Can Tensor Cores Accelerate Non-GEMM Workloads? An Analytical Study

Lingqi Zhang[2,a)] Jiajun Huang[3,b)] Sheng Di[4,c)] Satoshi Matsuoka[2,d)] Mohamed Wahib[2,e)]

Abstract:
Tensor Cores are specialized units integrated in modern GPUs, designed to accelerate dense matrix operations with remarkable efficiency. They have proven particularly effective in compute-bound workloads, such as those found in deep learning training, where general matrix-matrix multiplication (GEMM) is prevalent. Motivated by this success, recent efforts have explored extending Tensor Core usage to non-GEMM computational patterns. However, despite their potential, effectively utilizing Tensor Cores in broader contexts requires a thorough understanding of their performance characteristics across diverse workloads. This work investigates the applicability of Tensor Cores to non-GEMM workloads, seeking to answer a fundamental question: Can Tensor Cores accelerate non-GEMM kernels?

1. Introduction

Since Nvidia introduced tensor cores in their Volta architecture in 2017 [1], these specialized computational units have revolutionized high-performance computing by enabling mixed-precision or low repcision matrix operations with unprecedented throughput. Tensor cores are designed to accelerate matrix multiplication operations by performing computations in lower precision formats (such as half-precision or TF32), thereby achieving significant performance improvements over traditional floating-point units.

The introduction of tensor cores has catalyzed extensive research across diverse computational domains. In dense linear algebra, researchers have developed automatic kernel generation techniques [2] to optimize matrix operations for tensor core architectures. The sparse linear algebra community has similarly embraced these units, developing specialized algorithms for sparse matrix computations [3, 4] that leverage tensor cores' unique capabilities. Additionally, spectral methods have been adapted to exploit tensor core performance [5, 6], demonstrating the broad applicability of these computational units. The low-precision computation capabilities of tensor cores, which offer substantially higher performance compared to traditional double-precision operations, have spawned innovative algorithmic approaches. Mixed-precision algorithms [7, 8] strategically combine different precision levels to maintain numerical stability while maximizing computational throughput. Complementing these approaches, precision recovery techniques [9–11] have emerged to restore full precision accuracy when required, creating a comprehensive ecosystem of tensor core-optimized numerical methods. However, despite this broad adoption across multiple research areas, fundamental analysis of tensor core performance characteristics remains surprisingly limited. Only a handful of studies have conducted systematic microbenchmarking [12–14] to understand the intricate performance behavior of these units. This gap in understanding presents a significant challenge for developers and researchers seeking to maximize the potential of tensor core architectures.

While tensor cores represent a powerful tool for computational acceleration, their effective utilization requires a thorough understanding of their performance characteristics, computational limitations, and optimal usage patterns. This research addresses this knowledge gap by focusing specifically on memory-bound kernels, which constitute a significant portion of HPC workloads [15]. Our study seeks to answer two fundamental questions that are essential for understanding the practical utility of tensor cores:

- What is the theoretical performance ceiling for tensor cores when applied to memory-bound kernels, compared to the conventional CUDA implementation?
- Do current tensor core implementation strategies provide measurable performance benefits for memory-bound kernels, or are the advantages primarily theoretical?

To address these questions, we make the following contributions:

- A comprehensive theoretical analysis of tensor core performance for memory-bound kernels.

[1] Tokyo Institute of Technology, Japan
[2] RIKEN Center for Computational Science, Japan
[3] University of South Florida, USA
[4] Argonne National Laboratory, USA
[a)] lingqi.zhang@riken.jp
[b)] jiajunhuang@usf.edu
[c)] sdi1@anl.gov
[d)] matsu@acm.org
[e)] mohamed.attia@riken.jp

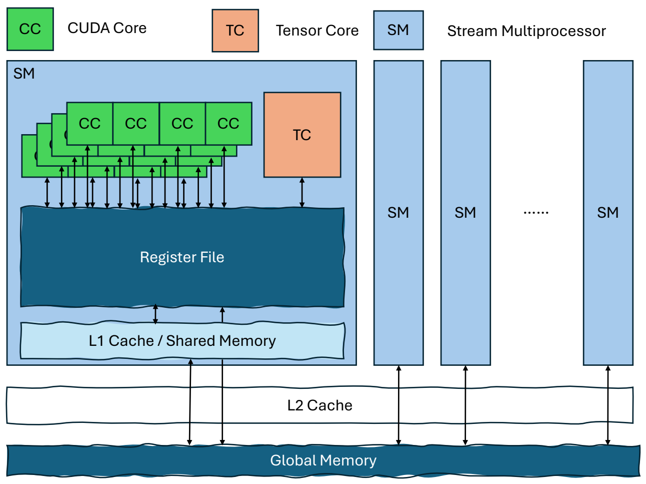


Fig. 1: Nvidia GPU memory hierarchy.

- An empirical evaluation analysis comparing tensor core implementations against their CUDA core counterparts across representative memory-bound kernels.

The rest of the paper is organized as follows. Section 2 provides essential background concepts. Section 3 introduces our studied memory-bound workloads. Section 4 analyzes two extreme scenarios to establish tensor core performance bounds. Section 5 empirically substantiates our claims using representative memory-bound kernels. Finally, we present key takeaways and conclusions.

2. Background

2.1 Tensor Core

Tensor Core is a specialized systolic array matrix engine [16] integrated within the Stream Multiprocessor (SM) of modern Nvidia GPUs. It operates alongside traditional CUDA cores. The execution pathway for both units follows the GPU's memory hierarchy: data is first loaded from global memory into the register file, from which either CUDA cores or Tensor Cores can access it for computation. This memory access abstraction aligns with previous studies [12, 13]. fig. 1 illustrates this memory hierarchy and the relationship between different memory levels in Nvidia GPUs.

2.2 Machine Balance

We define machine balance ($\mathbb{B}$) [17] as the ratio between peak computational performance (P) and memory bandwidth (B):

$$\mathbb{B} = \frac{P}{B} \qquad (1)$$

2.3 Roofline Model

The roofline model [18, 19] provides an upper-bound performance prediction framework based on operational intensity ($\mathbb{I}$), which is defined as the ratio of computational work ($\mathbb{W}$) to memory traffic ($\mathbb{Q}$):

$$\mathbb{I} = \frac{\mathbb{W}}{\mathbb{Q}} \qquad (2)$$

The model calculates attainable performance ($\mathbb{P}$) as:

$$\mathbb{P} = \min(P, B \times \mathbb{I}) \qquad (3)$$

This visualization framework helps identify system bottlenecks and performance limits.

2.4 Tensor Core in the Roofline Model

As discussed in Section 2.1, tensor cores can be represented as an additional performance ceiling above the CUDA core baseline in the roofline model, because tensor cores and CUDA cores share the same memory hierarchy and cannot operate simultaneously due to the Dark Silicon Effect. This roofline abstraction aligns with an existing study [20].

2.5 Relationship Between Machine Balance and Roofline Model

The roofline model provides a framework for understanding performance bottlenecks through the interplay of two critical metrics: operational intensity and machine balance. While operational intensity ($\mathbb{I}$) characterizes a kernel's computational density, machine balance ($\mathbb{B}$) represents the hardware's ratio of computational capability to memory bandwidth. The relationship between these metrics determines whether a kernel's performance is mainly limited by computation or memory access:

$$A\ kernel\ is = \begin{cases} compute-bound, & \text{if } \mathbb{I} > \mathbb{B} \\ memory-bound, & \text{if } \mathbb{I} < \mathbb{B} \end{cases} \qquad (4)$$

As illustrated in Figure 2, machine balance manifests as the inflection point in the roofline curve for both GH200 and A100-80GB GPUs. This point marks the transition from memory-bound to compute-bound behavior.

3. Workloads: Memory-Bound Kernels

This section examines three representative memory-bound kernels: Sparse Matrix-Vector Multiplication (SpMV) (Section 3.2), and Stencil (Section 3.3). We analyze these kernels through the lens of operational intensity ($\mathbb{I}$), focusing on double-precision operations (data size $\mathbb{D}$ = 8 bytes).

While our analysis centers on double precision, the methodology can be extended to lower-precision scenarios.

3.1 SCALE

SCALE, one of the STREAM benchmark [21], is defined as

$$a_i = qb_i, \quad \forall i \in 1, \ldots, n, \quad a, b \in \mathbb{R}^n, \quad q \in \mathbb{R} \qquad (5)$$

Each element operation requires one load, one store, and one computation, yielding: $\mathbb{W}(\text{SCALE}) = 1$, $\mathbb{Q}(\text{SCALE}) = 2 \times \mathbb{D}$, and consequently $\mathbb{I}(\text{SCALE}) = \frac{1}{16}$. STREAM benchmark is commonly used to measure sustainable memory bandwidth due to its low computational intensity.

3.2 Sparse Matrix–Vector Multiplication (SpMV)

SpMV, crucial for iterative solvers, has format-dependent operational intensity. In this section, we begin by analyzing dense matrix-vector multiplication (GEMV) as a baseline.

GEMV: For matrix $A \in \mathbb{R}^{m \times n}$ and vectors $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, GEMV is defined as:

$$y = Ax \qquad (6)$$

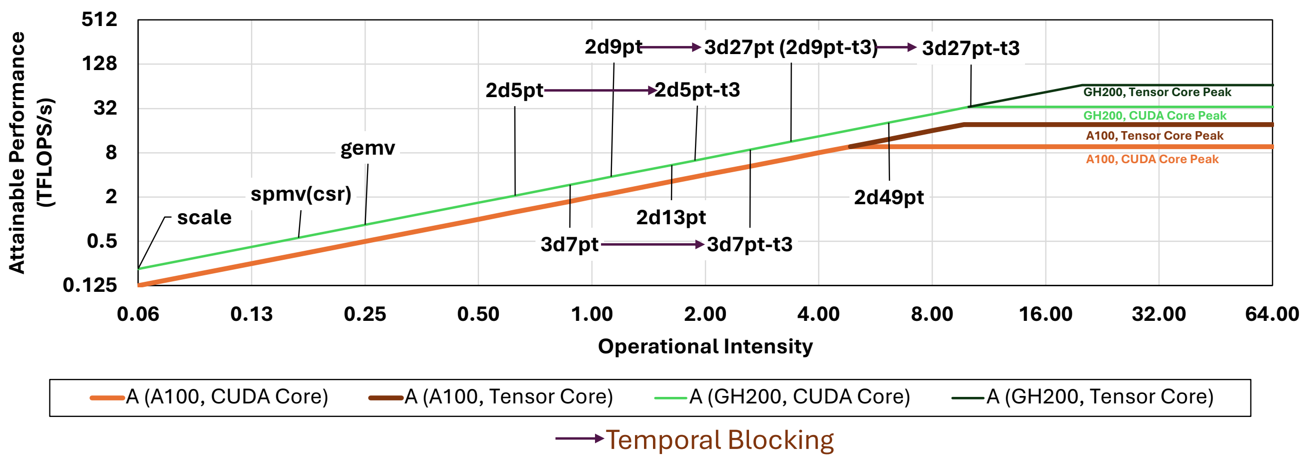


Fig. 2: An example of the roofline model for both GH200 and A100-80GB GPU.

With computing $\mathbb{W}(\text{GEMV}) = m \times n \times 2$ operations and memory traffic $\mathbb{Q}(\text{GEMV}) = (m \times n + m + n) \times \mathbb{D}$ yields:

$$\mathbb{I}(\text{GEMV}) = \frac{m \times n \times 2}{(m \times n + m + n) \times \mathbb{D}} \approx \frac{2}{\mathbb{D}} = \frac{1}{4} \tag{7}$$

SpMV: For sparse matrices with nnz non-zeros, SpMV is defined as:

$$y = Ax,\ A \in \mathbb{R}^{m \times n}, \text{nnz}(A) \quad \ll mn, x \in \mathbb{R}^{n}, y \in \mathbb{R}^{m} \tag{8}$$

With computation $\mathbb{W}(\text{SpMV}) = 2 \times nnz$ and memory traffic including coordinate information $\alpha\mathbb{I}$ or packed values $\beta\mathbb{Z}$:

$$\mathbb{I}(\text{SpMV}) = \frac{nnz \times 2}{(nnz + m + n) \times \mathbb{D} + \alpha\mathbb{I} + \beta\mathbb{Z}} \tag{9}$$

Given $nnz \ll m \times n$, we have $\mathbb{I}(\text{SpMV}) < \mathbb{I}(\text{GEMV})$.

Compressed Sparse Row (CSR) format: CSR format, the most common sparse representation, requires storing column indices and row pointers. With memory traffic $\mathbb{Q}(\text{SpMV,CSR}) = (nnz + m + n) \times \mathbb{D} + (nnz + m + 1) \times \mathbb{I}$ and computation $\mathbb{W}(\text{SpMV,CSR}) = 2 \times nnz$:

$$\begin{aligned} \mathbb{I}(\text{SpMV,CSR}) &= \frac{2 \times nnz}{(nnz + m + n) \times \mathbb{D} + (nnz + m + 1) \times \mathbb{I}} \\ &\approx \frac{2}{\mathbb{D} + \mathbb{I}} = \frac{1}{6} < \mathbb{I}(\text{GEMV}) \end{aligned} \tag{10}$$

This analysis confirms SpMV's memory-bound nature, consistent with prior works [22, 23].

3.3 Iterative Stencils

Stencil computations is common in HPC [24]. For 2D stencil, we have:

$$v(i, j) = \sum_{(p,q)\in\mathbb{S}} w_{p,q} \cdot u(i + p, j + q) \tag{11}$$

where $v(i, j)$ and $u(i, j)$ are updated and original values at point (i, j), and $\mathbb{S}$ defines relative offsets (e.g., 5-point stencil: $(-1, 0)$, $(1, 0)$, $(0, 1)$, $(0, -1)$, $(0, 0)$). Ideally, only one load of u and one store of v are necessary:

$$\mathbb{Q} = 2 \times \mathbb{D}, \quad \mathbb{W} = 2 \times |\mathbb{S}|, \quad \mathbb{I} = \frac{|\mathbb{S}|}{\mathbb{D}} \tag{12}$$

For a 2d5pt stencil where $|\mathbb{S}(\text{2d5pt})| = 5$, $\mathbb{I}(\text{2d5pt}) = \frac{5}{8}$.

Temporal blocking [25, 26] combines t timesteps together:

$$\mathbb{W}_t = t \times 2 \times |\mathbb{S}|, \quad \mathbb{I}_t = t \times \frac{|\mathbb{S}|}{\mathbb{D}} \tag{13}$$

While temporal blocking can theoretically transform memory-bound stencils into compute-bound kernels by increasing operational intensity, practical limitations exist.

For a 2d5pt stencil on GH200 ($\mathbb{B}_{GH200} = 9.99$), compute-bound behavior requires:

$$t \times \mathbb{I}(\text{2d5pt}) > \mathbb{B}_{GH200} \implies t \times 0.625 > 9.99 \implies t > 15.98 \tag{14}$$

However, deep temporal blocking (e.g., $t > 16$) usually faces hardware limits from register pressure [25, 26].

Thus, shallow temporal blocking ($t < 16$) 2d5pt stencil remains memory-bound, while deep temporal blocking might make stencil kernel register-bound.

4. Theoretical Analysis

To simplify the following discussion, we assume that all kernels are throughput-bound, which is usually the case in High-Performance workloads. We have time for computation $T_{\text{cmp}} = \frac{\mathbb{W}}{P}$. and time for memory access $T_{mem} = \frac{\mathbb{Q}}{B}$. So we have:

$$\frac{T_{mem}}{T_{cmp}} = \frac{\frac{\mathbb{Q}}{B}}{\frac{\mathbb{W}}{P}} = \frac{\mathbb{B}}{\mathbb{I}} \tag{15}$$

For memory-bound kernel, $\mathbb{B} > \mathbb{I}$, we have:

$$T_{\text{mem}} > T_{\text{cmp}} \tag{16}$$

We analyze two extreme cases: fully overlapped and fully unoverlapped for memory access and computation.

4.1 Fully Overlapped

An example of an overlapped kernel time breakdown is shown in Figure 3. For memory-bound kernels:

$$T = \max(T_{\text{cmp}}, T_{\text{mem}}, T_{\text{others}}) = \max(T_{\text{mem}}, T_{\text{others}}) \tag{17}$$

where T_{mem}, T_{cmp}, T_{others} and T represent memory access, computation, other operation and total times, respectively. In this scenario, reducing computation time cannot influence total runtime.

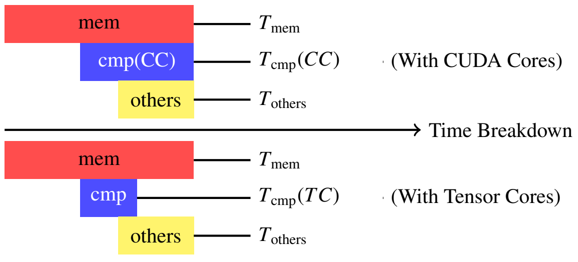


Fig. 3: Fully overlapped kernel time breakdown

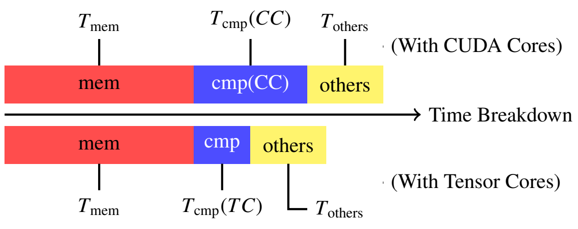


Fig. 4: Fully un-overlapped kernel time breakdown

4.2 Fully Un-overlapped

For fully un-overlapped kernels (fig. 4):

$$T = T_{\text{cmp}} + T_{\text{mem}} + T_{\text{others}} \tag{18}$$

With tensor cores providing speedup α ($\alpha = \frac{P(TC)}{P(CC)}, \alpha > 1$): $T'_{\text{cmp}}(TC) = \frac{1}{\alpha} T_{\text{cmp}}(CC)$, yielding:

$$Speedup = \frac{T(CC)}{T(TC)} = \frac{T_{\text{cmp}(CC)} + T_{\text{mem}} + T_{\text{others}}}{\frac{1}{\alpha} T_{\text{cmp}(CC)} + T_{\text{mem}} + T_{\text{others}}} \tag{19}$$

$$= 1 + \frac{\alpha - 1}{1 + \alpha \frac{T_{\text{mem}} + T_{\text{others}}}{T_{\text{cmp}}(CC)}} \tag{20}$$

$$= 1 + \frac{\alpha - 1}{1 + \alpha \frac{T_{\text{cmp}}(CC) \times \frac{\mathbb{B}}{\mathbb{I}} + T_{\text{others}}}{T_{\text{cmp}}(CC)}} \tag{21}$$

$$< 1 + \frac{\alpha - 1}{1 + \alpha (\frac{\mathbb{B}}{\mathbb{I}})} \tag{22}$$

Tensor Core Upper Bound: For memory-bound kernel, we have $T_{\text{cmp}} \to T_{\text{mem}}$:

$$Speedup < 1 + \frac{\alpha - 1}{1 + \alpha} = 2 - \frac{2}{1 + \alpha} \tag{23}$$

Example: FP64 Nvidia GPUs (with $\alpha = 2$): $Speedup < 1.33$.
Example: Assuming that $\alpha \to \infty$, we have $Speedup < 2$.
Workload Upper Bound: we assume that $\alpha \to \infty$:

$$Speedup < 1 + \frac{\mathbb{I}}{\mathbb{B}} \tag{24}$$

Example: $Speedup_{A100}(\text{GEMV}) < 1.05$.

4.3 Summary

Our analysis covers the two extremes of memory-computation overlap. Real-world kernels typically exhibit partial overlap, resulting in speedups between 1× and 1.33× for double precision. Performance differences beyond that would require memory access optimizations, which, we argue, function equally when applied to tensor and CUDA cores since both access data through the register file (As fig. 1 shows).

Table 1: Specifications of the experimental platforms.

Metric		A100-80GB
CUDA Version		12.6
L2 Cache (MB)		40
Memory Bandwidth (TB/s)		1.94
FP64 Peak (TFLOPS)	CUDA Core	9.7
	Tensor Core	19.5

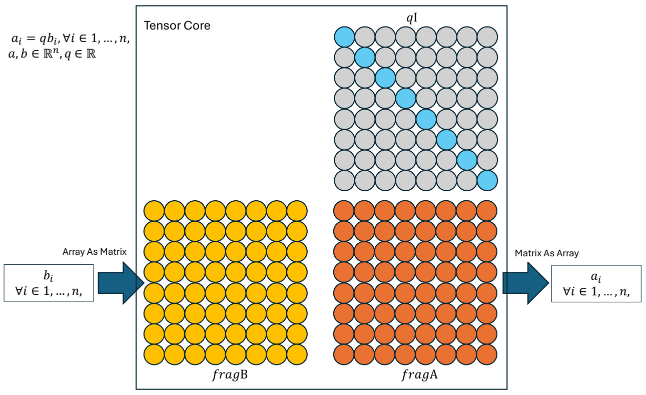

$a_i = qb_i, \forall i \in 1, \dots, n,$
$a, b \in \mathbb{R}^n, q \in \mathbb{R}$
Tensor Core
qI
Array As Matrix
b_i
$\forall i \in 1, \dots, n,$
Matrix As Array
a_i
$\forall i \in 1, \dots, n,$
$frag$B
$frag$A

Fig. 5: Tensor core SCALE implementation.

5. Empirical Analysis

In this section, we conduct empirical analysis to verify our theoretical findings regarding tensor core performance on memory-bound kernels. We seek to understand why recent literatures report superior tensor core performance, which appears to conflict with our theoretical analysis for memory-bound scenarios. Experiments are performed on A100-80GB, detailed in table 1.

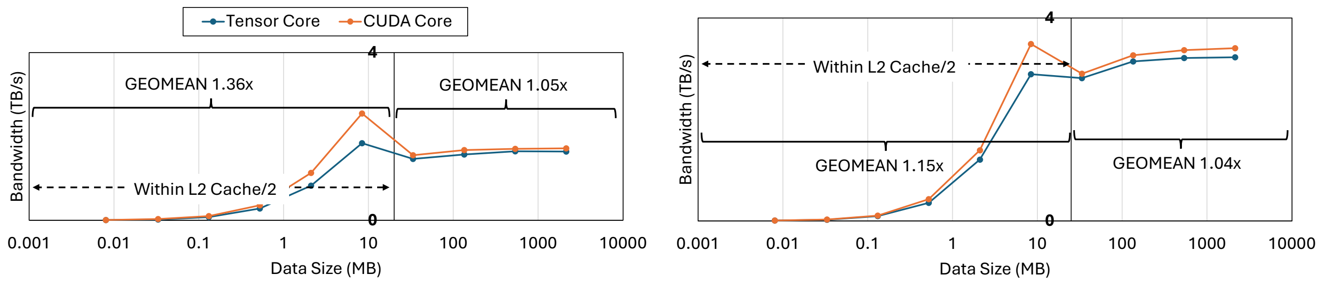
5.1 SCALE

Tensor Core Implementation: Following the approach of work [27], we implement tensor core SCALE as matrix multiplication $A = B(qI)$, where I is the identity matrix, as shown in fig. 5. This implementation has a significant limitation: it utilizes only $\frac{1}{\max(m,n)}$ of the tensor core's compute capacity, where m and n represent the tensor core dimensions. For the 8×4 double precision tensor cores on A100 and H100 GPUs, this means only 1/8 of the available compute power is used, yielding performance of $\mathbb{P}A100(TC, SCALE) = 2.4$ TFLOPS and $\mathbb{P}GH200(TC, SCALE) = 8.37$ TFLOPS. This performance is actually lower than what CUDA cores can achieve. However, given SCALE's extremely low operational intensity $\mathbb{I}$ (as discussed in Section 3.1), this compute underutilization should not significantly impact overall kernel performance, regardless of whether computation and memory operations are overlapped.
CUDA Core Implementation: The CUDA core baseline uses a ChatGPT-generated STREAM implementation[*1], modified only to include warmup iterations.

5.1.1 Evaluation

fig. 6 shows consistent performance degradation when using tensor cores compared to CUDA cores, though the difference is modest. We include results from both A100 and GH200 systems, with the latter featuring the additional Tensor Memory Accelerator (TMA) component.

[*1] `https://chatgpt.com/share/67570ae8-4554-8007-9f91-48f01722af85`



(a) A100 Performance

(b) GH200 Performance

Fig. 6: SCALE performance evaluation on A100 (top) and GH200 (bottom). We show the speedup geometric mean (GEOMEAN) of the CUDA cores over the tensor cores is reported for input domain sizes larger and smaller than half of the L2 cache, respectively.

5.1.2 Analysis of Performance Differences Between CUDA Core and Tensor Core

To understand the observed performance gap, we investigated several potential causes:

Padding computation: While the tensor core implementation does waste computation on padding operations, this should have a negligible impact since SCALE kernels are primarily memory-bound rather than compute-bound.

To quantify this effect, consider a problem of size N with $2N$ computational operations. Let $\mathbb{B}$ represent memory bandwidth, $\mathbb{P}cc$ the CUDA core peak performance, and $\mathbb{P}tc$ the effective tensor core performance. Assuming no overlap between computation and memory operations, the runtimes are: $T_{cc} = \frac{N\times8}{\mathbb{B}} + \frac{N\times2}{\mathbb{P}_{cc}}$ $T_{tc} = \frac{N\times8}{\mathbb{B}} + \frac{N\times2}{\mathbb{P}_{tc}}$ The speedup of CUDA cores over tensor cores becomes: Speedup $= \frac{T_{tc}}{T_{cc}} = \frac{\frac{8}{\mathbb{B}}+\frac{2}{\mathbb{P}_{tc}}}{\frac{8}{\mathbb{B}}+\frac{2}{\mathbb{P}_{cc}}}$ Using A100 specifications ($\mathbb{B}$ = 2 TB/s, $\mathbb{P}cc$ = 19.5 TFLOPS, $\mathbb{P}tc$, eff = 2.5 TFLOPS), we obtain a speedup of 1.17×. For GH200, the speedup is 1.08×.

This analysis confirms that when computation and memory operations cannot be fully overlapped, ***the tensor core implementation's reduced effective computation does indeed harm overall performance.***

Register Usage: Tensor core padding requires additional registers, which could affect occupancy. However, our analysis shows that the CUDA core version consumes 10 registers while tensor core versions consume 20 and 22 registers on SM_80 and SM_90 architectures, respectively. Since register usage below 32 typically doesn't impact occupancy, this is unlikely to explain the performance difference.

TMA: We attempted to leverage TMA to overlap memory access through its advanced API. However, our initial implementation resulted in a 20% performance degradation despite pipeline optimizations.

5.1.3 Conclusion and Future Work

Our analysis reveals that tensor cores exhibit a theoretical disadvantage for SCALE operations, with our implementation confirms that. Although SCALE may not directly correspond to real-world applications, the failure of tensor cores to achieve competitive performance on this fundamental memory-bound kernel raises broader questions about the viability of tensor cores for general memory-bound workloads.

Future work should explore more sophisticated tensor core utilization strategies, overlapping computation, like optimized TMA implementations.

5.2 SpMV

DASP [28] (Tensor Core) & cuSPARSE (CUDA Core): DASP represents the current state-of-the-art tensor core SpMV implementation. It employs a hybrid approach that categorizes matrix rows as long, middle, or small, applying specialized processing strategies for each category, including row sorting for middle-length rows. The original study reported a **1.70×** speedup over cuSPARSE on A100.

cuSPARSE (CUDA Core Implementation): We use cuSPARSE as our CUDA core baseline for comparison.

5.2.1 Experimental Modifications

To ensure fair comparison, we made several critical modifications to the original DASP artifact evaluation setup:

- **CUDA version**: We utilize CUDA 12.6 (AD/AE uses 12.0). While newer CUDA versions may include library improvements, we consider this change to have a negligible impact on our analysis.

- **Warm-up**: We identified insufficient warm-up procedures in the original measurements, which significantly affects performance evaluation for small matrices.

- **cuSparse Pre-processing**: We enabled cuSPARSE's optional preprocessing API [29], which is recommended for iterative sparse matrix-vector operations but was omitted in the original comparison (and many other research).

5.2.2 Warm-up Effect

GPU accelerators require adequate warm-up for accurate performance measurement. Most sparse matrices in the SuiteSparse collection (the dataset used by DASP) are small, with SpMV routines typically completing within 10 ms. Consequently, static iteration-based warm-up protocols prove insufficient.

We replaced the original warm-up procedure with a time-based approach: executing kernels for 1 second before measurement (Listing 1), and the warm-up result is shown in fig. 7. As the figure shown, replacing warm-up can contribute to a geometric mean speedup of 1.2x for DASP in the SpMV workload. This improvement is observed for both DASP and cuSPARSE implementations. This effect is particularly significant for small matrices.

```
// Phase 1: Initial calibration (10 iterations)
float measure_initial_runtime(KernelFunction kernel, KernelArgs args) {
    const int CALIBRATION_RUNS = 10;
    auto start_time = high_resolution_clock::now();
    for (int i = 0; i < CALIBRATION_RUNS; i++) {
        kernel(args);
        synchronize_device(); // Ensure completion
    }
    auto end_time = high_resolution_clock::now();
    float total_time = duration_cast<microseconds>(end_time - start_time).count();
    return total_time / CALIBRATION_RUNS; // Average time per iteration
}
// Phase 2: Warmup until target runtime (1 second) + final measurement
float measure_kernel_runtime(KernelFunction kernel, KernelArgs args) {
    // Phase 1: Calibration
    float avg_iteration_time = measure_initial_runtime(kernel, args);
    // Calculate iterations needed for 1-second warmup
    const float TARGET_WARMUP_TIME = 1000000.0f; // 1 second in microseconds
    int warmup_iterations = static_cast<int>(TARGET_WARMUP_TIME /
      avg_iteration_time);
    // Phase 2: Warmup execution
    for (int i = 0; i < warmup_iterations; i++) {
        kernel(args);
    }
    synchronize_device(); // Complete warmup phase
    // Final measurement after warmup
    auto start_time = high_resolution_clock::now();
    kernel(args);
    synchronize_device();
    auto end_time = high_resolution_clock::now();
    return duration_cast<microseconds>(end_time - start_time).count();
}
```

Listing 1: Two-Phase Kernel Runtime Measurement

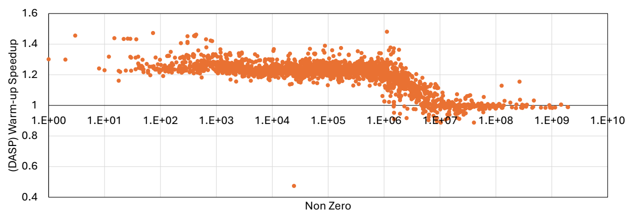


Fig. 7: Speedup of DASP with improved warm-up protocol (our setting) over the original measurement setting. The effect is particularly pronounced for small matrices, where inadequate warm-up significantly impacts performance measurement.

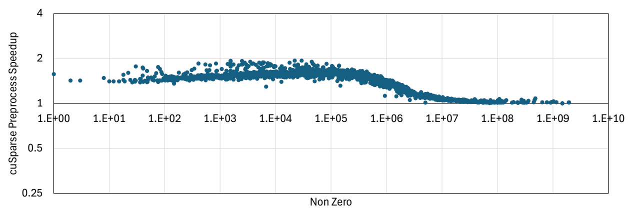


Fig. 8: Performance improvement of cuSPARSE with preprocessing enabled compared to the baseline implementation. Preprocessing provides the most significant benefits for smaller matrices.

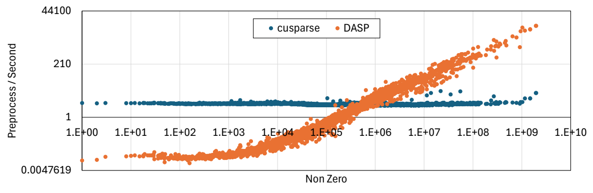


Fig. 9: Preprocessing overhead of cuSPARSE and DASP across different matrix sizes. The overhead of cuSPARSE remains relatively constant regardless of matrix dimensions.

5.2.3 cuSparse Preprocess

While cuSPARSE's preprocessing API is optional, it's recommended for iterative workloads. Despite its importance, few research studies benchmark its performance impact. We analyzed two aspects:

Performance Impact: fig. 8 shows that preprocessing primarily benefits small matrix operations

Overhead Analysis: fig. 9 reveals that cuSPARSE preprocessing overhead remains relatively constant across matrix sizes

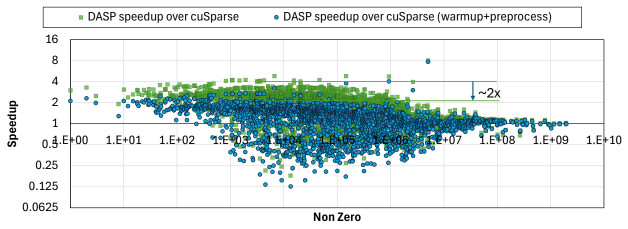


Fig. 10: DASP speedup over cuSPARSE before and after applying our experimental setting (improved warm-up and cuSPARSE preprocessing). Our setting significantly reduce the reported performance advantage (1.70x − > 1.06x).

Table 2: Stencil benchmarks and domain sizes we use. A detailed description of the stencil benchmarks can be found in [30, 31].

	domain(temporal blocking depth)	
	ConvStencil [32]	EBISU [25]
2d5pt	$10240^2(3)$	$9000^2(3)$
2d13pt	$10240^2(1)$	$9000^2(1)$
2d9pt	$10240^2(3)$	$9000^2(3)$
2d49pt	$10240^2(1)$	$9000^2(1)$
3d7pt	$1024^3(3)$	$234 \times 312 \times 2560(3)$
3d27pt	$1024^3(3)$	$234 \times 312 \times 2560(3)$

5.2.4 Revised Performance Comparison

fig. 10 presents the updated DASP vs. cuSPARSE comparison incorporating our modifications. With proper warm-up and cuSPARSE preprocessing enabled, the geometric mean speedup of DASP over cuSPARSE reduces dramatically from 1.57× (close to the originally reported 1.70×) to just 1.06×.

More critically, when considering only matrices with more than 1,000 non-zero elements, the geometric mean speedup becomes 0.95×, indicating that cuSPARSE actually outperforms DASP for larger, more realistic matrices. The majority of DASP's reported performance advantage appears to stem from very small matrices ($non - zeros < 10,000$).

5.2.5 Challenge of Fair Comparison

Conducting fair comparisons between CUDA core and tensor core implementations presents inherent challenges. Ideally, comparisons should use identical data formats to eliminate memory access pattern variations. However, CUDA cores and tensor cores typically require different memory layouts for optimal performance, making direct format comparison impractical.

The most viable approach involves comparing implementations that follow the same design principles, though this introduces variability in implementation quality. What remains clear is that tensor core programming presents significantly greater complexity compared to CUDA core development, potentially limiting adoption and optimization efforts.

5.3 Iterative Stencils

We evaluated stencil implementations using the ConvSten-

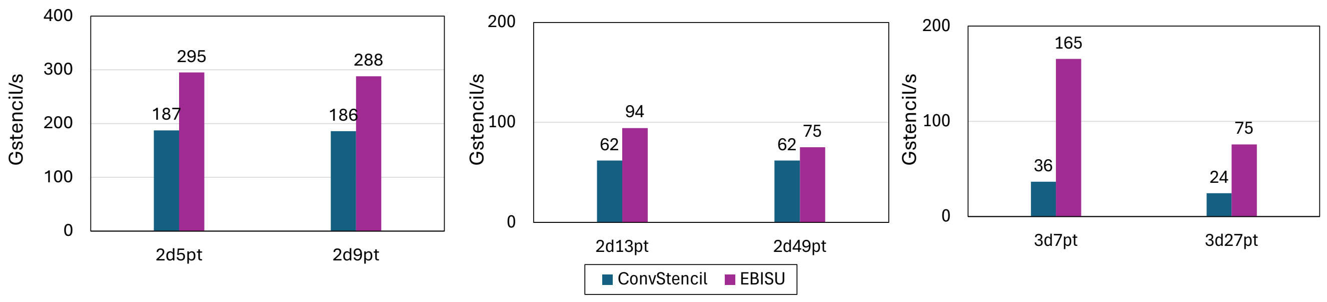


Fig. 11: Comparison of EBISU and ConvStencil on A100 using a suite of stencil benchmarks.

cil [32] benchmark suite (table 2), which provides a comprehensive set of representative stencil patterns.

ConvStencil [32] (Tensor Core): ConvStencil leverages tensor cores by transforming stencil computation into matrix-matrix multiplication, incorporating temporal blocking through kernel fusion. We evaluate using their default configuration and domain sizes.

EBISU [25] (CUDA Core): EBISU is a state-of-the-art CUDA Core implementation that incorporates temporal blocking. To ensure a fair comparison, we configured EBISU's temporal blocking parameters to match those of ConvStencil.

5.3.1 Evaluation

fig. 11 shows that equivalently optimized tensor core implementations generally underperform their CUDA core counterparts across the benchmark suite. This consistent performance gap suggests fundamental limitations in applying tensor cores to stencil computations.

5.3.2 Discussion

Our analysis reveals several inherent disadvantages when using tensor cores for stencil computations:

Memory-Bound Bottleneck: Stencil operations are fundamentally memory-bound workloads. In theory, optimizing computation alone provides minimal performance benefits, as memory bandwidth remains the limiting factor. This characteristic makes the computational advantages of tensor cores largely irrelevant for stencil performance.

Computational Overhead: Implementing stencil operations on matrix processing units requires significant redundant computation. For example, ConvStencil requires 2 times computation to finish the stencil computation, making it unprofitable to using a double precision stencil tensor core. Beyond that, when doing 2d5pt stencil, for example, the effective computation is $3\times27/49$, making it theoretically impossible to surpass CUDA core performance.

Precision Constraints: Most tensor core stencil research prioritizes performance over numerical precision [33–35]. For instance, LoraStencil [34] employs low-rank decomposition—an approximation technique that introduces precision loss to reduce computational requirements. However, precision is critical for scientific computing applications where stencils are commonly used. While techniques like the Ozaki scheme [11] can emulate double precision using integer units, they introduce substantial memory overhead. This creates a fundamental conflict: the Ozaki scheme increases memory requirements, which directly opposes the goal of optimizing memory-bound kernels.

5.3.3 Implications

Given these unresolved fundamental limitations—memory-bound bottlenecks, computational inefficiency, and precision constraints—tensor cores are unsuitable for stencil computations in scientific workloads.

6. Key Takeaways

We highlight the key takeaways from a practical perspective:

- Identifying kernel characteristics (compute-bound/ memory-bound) (Section 3).
- Compute-bound
  - Tensor cores remain advantageous for compute-bound operations.
- Memory-bound
  - Prioritizing CUDA cores for memory-bound kernels due to their simplicity and effectiveness (Section 5).
  - Focusing on memory access optimizations, e.g. cache-aware algorithms and reducing memory traffic [23, 25].
  - Prioritizing pipeline and overlap optimizations before considering tensor core adoption (Section 4.1).
  - Considering theoretical limits: tensor cores provide at most 1.33× speedup in double precision, with a 2× ceiling assuming infinite tensor core computational speedup (Section 4.2).

7. Conclusion

Through systematic theoretical and empirical analysis, we demonstrate that leveraging the tensor cores for computation in memory-bound kernels fails to deliver sound performance benefits. Our theoretical analysis establishes an upper bound of 1.33× speedup for double-precision memory-bound kernels, while empirical results across SCALE, SpMV, and stencil show that tensor core implementations usually underperform their CUDA core counterparts. While these findings may temper expectations for using tensor cores in memory-bound kernels, efforts leveraging tensor cores still provide valuable insights for the design and utilization of matrix processing units in broader contexts.

References

[1] Choquette, J., Giroux, O. and Foley, D.: Volta: Performance and Programmability, *IEEE Micro*, Vol. 38, No. 2, pp. 42–52 (online), DOI: 10.1109/MM.2018.022071134 (2018).

[2] Bhaskaracharya, S. G., Demouth, J. and Grover, V.: Automatic Kernel Generation for Volta Tensor Cores (2020).

[3] Fan, R., Wang, W. and Chu, X.: DTC-SpMM: Bridging the Gap

in Accelerating General Sparse Matrix Multiplication with Tensor Cores, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, New York, NY, USA, Association for Computing Machinery, p. 253–267 (online), DOI: 10.1145/3620666.3651378 (2024).

[4] Okanovic, P., Kwasniewski, G., Labini, P. S., Besta, M., Vella, F. and Hoefler, T.: High Performance Unstructured SpMM Computation Using Tensor Cores, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'24)*, IEEE Press, pp. 154:1–154:14 (online), DOI: 10.1109/SC41406.2024.00060 (2024).

[5] Durrani, S., Chughtai, M. S., Hidayetoglu, M., Tahir, R., Dakkak, A., Rauchwerger, L., Zaffar, F. and Hwu, W.-m.: Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles, *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 345–355 (online), DOI: 10.1109/PACT52795.2021.00032 (2021).

[6] Pisha, L. and Ligowski, Ł.: Accelerating non-power-of-2 size Fourier transforms with GPU Tensor Cores, *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 507–516 (online), DOI: 10.1109/IPDPS49936.2021.00059 (2021).

[7] Haidar, A., Tomov, S., Dongarra, J. and Higham, N. J.: Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers, *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, pp. 603–613 (2018).

[8] Lu, Y., Zeng, L., Wang, T., Fu, X., Li, W., Cheng, H., Yang, D., Jin, Z., Casas, M. and Liu, W.: Amgt: Algebraic multigrid solver on tensor cores, *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*, IEEE Computer Society, pp. 823–838 (2024).

[9] Wu, D., Chen, P., Wang, X., Lyngaas, I., Miyajima, T., Endo, T., Matsuoka, S. and Wahib, M.: Real-time High-resolution X-Ray Computed Tomography, *Proceedings of the 38th ACM International Conference on Supercomputing*, ICS '24, New York, NY, USA, Association for Computing Machinery, p. 110–123 (online), DOI: 10.1145/3650200.3656634 (2024).

[10] Ootomo, H. and Yokota, R.: Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance, *The International Journal of High Performance Computing Applications*, Vol. 36, No. 4, pp. 475–491 (2022).

[11] Ootomo, H., Ozaki, K. and Yokota, R.: DGEMM on integer matrix multiplication unit, *The International Journal of High Performance Computing Applications*, p. 10943420241239588 (2024).

[12] Sun, W., Li, A., Geng, T., Stuijk, S. and Corporaal, H.: Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 34, No. 1, pp. 246–261 (online), DOI: 10.1109/TPDS.2022.3217824 (2023).

[13] Luo, W., Fan, R., Li, Z., Du, D., Wang, Q. and Chu, X.: Benchmarking and Dissecting the Nvidia Hopper GPU Architecture , *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 656–667 (online), DOI: 10.1109/IPDPS57955.2024.00064 (2024).

[14] Abdelkhalik, H., Arafa, Y., Santhi, N. and Badawy, A.-H. A.: Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis, *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8 (online), DOI: 10.1109/HPEC55821.2022.9926299 (2022).

[15] Austin, B., Kulkarni, D., Cook, B., Williams, S. and Wright, N. J.: System-Wide Roofline Profiling-a Case Study on NERSC's Perlmutter Supercomputer, *PMBS24: The 15th International Workshop on Performance Modeling, Benchmarking, and Simulation of High-Performance Computer Systems* (2024).

[16] Domke, J., Vatai, E., Drozd, A., ChenT, P., Oyama, Y., Zhang, L., Salaria, S., Mukunoki, D., Podobas, A., WahibT, M. and Matsuoka, S.: Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?, *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1056–1065 (online), DOI: 10.1109/IPDPS49936.2021.00114 (2021).

[17] McCalpin, J. D. et al.: Memory bandwidth and machine balance in current high performance computers, *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, Vol. 2, No. 19-25 (1995).

[18] Williams, S., Waterman, A. and Patterson, D.: Roofline: an insightful visual performance model for multicore architectures, *Communications of the ACM*, Vol. 52, No. 4, pp. 65–76 (2009).

[19] Ofenbeck, G., Steinmann, R., Caparros, V., Spampinato, D. G. and Püschel, M.: Applying the roofline model, *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, pp. 76–85 (2014).

[20] Yang, C., Kurth, T. and Williams, S.: Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system, *Concurrency and Computation: Practice and Experience*, Vol. 32, No. 20, p. e5547 (2020).

[21] McCalpin, J. D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers, Technical report, University of Virginia, Charlottesville, Virginia (1991-2007).

[22] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, p. 451–460 (online), DOI: 10.1145/1816038.1816021 (2010).

[23] Zhang, L., Wahib, M., Chen, P., Meng, J., Wang, X., Endo, T. and Matsuoka, S.: PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications, *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, New York, NY, USA, Association for Computing Machinery, p. 167–179 (online), DOI: 10.1145/3577193.3593705 (2023).

[24] Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S. and Dubach, C.: High performance stencil code generation with lift, *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 100–112 (2018).

[25] Zhang, L., Wahib, M., Chen, P., Meng, J., Wang, X., Endo, T. and Matsuoka, S.: Revisiting Temporal Blocking Stencil Optimizations, *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, New York, NY, USA, Association for Computing Machinery, p. 251–263 (online), DOI: 10.1145/3577193.3593716 (2023).

[26] Matsumura, K., Zohouri, H. R., Wahib, M., Endo, T. and Matsuoka, S.: AN5D: automated stencil framework for high-degree temporal blocking on GPUs, *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, New York, NY, USA, Association for Computing Machinery, p. 199–211 (online), DOI: 10.1145/3368826.3377904 (2020).

[27] Navarro, C. A., Carrasco, R., Barrientos, R. J., Riquelme, J. A. and Vega, R.: GPU tensor cores for fast arithmetic reductions, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 1, pp. 72–84 (2020).

[28] Lu, Y. and Liu, W.: DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3581784.3607051 (2023).

[29] NVIDIA: cuSPARSE Library: cusparsespmv, `https://docs.nvidia.com/cuda/archive/12.0.0/cusparse/index.html#cusparsespmv` (2022).

[30] Zhao, T., Basu, P., Williams, S., Hall, M. and Johansen, H.: Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–44 (2019).

[31] Rawat, P. S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.-N. and Sadayappan, P.: Effective resource management for enhancing performance of 2D and 3D stencils on GPUs, *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pp. 92–102 (2016).

[32] Chen, Y., Li, K., Wang, Y., Bai, D., Wang, L., Ma, L., Yuan, L., Zhang, Y., Cao, T. and Yang, M.: ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores, *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, New York, NY, USA, Association for Computing Machinery, p. 333–347 (online), DOI: 10.1145/3627535.3638476 (2024).

[33] Liu, X., Liu, Y., Yang, H., Liao, J., Li, M., Luan, Z. and Qian, D.: Toward accelerated stencil computation by adapting tensor core unit on gpu, *Proceedings of the 36th ACM International Conference on Supercomputing*, pp. 1–12 (2022).

[34] Zhang, Y., Li, K., Yuan, L., Cheng, J., Zhang, Y., Cao, T. and Yang, M.: LoRAStencil: Low-Rank Adaptation of Stencil Computation on Tensor Cores , *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 839–855 (online), DOI: 10.1109/SC41406.2024.00059 (2024).

[35] Han, H., Li, K., Cui, W., Bai, D., Zhang, Y., Yuan, L., Chen, Y., Zhang, Y., Cao, T. and Yang, M.: FlashFFTStencil: Bridging Fast Fourier Transforms to Memory-Efficient Stencil Computations on Tensor Core Units, *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 355–368 (2025).