10.1109/HCS61935.2024.10665247

[88] Hansheng Wang, Zhekai Duan, Zitian Zhao, Siqi Wu, Saiqi Zheng, Qiao Li, Xu Jiang, and Shaoshuai Zhang. 2025. Improving Tridiagonalization Performance on GPU Architectures. In *PPoPP '25*. doi:10.1145/3710848.3710894

[89] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP '16*. doi:10.1145/2851141.2851145

[90] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *PPoPP '22*. doi:10.1145/3503221.3508408

[91] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *USENIX ATC '23*. https://www.usenix.org/conference/atc23/presentation/wang-yuke

[92] Yueyao Wang, Samuel Furman, Nicolas Hardy, Margaret Ellis, Godmar Back, Yili Hong, and Kirk Cameron. 2024. A detailed historical and statistical analysis of the influence of hardware artifacts on SPEC integer benchmark performance. *IEEE Trans. Comput.* 73, 5 (2024). doi:10.1109/TC.2024.3365941

[93] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *ISCA '21*. doi:10.1109/ISCA52012.2021.00088

[94] Martin Weidmann. 2021. Introducing the Scalable Matrix Extension for the Armv9-A Architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/scalable-matrix-extension-armv9-a-architecture

[95] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying tensorized instruction compilation. In *CGO '21*. doi:10.1109/CGO51591.2021.9370330

[96] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet Another SpMV Framework on GPUs. In *PPoPP '14*. doi:10.1145/2555243.2555255

[97] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *ASPLOS '23*. doi:10.1145/3582016.3582047

[98] Xin You, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2021. DRStencil: Exploiting data reuse within low-order stencil on GPU. In *HPCC '21*. doi:10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00036

[99] 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 *ICPP '24*. doi:10.1145/3673038.3673108

[100] Ruge Zhang, Haipeng Jia, Yunquan Zhang, Baicheng Yan, Penghao Ma, Long Wang, and Wenxuan Zhao. 2024. OpenFFT-SME: An Efficient Outer Product Pattern FFT Library on ARM SME CPUs. In *IPDPS '24*. doi:10.1109/IPDPS57955.2024.00088

[101] Shaoshuai Zhang, Ruchi Shah, Hiroyuki Ootomo, Rio Yokota, and Panruo Wu. 2023. Fast symmetric eigenvalue decomposition via wy representation on tensor core. In *PPoPP '23*. doi:10.1145/3572848.3577516

[102] Yiwei Zhang, Kun Li, Liang Yuan, Jiawen Cheng, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. LoRAStencil: Low-Rank Adaptation of Stencil Computation on Tensor Cores. In *SC '24*. doi:10.1109/SC41406.2024.00059

A Artifact Evaluation

This artifact accompanies the submission #336 entitled 'Characterizing Matrix Multiplication Units across General Parallel Patterns in Scientific Computing'. It provides a benchmark suite for Matrix Multiplication Units (MMUs), covering ten workloads evaluated in terms of performance, power consumption, and numerical accuracy.

The artifact consists of two main components: the Getting Started Guide and the Step-by-Step Instructions: The first part introduces the prerequisites for running the artifact and provides a quick test that evaluates four representative workloads from the paper. It includes the necessary compilation and execution commands, along with the expected outputs for performance, power, and accuracy. This quick test is designed to complete in approximately 30 minutes. The second part presents the full evaluation workflow. It offers detailed, step-by-step instructions for testing all ten workloads in the suite, including the commands required for measuring performance, power consumption, and numerical accuracy, as well as the corresponding expected outputs.

A.1 Getting Started Guide

A.1.1 Prerequisites.

- **Hardware Requirements:**
  - GPU: At least one GPU with support for FP64 tensor cores (*e.g.*, NVIDIA Ampere, Hopper, or Blackwell GPU). In this paper, we use NVIDIA A100, H200 and B200 GPUs.
  - CPU: Any multicore CPU (Intel Xeon Silver 4210 as tested).
  - Disk Space: At least 8GB.
- **Software Requirements:**
  - For evaluating the artifact: GCC v9.4.0 or higher, NVIDIA CUDA Toolkit v12.0 or above, and CMake v3.30.4 or above.
  - For reproducing the figures: Python v3.9 or above, matplotlib v3.10.7 or above, numpy v2.2.6 or above, packaging v25.0 or above, and pandas v2.3.3 or above.
- **Datasets/Inputs:** Each workload is evaluated using five test cases. Sparse matrices and graphs used in SpMV, SpGEMM and BFS are sourced from the SuiteSparse Matrix Collection. Other test cases use custom inputs of different sizes. All input datasets have been fully prepared in the artifact.

A.1.2 Setup and Quick Test.

- **Download the artifact:** Two options are available for obtaining this artifact.
  (1) Download the prebuilt image, which contains the full required environment and the artifact already placed at `/workspace/`. Pull the prebuilt Docker image:

```
$ docker pull yuechen0210/cubie:v1
```

  Start a Docker container using the pulled image:

```
$ docker run -it --rm --runtime-nvidia \
--gpus all yuechen0210/cubie:v1
```

  (2) Download the artifact package directly to your local machine. The artifact can be downloaded from the following link: https://doi.org/10.5281/zenodo.17725527. Extract the package:

```
$ unzip Cubie-ppopp26.zip
```

- **Inspect and configure the environment settings.** Please check the configuration file at `Cubie/config.mk` and modify it if necessary. By default, the CUDA path is set to `/usr/local/cuda`, the target GPU architecture is Ampere, and the device ID is 0. If your local environment differs from these defaults, update the corresponding entries in `config.mk` accordingly.
- **Run the quick test:** Navigate to the `quick_test` directory and execute the provided script according to your GPU architecture (Ampere, Hopper, or Blackwell):

```
$ cd quick_test
$ nohup sh runme.sh Ampere
```

  This quick test is a lightweight evaluation designed to complete in approximately 30 minutes. It runs four representative workloads (SpMV, Reduction, Scan, and FFT) and produces their performance results, numerical accuracy reports, and power consumption measurements.
  Note: Please run the script with `nohup` so the job keeps running even if the terminal is closed or the SSH connection drops (otherwise it may be terminated by a hangup signal). The output will be written to `nohup.out` for later inspection.
- **Expected Outputs:** After the quick test finishes, the `quick_test` directory should contain the performance, power, and accuracy results for the four evaluated workloads.
  - The performance plots should include `Figure3_perf.pdf`, `Figure4_TCvsBaseline.pdf`, `Figure5_CCvsTC.pdf`, and `Figure6_CCEvsTC.pdf`. These figures record the absolute performance values and speedup comparisons across different kernel variants.
  - The power results should include `Figure7_edp.pdf` and `Figure8_power.pdf`. These figures summarize the energy-delay product (EDP) and the power consumption traces over time.

– The accuracy report is in the file `all_error.csv`. This file records the average and maximum numerical error measured for the four workloads.

Note: This artifact is a benchmarking suite, so the exact numbers and curves may vary across different GPUs, and do not need to match the results in the paper exactly.

A.2 Step-by-Step Instructions

A.2.1 Overview of Evaluation Goals. The full evaluation of this artifact is designed to run all ten workloads in our Cubie benchmarking suite, each implemented using three or four kernel variants (Baseline, TC, CC, and CC-E). The evaluation validates the performance (Figures 3–6 in the paper), power consumption (Figures 7–8 in the paper), and numerical accuracy results (Table 6 in the paper) presented in our paper.

A.3 Detailed Steps for Full Evaluation

- **Run all tests:** To execute the full evaluation, navigate to the `Cubie/` directory and run the provided script according to your GPU architecture (Ampere, Hopper, or Blackwell):

```
$ nohup sh runme.sh Ampere
```

  The complete evaluation is expected to finish in approximately **five hours**.
  Note: Please run the script with `nohup` so the job keeps running even if the terminal is closed or the SSH connection drops (otherwise it may be terminated by a hangup signal). The output will be written to `nohup.out` for later inspection.
- **The evaluation procedure:** The execution script `runme.sh` performs the following steps in sequence:
  – Compilation test: runs `compile_test.sh` to verify that all workloads and kernel variants can be successfully compiled.
  – Performance evaluation: runs `run_perf.sh` to measure the absolute performance of the ten workloads and compute speedups across the Baseline, TC, CC, and CC-E variants.
  – Power evaluation: runs `run_power.sh` to collect power traces for all workloads and compute their energy-delay product (EDP).
  – Accuracy evaluation: runs `run_error.sh` to compute the average and maximum numerical error for each workload and kernel variant.
- **Expected outputs of the full evaluation:** Upon completion, all evaluation results will be generated under the `Cubie/script/` directory. These outputs correspond to Figures 3–8 and Table 6 in the paper and summarize the performance, power, and numerical accuracy of all ten workloads across the Baseline, TC, CC, and CC-E implementations.
  – **Performance** Evaluation (Figures 3–6): The performance plots should include:
  (1) `Figure3_perf.pdf`: absolute performance of all workloads and variants.
  The plot will report throughput (*e.g.*, TFLOP/s) for each workload and kernel variant, showing all five test cases separately.
  (2) `Figure4_TCvsBaseline.pdf`: speedups of the Tensor Core (TC) version over the vector-based baseline version.
  The figure will report the average speedup across the five test cases for each workload.
  (3) `Figure5_CCvsTC.pdf`: speedups of the CUDA Core (CC) version over the Tensor Core (TC) version.
  The figure will report the average speedup across the five test cases for each workload.
  (4) `Figure6_CCEvsTC.pdf`: speedups of the CUDA Core Essential (CC-E) version over the Tensor Core (TC) version.
  The figure will report the average speedup across the five test cases for each workload.
  – **Power and Energy** Evaluation (Figures 7–8): The power-related results should include:
  (1) `Figure7_edp.pdf`: energy-delay product (EDP) of all workloads and variants.
  It will evaluate one representative test case per workload, and report the corresponding EDP for each variant. The plot groups workloads by quadrant and additionally reports the geometric-mean EDP for each quadrant at the end.
  Clarify the definition used (*e.g.*, $EDP = Power \times Time^2$) and the time window is kernel-only.
  (2) `Figure8_power.pdf`: power curves over time.
  The figure will show instantaneous power (W) versus time for representative workloads/variants, aligned to kernel start/end. Similar to Figure 7, we evaluate one representative test case per workload. To capture stable power values, each kernel is executed repeatedly in a loop during measurement.
  – **Numerical Accuracy** Evaluation (Table 6): The accuracy output should include
  (1) `all_error.csv`: average and maximum numerical error for each workload and kernel variant.
  It will evaluate one representative test case per workload. The table reports `workload name`, `variant name`, `Average_Error`, and `Max_Error` for each entry. Empirically, the TC and CC variants exhibit identical results for all workloads; thus, they are grouped and reported together in the table.

Received 2025-09-01; accepted 2025-11-10