RESEARCH-ARTICLE

MatXtract: Sparsity-Aware Matrix Transformation via Cascaded Compute Density EXtraction for SpMV

Accepted: 18 January 2026
Revised: 07 January 2026
Received: 23 November 2025

MatXtract: Sparsity-Aware Matrix Transformation via Cascaded Compute Density EXtraction for SpMV

LUHAN WANG, School of Computer Science, Peking University, Beijing, China
KUN LI[*], Institute for AI Industry Research, Tsinghua University, Beijing, China
YIFENG CHEN, Peking University, Beijing, China
HAIPENG JIA, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China
YUNQUAN ZHANG, Institute of Computing Technology Chinese Academy of Sciences, Beijing, China
TING CAO, Institute for AI Industry Research, Tsinghua University, Beijing, China
YUNXIN LIU, Institute for AI Industry Research, Tsinghua University, Beijing, China

Sparse Matrix-Vector multiplication (SpMV) is a fundamental kernel across scientific computing and graph analytics. Modern GPUs feature specialized processing units such as Tensor Core Units (TCUs) for accelerating dense matrix operations. While recent efforts leverage TCUs for SpMV acceleration, direct TCU-only mappings suffer from poor cache utilization and limited performance generalization across diverse sparse matrices. This paper presents MatXtract[1], a sparsity-aware matrix transformation framework that restructures sparse matrices through cascaded compute density extraction, enabling efficient TCU-accelerated SpMV. MatXtract introduces three key components: 1) *Compute Density Extraction* optimizes memory hierarchy utilization while creating Tensor Core- and CUDA core-friendly computation substrates to enhance TCU compute density and CUDA core load balancing; 2) *Cascaded Execution Kernel* employs a two-stage cascaded compression that systematically exploits *hierarchical sparsity* to maximize GPU computational throughput; and 3) *Sparsity-Aware Predictor* mitigates the mismatch between fixed TCU computation and diverse sparsity through Bayesian optimization for enhanced performance generalization. Extensive evaluations on 2,059 matrices show that MatXtract outperforms cuSPARSE, DASP, CSR5, and Merge-SpMV, achieving higher performance on 96.64% of matrices, with an average 1.98× and up to 8.83× speedup over cuSPARSE on NVIDIA A100 GPUs.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Vector / streaming algorithms**.

Additional Key Words and Phrases: SpMV, Auto-tuner, Tensor Core Units, Matrix Multiplication, GPU

[*]corresponding author.
[1]New Paper, Not an Extension of a Conference Paper.

This work is supported by National Key Research & Development Program of China under Grant 2023YFB3001700 and National Natural Science Foundation of China under Grant 62372432.
Authors' Contact Information: Luhan Wang, School of Computer Science, Peking University, Beijing, Beijing, China; e-mail: iswangluhan@foxmail.com; Kun Li, Institute for AI Industry Research, Tsinghua University, Beijing, Beijing, China; e-mail: likungw@gmail.com; Yifeng Chen, Peking University, Beijing, Beijing, China; e-mail: iswanglh@gmail.com; Haipeng Jia, Institute of Computing Technology Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: jiahaipeng@ict.ac.cn; Yunquan Zhang, Institute of Computing Technology Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: 1422665992@qq.com; Ting Cao, Institute for AI Industry Research, Tsinghua University, Beijing, Beijing, China; e-mail: isbella_bing@163.com; Yunxin Liu, Institute for AI Industry Research, Tsinghua University, Beijing, Beijing, China; e-mail: 2292628492@qq.com.

ACM 1544-3973/2026/1-ART
https://doi.org/10.1145/3793864

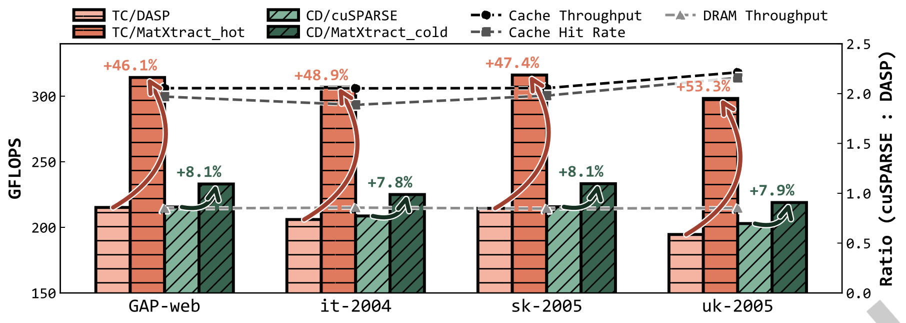


Fig. 1. SpMV performance (bars, GFLOPS) and memory access behavior (lines, Ratio). Legend format: [GPU core type/implementation], where TC denotes Tensor Core and CD denotes CUDA core. For implementations, MatXtract_hot and MatXtract_cold represent our computations on hot and cold regions using Tensor Cores and CUDA cores, respectively.

1 Introduction

Sparse Matrix-Vector multiplication (SpMV) is a cornerstone kernel in high-performance computing (HPC), underpinning a wide range of applications, including scientific simulations [19], graph algorithms [17], and sparse linear solvers [20]. However, despite its ubiquity, SpMV remains one of the most challenging kernels to optimize, largely due to its irregular memory access patterns and low arithmetic intensity. As one of the Seven Dwarfs in HPC [3], SpMV epitomizes the difficulty of accelerating sparse computations efficiently on modern architectures, which are increasingly optimized for dense, high-throughput workloads.

The rise of Tensor Core Units (TCUs) presents a promising avenue for SpMV acceleration by exploiting high-throughput Matrix Multiply-Accumulate (MMA) operations. Recent works [9, 48] have explored TCU-only SpMV implementations, seeking to exploit the raw computational power of TCUs. However, these approaches fall short of addressing the memory-bound nature of SpMV. While TCUs effectively accelerate floating-point computations, loading sparse data into TCUs incurs inefficient cache utilization, often offsetting FLOP efficiency gains.

Figure 1 illustrates this problem with four representative matrices on A100: CUDA core-based cuSPARSE achieves higher GFLOPS than TCU-based DASP [48], despite TCUs' superior theoretical FLOP efficiency. This discrepancy arises primarily from DASP's ***poor cache utilization***: on average, cuSPARSE achieves 2.09× higher cache throughput, 1.99× better cache hit rate, and 0.85× lower DRAM traffic than DASP. Beyond cache inefficiency, current TCU approaches also exhibit ***limited performance generalization***. In an extensive evaluation across 2,059 matrices, DASP outperforms CUDA 12.8's cuSPARSE in only 81.2% of cases, while Spaden [9] is effective only when the average nonzeros per row exceeds 32, a condition met by merely 19.8% of matrices in the SuiteSparse Matrix Collection [18].

The root causes of these inefficiencies stem from three limitations of TCU-only SpMV approaches. ***First***, vast irregular memory accesses to the input vector **x** persist. The sparse matrix structure inherently generates scattered access patterns that the TCU cannot address, resulting in inefficient global memory accesses. ***Second***, the sparse matrix is entirely loaded from global memory into MMA register fragments, leading to heavy register pressure and preventing efficient caching of massive intermediate results. ***Third***, TCU architectures lack computational flexibility. Sparse matrices exhibit diverse sparsity patterns that conflict with TCUs' fixed dense computation mode, limiting optimization opportunities and reducing performance portability.

These limitations underscore a fundamental issue: Blindly offloading SpMV to TCUs fails to align with irregular memory access, restricting both performance and generalization. The key question, then, is not merely how to

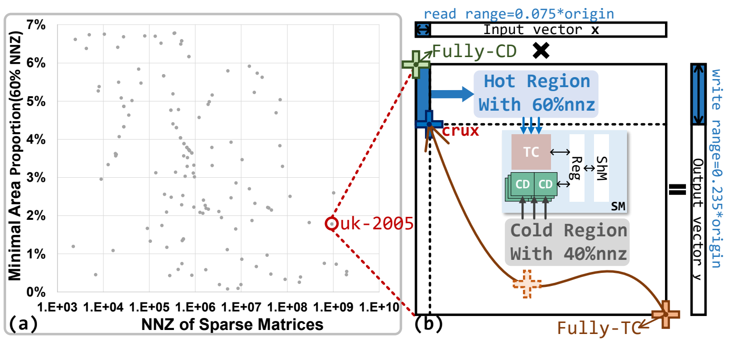


Fig. 2. Extracted compact partition of sparse matrices (each point in (a) represents a sparse matrix). (a) Area proportion of compact partition (with 60% nonzeros) to the original matrix. (b) High-density region schematic of UK-2005 matrix. The hot region (blue) contains 60% of nonzeros (nnz) within only 1.76% of the matrix area (0.235 height×0.075 width), while the cold region holds the remaining 40% nnz sparsely distributed across 98.24% of the area. The crux point marks the threshold separating these two regions.

force-fit sparse computations into dense accelerators, but ***how to structure, extract, and leverage sparsity*** to fully unlock these accelerators' computational potential in modern architectures.

In this paper, we propose a sparsity-aware matrix transformation system, MatXtract, designed to dynamically restructure computational sparsity via cascaded compute density extraction for SpMV. It intelligently structures and extracts matrix portions well-suited for TCU acceleration while enabling better load balancing for the remaining computations, thereby improving overall SpMV performance.

The design of MatXtract is grounded in four crucial observations: ***First***, *sparse matrices exhibit highly skewed compute density*. Many real-world sparse matrices follow a power-law distribution [31], where a small subset of rows and columns accounts for the majority of the computational load. Our analysis shows that for numerous sparse matrices, 60% of nonzeros can be extracted into a compact partition occupying less than 5% of the total matrix area (Figure 2(a)). ***Second***, *higher compute density significantly improves memory efficiency while leveraging TCU's inherent computational advantages, making acceleration highly effective*. Taking the uk-2005 matrix as an example, isolating the high-density region reduces TCU accesses to the input vector to just 7.5% of the original elements, effectively alleviating the memory wall problem and activating the high-throughput potential of TCU (Figure 2(b)). ***Third***, *compute density can be structurally extracted by leveraging the permutation invariance of SpMV*. The permutation invariance property allows rows and columns to be permuted without altering correctness, enabling a structured transformation that reorganizes sparse matrices to expose high-compute-density regions. ***Fourth***, *the extracted high-density regions exhibit hierarchical sparsity characteristics*. Our analysis demonstrates that these regions still retain considerable internal sparsity (Section 4), which can be further exploited through cascaded compression techniques.

Guided by these observations, the core insight behind MatXtract emerges: SpMV performance can be improved through sparsity-aware extraction that identifies and concentrates computation-intensive regions onto TCUs. To achieve this, MatXtract comprises three key techniques:

MatXtract employs *Compute Density Extraction* (CDE) to optimize memory hierarchy utilization by restructuring sparse matrices into TCU- and CUDA core-friendly computation substrates. CDE introduces the crux point to separate compute-dense (hot) and compute-sparse (cold) regions. A column-row permutation then reorganizes the matrix structure to improve locality and density while preserving mathematical equivalence. The resulting

Table 1. Comparison with state-of-the-art SpMV implementations.

	Optimization		**Generalization**
Method	CUDA Core	Tensor Core	Auto-tuning
cuSPARSE [60]	✓	✗	✗
CSR5 [45]	✓	✗	✗
Merge-SpMV [51]	✓	✗	✗
DASP [48]	✗	✓	✗
Spaden [9]	✗	✓	✗
MatXtract	✓	✓	✓

hot sub-matrix concentrates on TCU while the cold remainder utilizes CUDA cores, enhancing compute density and load balancing respectively.

MatXtract utilizes the *Cascaded Execution Kernel* (CEK) to exploit *hierarchical sparsity* beyond CDE through cascaded optimizations. CEK designs a two-stage cascaded compression for CDE-extracted hot regions that maximizes TCU computational throughput. The first-cascade TCU-aware compression mitigates structural sparsity, followed by the second-cascade dual-mode compression that optimizes data layout according to density levels. Meanwhile, CEK deploys memory-coalesced CUDA kernels to leverage the enhanced uniformity of cold-region non-zero distributions.

MatXtract involves the *Sparsity-Aware Predictor* (SAP) to resolve the conflict between the fixed TCU computation mode and diverse sparsity patterns. SAP constructs a 2D search space defined by the hot-cold threshold point crux to improve sensitivity for varying sparsity characteristics. To efficiently navigate this search space and locate the optimal solution, SAP integrates a Bayesian optimization-driven crux generation module with tree-based crux prediction.

Table 1 summarizes the processing units and auto-tuning techniques employed by different works for performance optimization and generalization. We evaluate MatXtract on 2,059 sparse matrices from the SuiteSparse Matrix Collection [18] using NVIDIA A100 GPUs, and compare against both vendor-optimized and state-of-the-art SpMV implementations, including cuSPARSE [60], DASP [48], CSR5 [45], and Merge-SpMV [51]. MatXtract demonstrates competitive performance across diverse sparsity patterns, outperforming cuSPARSE on 96.64% of matrices (vs. DASP's 81.2% success rate), achieving up to 8.83× speedup over cuSPARSE, and up to 8.07× speedup over TCU-only DASP in both FP16 and FP64 precision. The rest of this paper is organized as follows. Section 2 introduces the background on SpMV, Tensor Core Units, and challenges in extracting compute-dense subregions. Sections 3, 4, and 5 present the design of compute density extraction, cascaded execution kernel, and sparsity-aware predictor, respectively. Section 6 evaluates MatXtract's performance against other baselines. Section 7 discusses related work. Section 8 concludes with future research directions.

2 Background

2.1 SpMV Computation

Sparse Matrix-Vector multiplication, denoted as $\mathbf{y} = \mathbf{Ax}$, where $\mathbf{A}$ is a sparse matrix and $\mathbf{x}$ and $\mathbf{y}$ are dense vectors, is fundamental to many algorithms. Since the sparsity pattern of $\mathbf{A}$ typically remains fixed throughout iterations, specialized matrix formats can be adopted to amortize preprocessing costs and enable optimization [21]. For the widely used Compressed Sparse Row (CSR) format, processing each non-zero element involves four memory access operations: Retrieving the row pointer, fetching the column index, accessing the matrix value, and reading $\mathbf{x}$ based on the column index. Due to non-contiguous column indices, access to $\mathbf{x}$ is inherently random, resulting in poor data locality. This combination of intensive memory operations and irregular access patterns challenges SpMV optimization on GPUs.

2.2 Tensor Core Units

Modern NVIDIA GPUs are highly parallel computing platforms with numerous Streaming Multiprocessors (SMs). Each SM includes conventional CUDA cores (e.g., Floating-point Units) and specialized Tensor Core Units. Tensor Cores, first integrated into NVIDIA Volta architecture [56], are specialized hardware units tailored for matrix multiplication tasks in domains like deep learning. These units execute Matrix Multiply-Accumulate (MMA) operations: $frag_D = frag_A \times frag_B + frag_C$, where $frag_A$ and $frag_B$ are input matrices with dimensions $m \times k$ and $k \times n$, respectively, $frag_C$ is the accumulator, and $frag_D$ is the result matrix. CUDA provides two programming interfaces for TCUs: the high-level C++ WMMA APIs [58], which abstract away register management complexities using data fragments, and the low-level PTX MMA APIs [59], which offer developers finer control over register layouts. Our work focuses on the PTX level. For example, mma.m16n8k8 computes the product of a 16×8 left-hand operand and an 8×8 right-hand operand within a single warp.

2.3 Challenges

While extracting compute-dense subregions offers a promising path to unlock TCU acceleration for SpMV, our design is fundamentally shaped by three technical challenges arising from the nature of real-world sparsity:

Extreme intra-matrix sparsity irregularity. While many real-world sparse matrices contain regions of relatively high local density, these regions are typically latent, fragmented, and deeply embedded within globally sparse structures—often exceeding 99%+ sparsity [28]. This fragmentation leads to low surface compute density, making it difficult to isolate accelerator-friendly blocks without global structural reorganization [37]. Even after applying condensation techniques such as Sparse Graph Translation [71], the resulting matrices often retain over 90% sparsity [25], further highlighting the challenge of directly extracting usable dense regions.

Severe density variation within extracted regions. Even after isolating compute-dense regions, our analysis shows that these subregions exhibit significant internal variation, with the density ranging from as low as 6.1% to as high as 99.3% (Figure 4(b) in Section 4). This wide variability makes it infeasible to design a single TCU kernel that performs efficiently across all density levels. Without further refinement, many extracted regions fall into a *sparsity gray zone* that underutilizes sparse kernel optimizations.

Significant inter-matrix sparsity diversity. Sparse matrices vary widely in structure and density distribution, posing substantial challenges for generalizable optimization [76]. While traditional SpMV auto-tuners tailored for CPUs [44, 65, 75] and CUDA cores [6, 7, 24, 54, 62, 65, 72] leverage a rich set of storage formats and kernel variants, ***TCU-based execution is constrained by a rigid dense compute mode with limited format support***. This restricts the tuning search space and complicates performance portability across matrices, especially when targeting diverse real-world workloads.

MatXtract contains three key components to address the above challenges: ***Compute Density Extraction*** exposes latent high-density regions in extremely irregular sparse matrices; ***Cascaded Execution Kernel*** further leverages the *hierarchical sparsity* beyond CDE through a two-stage cascaded compression; ***Sparsity-Aware Predictor*** bridges the gap between diverse sparse patterns and fixed dense accelerator utilization, supporting adaptive configuration selection. The remainder of this section details each component.

3 Compute Density Extraction

Compute Density Extraction (CDE) systematically restructures sparse matrices into compute-dense hot regions for TCU execution and compute-sparse cold regions for CUDA core processing through column-row permutation.

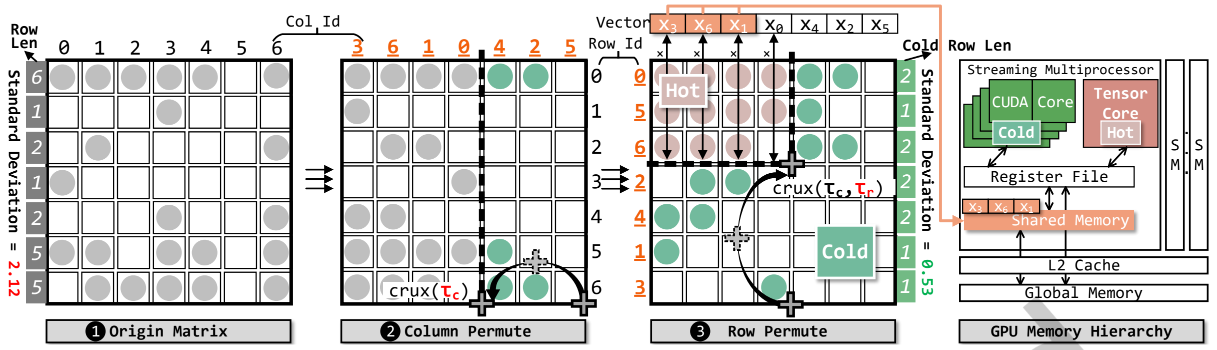


Fig. 3. Compute density extraction. The crux (τ_c, τ_r) derived from Sparsity-Aware Predictor guides the column and row permutation, where $nnz = 22$, $\tau_c = 0.77$ and $\tau_r = 0.5$.

3.1 Hot-Cold Region Isolation

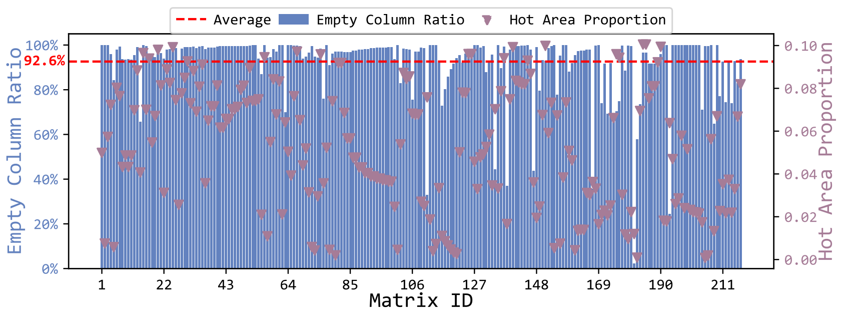
To prioritize memory access locality for the input vector **x**, CDE begins by sorting the columns of the sparse matrix in descending order based on their non-zero counts (column lengths). From this sorted sequence, the method selects the top c columns that collectively encompass $\lceil \tau_c \times nnz \rceil$ nonzeros (❷ in Figure 3), where nnz represents the total number of nonzeros in the original matrix and τ_c represents the column coverage ratio. Subsequently, within the submatrix formed by these c columns, the method sorts the rows according to their local row lengths and extracts the top r rows that cover $\lceil \tau_r \times nnz \rceil$ nonzeros (❸ in Figure 3), where τ_r represents the row coverage ratio. This resulting $r \times c$ submatrix is called the hot region, accounting for a fraction τ_r of the total nonzeros. The parameter pair (τ_c, τ_r), which separates the hot and cold regions, is defined as the crux. Since the hot region is a subset of the selected c columns, it follows that $0 \le \tau_r \le \tau_c \le 1$.

Computing the column length is efficient: it requires only a single subtraction at the corresponding indices in the `colPtr` array of the Compressed Sparse Column (CSC) format. For the submatrix comprising the selected c columns, CDE converts its CSC representation into the CSR format. Through subtractions within the `rowPtr` array of the CSR format, row lengths in this submatrix are efficiently determined. It is worth noting that MatXtract requires the matrix in CSC or CSR format (either one suffices). CSC is used for efficient global column length computation, but is not strictly required. CSR-only inputs are feasible. Furthermore, many high-level frameworks provide CSC by default, such as Ligra [64], GraphBLAS [17], and Gunrock [69].

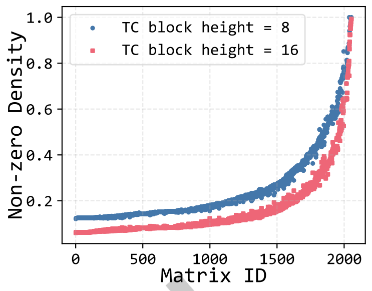
3.2 Tensor-Scalar Hybrid Optimization

After isolation, CDE leverages MMA-based TCUs to process the high-compute-density hot region, while the remaining cold region is handled using scalar computations on CUDA cores. This tensor-scalar hybrid execution effectively exploits ***improved cache utilization and TCU's higher computational throughput for hot regions, complemented by more uniform workloads and better load balancing for cold regions*** to achieve acceleration across both regions.

3.2.1 Hot region TCU acceleration. In the hot region, CDE improves memory access locality for the input vector **x** and increases the non-zero density within Tensor Core (TC) blocks, thereby maximizing the TCU computational efficiency. Here, *TC blocks* refer to dense tiles with the same shape as matrix fragments of the TCU MMA API (i.e., 16 × 8), where the size of the first dimension is called the *TC block height* (i.e., 16). Figure 3 ❸ shows that, for the TCU-accelerated hot region, our method prioritizes sorting columns by length, enabling the placement of high-reuse input vector elements into shared memory, improving memory access locality for **x**. As for the TCU



(a) Empty column ratio in extremely high density hot regions (area proportion < 0.10 and non-zero proportion > 60%)



(b) TC block density levels after TCA with two different TC block heights

Fig. 4. Hierarchical sparsity in hot regions. (a) Structural sparsity within hot regions. (b) Density level variations across different matrices after TCU-aware compression.

efficiency optimization, our strategy of concentrating more nonzeros into a smaller region naturally increases the density of nonzeros within TC blocks in the *dense unfolding* format (detailed in Section 4.2.1). Furthermore, the subsequent row sorting ensures that rows in the hot region are arranged in descending order of their non-zero counts. This similarity in adjacent row lengths significantly reduces zero padding during *sparse encoding* (Section 4.2.2).

3.2.2 Cold region load balancing enhancement. CDE benefits TCUs and CUDA cores concurrently. After extracting TCU-friendly high-compute-density regions, the remaining cold regions processed by CUDA cores exhibit better load balancing, leading to improved *Warp Occupancy* (Average 1.2× compared to the entire matrix in Figure 6). This enhancement stems from the more uniform non-zero distribution in the post-CDE cold region, as illustrated by the *Row Length Standard Deviation* being reduced by 75% from 2.12 (❶ in Figure 3) to 0.53 (❸ in Figure 3) compared to the original sparse matrix.

4 Cascaded Execution Kernel

Although CDE effectively identifies computationally concentrated hot regions, our analysis reveals two *hierarchical sparsity* phenomena: 1) substantial structural sparsity persists within hot regions (Figure 4(a)), and 2) significant density level variations exist across different matrices (Figure 4(b)). These issues demand progressive optimization beyond the CDE. This section introduces the Cascaded Execution Kernel (CEK), which implements two compression stages on CDE-identified hot regions and employs memory-coalesced CUDA kernels for cold regions with more uniform non-zero distributions.

4.1 First-Cascade TCU-Aware Compression

Sparse matrices in Figure 2(a) exhibit hot regions that occupy less than 10% area but account for up to 60% nonzeros. Figure 4(a) shows that when these dense hot regions are partitioned into chunks (groups of consecutive rows with height equal to the TC block height) using a TC block height of 8, on average 92.6% of the columns inside the chunks contain no nonzeros. These empty columns can be safely ignored during TCU loading without affecting computational correctness. We refer to this intrinsic property as *structural sparsity*.

The TCU-aware compression shown in Figure 5(a) reduces the structural sparsity. The term *TCU-aware* implies that the density improvement after compression is closely related to the TC block height. Specifically, when the TC block height is set to 8, the compressed density is considerably higher than when it is 16 (Figure 4(b)).

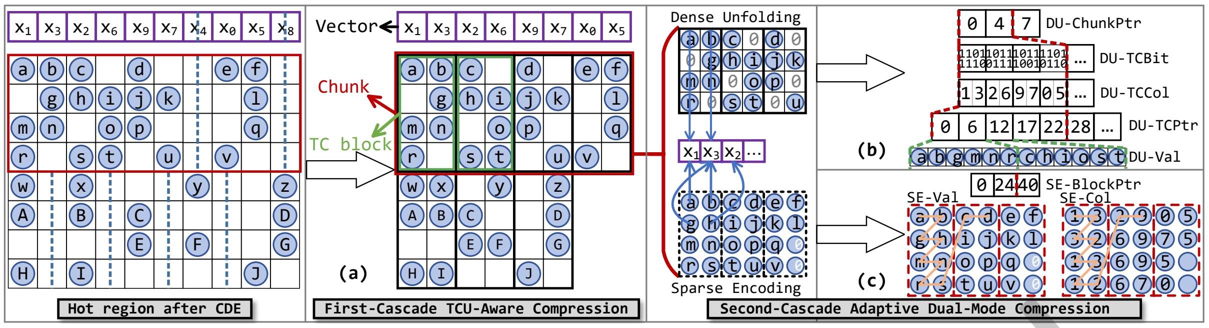


Fig. 5. Two-stage cascaded compression: TCU-aware compression, *Dense unfolding* (DU) and *sparse encoding* (SE) compression on the hot region. A chunk represents a row-block covering TC block height rows and comprises multiple 4×2 TC blocks, with relevant components highlighted in red.

This is because smaller chunks with fewer rows (smaller TC block height) provide finer granularity and reduce the likelihood of non-zero entries appearing in any given column within the chunk. This ultimately enables TCU-aware compression to discard empty columns more effectively and improve density (Figure 4(b) 8 vs. 16). For the FP64 data type, we use the mma.m8n8k4 instruction, which corresponds to a TC block height of 8. In contrast, mma.m16n8k8 is more efficient for FP16. We apply an *implicit register-level transposition* to accommodate the m16n8k8 MMA layout while maintaining high compressed density–that is, preserving the TC block height of 8.

TCU-aware compression is designed to maximize density by aggressively removing empty columns, with its effectiveness depending on the specific distribution of empty columns in the matrix. As shown in Figure 4(b), the density level of hot regions in different sparse matrices varies significantly after TCU-aware compression. This variation highlights the need for second-cascade compression—Adaptive Dual-Mode Compression.

4.2 Second-Cascade Adaptive Dual-Mode Compression

Building upon the compressed hot regions from the previous stage, second-cascade adaptive dual-mode compression includes two complementary compressed storage formats that adapt to the diverse density levels:

4.2.1 Dense unfolding for high density. Dense unfolding (DU) inserts explicit zeros between nonzeros at specific positions, transforming the original sparse format into a dense format that maintains the original non-zero relative positional relationships. As shown in Figure 5(b), this format uses five arrays: 1) `ChunkPtr` stores the starting TC block index for each chunk; 2) `TCBit` encodes the non-zero layout as bitmaps within each TC block; 3) `TCCol` contains column indices for each TC block column; 4) `TCPtr` records the starting index within each TC block in the `Val` array, and 5) `Val` stores non-zero values in row-major order within TC blocks. *Dense unfolding* enables efficient reuse of both the input vector and `TCCol` arrays through explicit column index alignment. This format excels in scenarios with high matrix density, as the column index storage provides higher reuse.

Densely unfolding sparse blocks in shared memory wastes significant bandwidth and capacity due to zero-padding. We bypass shared memory, fetching nonzeros directly from global memory into their corresponding register locations. Using bitwise operations (Algorithm 1, Line 13) and the `__popcll` hardware intrinsic (Line 14), we decode the sparse format on-the-fly: non-zero values (val_{off}) are placed into their correct register positions, while zeros are implicitly synthesized through conditional assignment. For FP16, we propose an *implicit register-level transposition* algorithm, which exploits the *m16n8k8* MMA instruction without incurring transposition overhead while maintaining high compression density. Specifically, line 14 of Algorithm 1 loads sparse matrix TC

Algorithm 1: FP16 *dense unfolding* kernel with implicit transposition

```
   Input: $\mathbf{x}[]$, $chunkPtr[]$, $tcBit[]$, $tcCol[]$, $tcPtr[]$, $val[]$, $fragM|K|N$
   Output: $\mathbf{y}[]$
1  Initialize $warpsPerBlock \leftarrow 4$;
2  $warpIdx \leftarrow threadIdx.x \gg 5$; $laneIdx \leftarrow threadIdx.x \;\&\; 31$;
3  $groupIdx \leftarrow laneIdx \gg 2$; $threadId\_in\_group \leftarrow laneIdx \;\&\; 3$;
4  $chunkIdx \leftarrow blockIdx.x \times warpsPerBlock + warpIdx$;
5  $tcBlockStart \leftarrow chunkPtr[chunkIdx]$; $tcBlockEnd \leftarrow chunkPtr[chunkIdx + 1]$;
6  $acc[0:1] \leftarrow \{0\}$; $sum0, sum1 \leftarrow 0$;
7  $b\_col \leftarrow groupIdx$
8  for $tcBlockIdx \in [tcBlockStart, tcBlockEnd)$ do
9      $bitmap \leftarrow tcBit[tcBlockIdx]$; $val_{\text{off}} \leftarrow val + tcPtr[tcBlockIdx]$;
10     for $i \in \{0, 1\}$ do
11         $b\_row \leftarrow threadId\_in\_group \times 2 + i$;
12         $b\_bitPos \leftarrow b\_col \times fragK + b\_row$ ;                       ▷ Implicit transposition
13         $bit \leftarrow (bitmap \gg b\_bitPos) \;\&\; 1$;
14         $frag_B[i] = bit \;?\; val_{\text{off}}[\texttt{\_\_popcll}(bitmap\&((1 << b\_bitPos) - 1))] : 0.0$ ;   ▷ Load matrix to B fragment
15     end
16     $tcColPtr \leftarrow tcCol + tcBlockIdx \times fragK$;
17     for $i \in \{0, 1\}$ do
18         $x\_idx \leftarrow tcColPtr[threadId\_in\_group \times 2 + i]$;
19         $frag_A[i] \leftarrow \text{Load}(\mathbf{x}[x\_idx])$ ;                      ▷ Load dense vector to A fragment
20     end
21     $acc \leftarrow \texttt{mma.sync.aligned.m16n8k8.f16.f16}(frag_A, frag_B, acc)$ ;   ▷ Tensor Core PTX-level MMA
22     $sum0 \leftarrow sum0 + acc[0]$; $sum1 \leftarrow sum1 + acc[1]$;
23 end
24 if $groupIdx = 0$ then
25     $y\_idx \leftarrow chunkIdx \times fragN + threadId\_in\_group \times 2$;
26     Store($\mathbf{y}[y\_idx], sum0$); Store($\mathbf{y}[y\_idx + 1], sum1$) ;      ▷ Store results
27 end
```

blocks into the right-hand side $frag_B$, using a transposed layout. This transposition (Line 12) modifies register position computation without incurring memory access overhead.

Both our register-level unfolding and implicit transposition necessitate manual mapping of data to specific register fragments. High-level WMMA APIs abstract away the register layout and cannot support the fine-grained, non-aligned manipulation required to feed sparse data directly into TCUs. Therefore, we employ the PTX instruction (Algorithm 1, Line 21) to achieve explicit register-level control.

4.2.2 Sparse encoding for low density. *Sparse encoding* (SE) does not preserve the relative positions of nonzeros in the original matrix. Instead, it arranges the nonzeros sequentially to fit the shape required by the TCU in a manner similar to the CSR. Figure 5(c) illustrates that the *sparse encoding* format comprises three arrays: 1) `Val` stores non-zero and zero-padded values in a row-compressed manner; 2) `Col` contains corresponding column indices; 3) `BlockPtr` records the starting index of each chunk in the SE-Val array. In the final step of the compute density extraction, the row permutation step ensures that the number of nonzeros in the hot region decreases monotonically across rows, which significantly reduces the number of zero padding required between adjacent rows.

4.2.3 Memory footprint analysis. Regarding the *sparse encoding* and *dense unfolding* format memory efficiency, our *sparse encoding* format reduces overhead by replacing CSR's rowPtr with the more compact `BlockPtr`. Additionally, our dual-mode switching selects the smaller *dense unfolding* format (using bitmaps for dense blocks) under specific conditions, further minimizing the overall memory footprint. This collectively ensures that the hot region's intermediate memory overhead does not exceed its original CSR/CSC representation, while the cold region retains its standard CSR/CSC format.

4.2.4 Switching dual modes. To accommodate varying density levels in TCU-aware compressed hot regions, we dynamically switch between two modes based on storage efficiency.

1) Problem setup. For an $H \times W$ matrix with sparsity s, we have $C = \frac{H}{\text{tcH}}$ chunks containing TC blocks of size $\text{tcH} \times \text{tcW}$, $N_{\text{nz}} = (1-s)HW$ nonzeros, average empty column proportion E, and B_{val}, B_{idx} bytes per value and index.

2) Memory storage cost analysis. The *dense unfolding* format utilizes dense-bitmap storage per TC block, with each chunk needing a pointer (B_{idx} bytes) and each TC block requiring a TC bitmap ($\frac{\text{tcH}\cdot\text{tcW}}{8}$ bytes), TC column indices ($\text{tcW} \cdot B_{\text{idx}}$ bytes), and a TC block pointer (B_{idx} bytes). Nonzeros take $N_{\text{nz}} \cdot B_{\text{val}}$ bytes. The TC block count per chunk is $\frac{(1-E)W}{\text{tcW}}$, resulting in the following storage cost:

$$\text{Cost}_{\text{DU}} = CB_{\text{idx}} + C \cdot \frac{(1-E)W}{\text{tcW}} \left(\frac{\text{tcH} \cdot \text{tcW}}{8} + \text{tcW} \cdot B_{\text{idx}} + B_{\text{idx}} \right) + N_{\text{nz}} B_{\text{val}}$$

The *sparse encoding* format exclusively stores nonzeros, with each chunk needing an index (B_{idx} bytes) and each nonzero requiring $B_{\text{val}} + B_{\text{idx}}$ bytes, yielding Equations without considering zero-padding:

$$\text{Cost}_{\text{SE}} = CB_{\text{idx}} + N_{\text{nz}}(B_{\text{val}} + B_{\text{idx}})$$

3) Switching criterion. We select *dense unfolding* compressed format when $\text{Cost}_{\text{DU}} < \text{Cost}_{\text{SE}}$:

$$C \cdot \frac{(1-E)W}{\text{tcW}} \cdot \left(\frac{\text{tcH} \cdot \text{tcW}}{8} + \text{tcW} \cdot B_{\text{idx}} + B_{\text{idx}} \right) < N_{\text{nz}} B_{\text{idx}}$$

Substituting $N_{\text{nz}} = (1-s)HW$ and $C = \frac{H}{\text{tcH}}$ establishes a sparsity threshold: *dense unfolding* format is optimal for denser regions (lower s), while *sparse encoding* format excels for sparser regions. In practice, we evaluate hot region sparsity s to determine the optimal mode for efficient memory footprint.

4.3 Coalesced and Load-Balanced CUDA Kernel

TCU-optimized hot regions exhibit compact dimensions that enhance input vector locality, while cold regions show uniform, full-range access patterns that limit locality exploitation. Register-level `MMA` operations offer theoretical efficiency and are well-suited for high-locality hot regions. However, their coupled multiplication-accumulation phases prevent intermediate result caching, which hinders effective processing of cold regions. For cold regions, we employ CUDA core-based scalar approaches that decouple these phases, storing intermediate multiplication results in shared memory to improve cache utilization and reduce register pressure.

4.3.1 Memory-access efficiency. Shared memory is a high-bandwidth yet limited storage resource on each SM [58]. When used effectively, multiple thread blocks can be scheduled concurrently on an SM, increasing warp occupancy and hiding memory access latency. In our approach, each thread block is allocated fixed, contiguous segments of seg_nnz nonzeros, with shared memory used to store their intermediate multiplication results. This approach ensures linear, aligned mapping of threads to data. As shown in Algorithm 2 (Lines 5–7), thread $threadIdx.x$ accesses the global memory address gid and stores the result at shared memory index sid. Since both gid and

Algorithm 2: Coalesced kernel in balanced cold regions

```
Input  : x[], $coldRowPtr$[], $coldVal$[], $coldColId$[], mul_nnz
Output : y[]
1  $seg\_nnz \leftarrow blockSize \times$ mul_nnz;
2  __shared__ SMEM[$seg\_nnz$];
3  $seg\_nnz\_start \leftarrow seg\_nnz \times blockIdx.x$;
4  for $i \leftarrow 0$ to mul_nnz do
5      $sid = threadIdx.x + i \times blockSize$;
6      $gid = seg\_nnz\_start + sid$;
7      SMEM[$sid$] = $coldVal[gid] \times x[coldColId[gid]]$  ▷ Coalesced GMEM load, conflict-free SMEM store
8  end
9  $\_\_syncthreads()$;
10 $accuRowInThread(coldRowPtr,$ SMEM, y$)$ ;  ▷ Row accumulation in load-balanced cold regions
```

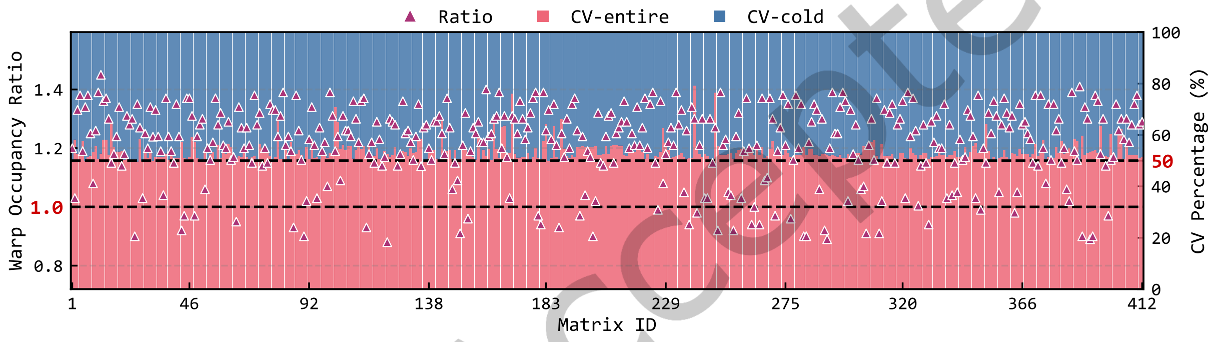


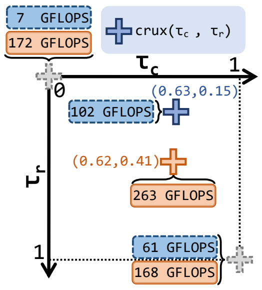
Fig. 6. Warp occupancy ratio (cold region vs. entire matrix, shown as points) and row length coefficient of variation (CV) stacked percentage distribution (shown as bars).

sid are calculated as a linear function of *threadIdx.x* with a stride of 1 (i.e., $sid = threadIdx.x + i \times blockSize$), consecutive threads in a warp access contiguous memory. This layout guarantees: 1) Coalesced global memory loads. Multiple memory requests from a warp are combined into a single transaction because the addresses are physically adjacent. 2) Conflict-free shared memory accesses. For FP16, vectorized mapping (e.g., half2) packs elements into 32-bit units aligned with hardware bank width, ensuring each thread targets a distinct bank. For FP64, unit-stride indexing naturally maps consecutive threads to independent banks without swizzling or padding. After multiplication, threads within each block collaboratively accumulate the intermediate results stored on-chip.

4.3.2 Load balancing. Row length variations cause load imbalance in SpMV due to intra-row accumulation dependencies. After Compute Density Extraction, cold regions exhibit more uniform non-zero distributions than the original matrix, resulting in a lower coefficient of variation for non-zero counts per row. This enhanced balance of row lengths directly translates into a significant improvement in *Warp Occupancy* for row accumulation (Line 10 in Algorithm 2), as demonstrated in Figure 6.

5 Sparsity-Aware Predictor

The sparsity-aware predictor (SAP) constructs an infinite two-dimensional (2D) search space for adaptive TCU selection. To efficiently navigate this search space, we design a Bayesian optimization-based module for generating



feature\matrix	loc-Gowalla	3dtube
pRatio_{row}	14730	236.4
Top10_{row}	190.007	33.261
Top50_{row}	78.920	1.143
Top100_{row}	50.993	1.143
Hot10_{row}	0.001	0.080
Hot20_{row}	0.008	0.168
Hot30_{row}	0.019	0.255
Hot40_{row}	0.039	0.343
crux^{BO}	(0.63,<u>0.15</u>)	(0.62,<u>0.41</u>)

Fig. 7. Crux diversity and high correlation with features.

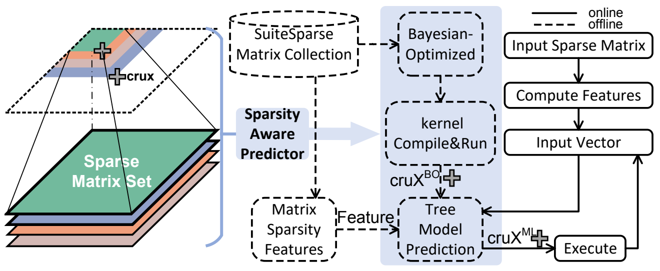


Fig. 8. Sparsity-aware prediction framework overview.

optimal crux^{BO} (Bayesian-optimized hot-cold threshold points). Using feature engineering and crux^{BO} values, we train a tree-based model for runtime adaptive tuning.

5.1 2D Search Space Construction

Integrating TCU acceleration into SpMV auto-tuning frameworks is challenging. The rigid dense-matrix computation mode of TCUs results in limited available formats and kernels, constraining the optimization space and performance generalization. Traditional ML-based SpMV methods [44, 62, 65, 75] select implementations over fixed computation regions; in contrast, we ***dynamically adjust TCU computation regions while keeping the implementation method fixed***. We recast the search space as an infinite 2D space defined by the crux (τ_c, τ_r). For each sparse matrix, we identify a near-optimal crux that maximizes SpMV performance, tailoring the computation region to the matrix's unique sparsity pattern.

5.2 Bayesian-Optimized Crux Generation

We define a performance function $f(\tau_c, \tau_r)$ representing the execution time of SpMV under a given parameter pair (line 3 in Algorithm 3). Directly evaluating f is computationally expensive, requiring kernel compilation and execution per sample, making exhaustive grid search infeasible. Moreover, f lacks analytical expression and derivative information, ruling out gradient descent methods.

We adopt Bayesian optimization, a probabilistic approach for locating optima with minimal evaluations. Bayesian optimization constructs a surrogate model to approximate f and iteratively selects sampling points based on prior evaluations. For each sparse matrix, this yields a crux^{BO}, which serves as labeled data for training ML models (lines 4-5 in Algorithm 3). The crux^{BO} exhibits both diversity and learnability, supporting the effectiveness of feature-based SAP.

Diversity. Sparse matrices have diverse non-zero distributions. For matrices with highly skewed distributions, our approach extracts dense blocks, with optimal block shape depending on the degree of imbalance. Figure 7 demonstrates different optimal crux configurations across two matrices. Conversely, uniformly distributed matrices resist dense block extraction, as row/column permutation fails to alter the non-zero distribution. For small matrices, exclusive reliance on Tensor Cores or CUDA cores may be optimal.

Learnability. The strong correlation between crux^{BO} and global matrix features enables learning-based prediction. Figure 7 shows that for loc-Gowalla and 3dtube with similar optimal τ_c values (0.63 and 0.62), optimal τ_r exhibits strong correlation with metrics such as $\text{pRatio}_{\text{row}}$, TopK_{row} and HotR_{row}, whose definitions and formulas are detailed in Table 2. According to the table on the right side of Figure 7, loc-Gowalla exhibits a significantly more skewed row distribution than 3dtube, manifested by a 62× higher $\text{pRatio}_{\text{row}}$ (14,730 vs. 236.4), higher TopK_{row}, and lower HotR_{row}. Consequently, the optimal τ_r for loc-Gowalla is lower than that of 3dtube

Algorithm 3: Sparsity-Aware Parameter Prediction

```
Input  : Sparse matrix collection $\mathscr{M}$, feature extractor $\phi(\cdot)$
Output : Sparsity-aware predictor $\mathscr{P} : \mathbb{R}^d \rightarrow [0,1]^2$
▷ Phase I: Generate crux$^{\mathrm{BO}}$ via Bayesian optimization
1  $\mathscr{D} \leftarrow \emptyset$;
2  for each matrix $M \in \mathscr{M}$ do
3      $f_M(\tau_c, \tau_r) \leftarrow$ SpMV execution time on $M$;
4      crux$^{\mathrm{BO}} \leftarrow$ BayesianOptimization$(f_M, \mathcal{S}_{2D})$ ;      ▷ 2D search space navigating
5      $\mathscr{D} \leftarrow \mathscr{D} \cup \{(\phi(M), \mathrm{crux}^{\mathrm{BO}})\}$ ;      ▷ Feature-crux$^{\mathrm{BO}}$ pair collection
6  end
▷ Phase II: Train tree-based sparsity-aware predictor
7  $\mathscr{D}_{train}, \mathscr{D}_{test} \leftarrow$ StratifiedSplit$(\mathscr{D}$, nnz_quantiles$)$;
8  $\mathscr{T} \leftarrow$ {RandomForest, XGBoost, GBDT, AdaBoost};
9  $\mathscr{P}^* \leftarrow \arg\min_{\mathscr{P} \in \mathscr{T}}$ CrossValidate$(\mathscr{P}, \mathscr{D}_{train})$;
10 $\mathscr{P}^*.train(\mathscr{D}_{train})$ ;      ▷ Train the best model
▷ Phase III: Runtime adaptive tuning
11 for unseen matrix $M_{new}$ do
12     crux$^{\mathrm{ML}} \leftarrow \mathscr{P}^*(\phi(M_{new}))$ ;      ▷ Predict optimal crux$^{\mathrm{ML}}$
13     Dynamically adjust TCU computation region using crux$^{\mathrm{ML}}$;
14 end
```

(0.15 vs. 0.41). This suggests that matrices with more skewed row distributions require fewer rows to encompass a larger share of nonzeros.

5.3 Tree Model-based Crux Prediction

Algorithm 3 formalizes the sparsity-aware parameter prediction framework, which employs Bayesian optimization to generate training data with crux$^{\mathrm{BO}}$ (lines 4-5), then selects the best-performing model from four tree-based ensemble models (lines 8-9) to enable runtime crux$^{\mathrm{ML}}$ (ML-predicted crux) prediction for unseen sparse matrices (line 12). The model training, validation, and effectiveness evaluation are detailed in Section 6.3. Figure 8 illustrates the Sparsity-Aware Predictor and its integration within the overall SpMV framework. The predictor is trained offline. The dataset comprises 2,059 sparse matrices, each characterized by engineered features that reflect sparsity patterns, as shown in Table 2. This approach ensures sparsity-driven tuning of SpMV, leveraging the complementary strengths of TCUs and CUDA cores across diverse sparsity patterns.

6 Evaluation

6.1 Experimental Setup

Platforms. Our experimental platform consists of an NVIDIA A100 80GB PCIe GPU (Ampere) and a GeForce RTX4070 Ti SUPER GPU (Ada Lovelace). 1) A100 has 432 Tensor Core units and 6912 CUDA cores, 80 GB graphics memory. 2) RTX4070 has 264 Tensor Core units and 8448 CUDA cores, 16 GB graphics memory.

Baselines. We compare the performance of MatXtract[2] with the following state-of-the-art SpMV kernels on TCUs and CUDA cores: cuSPARSE [60] v12.8 in FP16 and FP64, DASP [48] in FP16 and FP64, CSR5 [45] in FP64, and Merge-SpMV [51] in FP64. All of these approaches are tested on their available open-source code. Note that we only evaluate the FP16 data type in the RTX4070 platform, as it lacks native FP64 TCU support [57].

[2]https://github.com/luuhwy/MatXtract.git

Table 2. Sparse Matrix Features

Category	Feature	Description	Formula
Matrix Dimensions	nrows	Number of rows	n_{rows}
	ncols	Number of columns	n_{cols}
	nnz	Total non-zero elements	nnz
Sparsity Statistics	$\mu_{\text{row}}, \mu_{\text{col}}$	Mean nonzeros per row/column	$\mu_{\text{row}} = \frac{nnz}{n_{\text{rows}}}$
	$\sigma_{\text{row}}, \sigma_{\text{col}}$	Standard deviation of nonzeros per row/column	$\sigma_{\text{row}} = \sqrt{\frac{1}{n_{\text{rows}}} \sum_{i=1}^{n_{\text{rows}}} (nnz_i - \mu_{\text{row}})^2}$
	$\texttt{min}_{\text{row}}, \texttt{min}_{\text{col}}$	Minimum nonzeros per row/column	$\min(\{nnz_i\})$
	$\texttt{max}_{\text{row}}, \texttt{max}_{\text{col}}$	Maximum nonzeros per row/column	$\max(\{nnz_i\})$
	$\texttt{pRatio}_{\text{row}}, \texttt{pRatio}_{\text{col}}$	Maximum-to-minimum ratio	$\frac{\max(\text{nnz}_{\text{row/col}})}{\min(\text{nnz}_{\text{row/col}})}$
Density Skew	$\texttt{TopK}_{\text{row}}, \texttt{TopK}_{\text{col}}$	Ratio of k-th largest non-zero count to mean, where $k \in \{1, 2, 3, 4, 5, 10, 50, 100, 200, 300\}$	$\frac{nnz_{[k]}}{\mu_{\text{row/col}}}$
	$\texttt{HotR}_{\text{row}}, \texttt{HotR}_{\text{col}}$	Proportion of rows/columns covering $R\%$ nonzeros, where $R \in \{10, 20, \ldots, 90\}$	$\frac{\lvert\{i \mid \sum_{j=1}^{i} nnz_{[j]} \geq R\% \cdot nnz\}\rvert}{n_{\text{rows/cols}}}$

[1] nnz_i denotes the number of nonzeros in the i-th row/column, and $nnz_{[k]}$ denotes the k-th largest non-zero count.
[2] TopK: Heavy-tailed distribution indicators. Higher values indicate heavy-tailed distributions.
[3] HotR: Spatial concentration metrics. Lower values imply denser concentration (e.g., $\texttt{Hot10}_{\text{row}}$=0.01 means 1% rows contain 10% nonzeros).

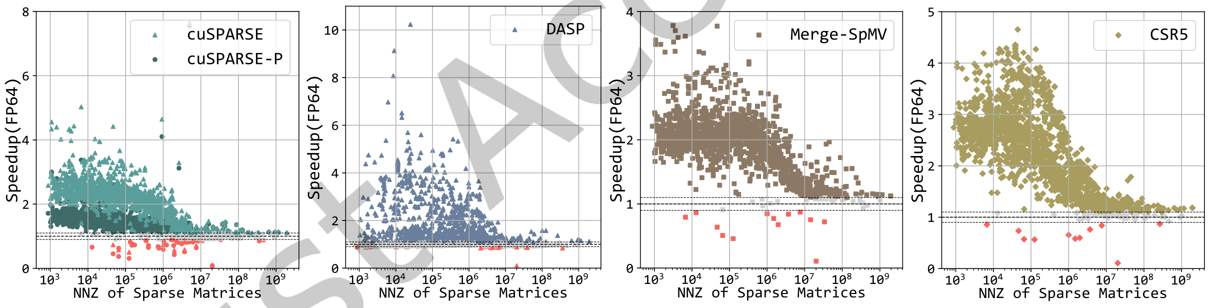


Fig. 9. Performance speedups of MatXtract (with Bayesian-optimized crux$^{\text{BO}}$) over vendor-tuned cuSPARSE and state-of-the-art approaches for FP64 on A100, where X-axes are matrix non-zero count and Y-axes are speedups. Points within the non-significant acceleration range are shown in gray.

Datasets. We use 2,059 sparse matrices from the SuiteSparse Matrix Collection [18], covering matrices with 1,000 to 2,000,000,000 nonzeros.

Overheads. Bayesian-optimized crux generation and tree-based model training are performed offline, once per hardware platform, and are excluded from the execution time. The inference of the tree model prediction (Line 12 in Algorithm 3), matrix reorganization, and compression overhead are performed once for each new sparse matrix, which we refer to as online preprocessing overhead. Discussion of both types of overhead is presented in Section 6.5.

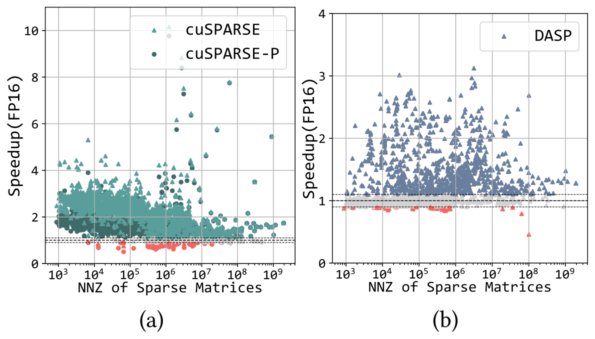


Fig. 10. Performance speedups for FP16 on A100.

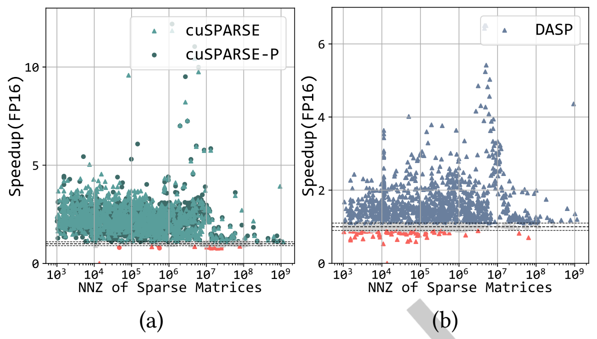


Fig. 11. Performance speedups for FP16 on RTX4070.

Table 3. Model Evaluation Results

Cross-Validation	**Final Test (RandomForest)**	
Model (MSE)	τ_c	τ_r
RandomForest (0.00755)	MSE: 0.00729	MSE: 0.00323
XGBoost (0.00873)	MAE: 0.02068	MAE: 0.01039
AdaBoost (0.01466)	R^2: 0.56232	R^2: 0.37203
GBDT (0.00813)	**Avg:** MSE = 0.00526, MAE = 0.01553	

6.2 Bayesian Optimization-based SpMV Performance

MatXtract is evaluated across the entire dataset using crux^{BO} identified through Bayesian optimization. Figures 9 and 10 illustrate the speedup of MatXtract compared to other methods on the A100 GPU, with ratios within ±10% of 1 shaded in gray to denote negligible differences. For FP64 computations, MatXtract achieves up to 7.64× (average 2.06×) over cuSPARSE, 7.55× (average 1.38×) over cuSPARSE with preprocessing (cuSPARSE-P), 10.23× (average 1.54×) over DASP, 3.78× (average 1.93×) over Merge-SpMV, and 4.65× (average 2.26×) over CSR5. For FP16, MatXtract outperforms cuSPARSE by up to 10.15× (2.24× average), cuSPARSE-P by up to 9.76× (1.6× average), and DASP by up to 3.12× (1.26× average).

On RTX4070 with Ada Lovelace architecture (Figure 11), the speedups reach up to 11.85× (2.21× average) over cuSPARSE, 12.1× (2.09× average) over cuSPARSE-P, and 6.5× (1.44× average) over DASP. The consistent performance gains across both Ampere (A100) and Ada Lovelace (RTX 4070) platforms demonstrate that MatXtract adapts robustly to diverse architectural characteristics, including varying Tensor Core capabilities, memory hierarchies, and register file organizations. MatXtract adapts the crux to generalize across a wide range of matrix sizes. For some small matrices that fully fit in the L2 cache, the crux may determine that exclusive TCU reliance is optimal, yielding greater performance benefits from our TCU-optimized cascaded compression.

Note that the total computational time equals the sum of Tensor Core and CUDA core execution times. While theoretically, full exploitation of parallelism between these units could yield more substantial speedups, our experimental platform shows limited parallel efficiency. This occurs because Tensor Cores require substantial register resources to store intermediate matrix operation results, while CUDA cores rely on the register file (RF) for thread-level data buffering. Concurrent RF access requests from both units create a register bandwidth bottleneck within the SM, limiting parallelism between CUDA cores and Tensor Cores.

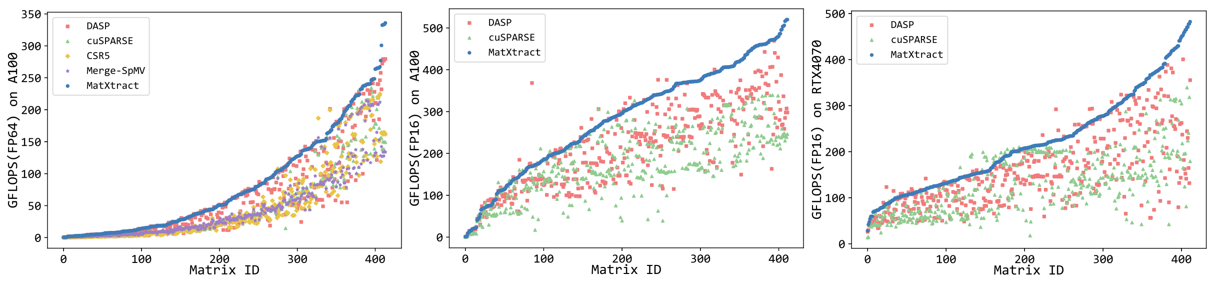


Fig. 12. Performance comparison on the test set. The X-axis is a test set of 412 sparse matrices based on 80:20 stratified sampling of nonzeros, and its Matrix ID is sorted from small to large according to MatXtract GFLOPS.

6.3 Sparsity-Aware Predictor Performance

Stratified sampling based on non-zero counts is applied to categorize matrices into five quantile-based strata with an 80%-20% train-test split. We evaluate Random Forest, XGBoost, AdaBoost, and GBDT using 5-fold cross-validation on training data, comparing mean squared error (MSE), mean absolute error (MAE), and R^2 score. As shown in Table 3, Random Forest achieves the best performance with an average cross-validation MSE of 0.00755.

On the test set, Random Forest exhibited low error metrics, with an average MSE of 0.00526 and an average MAE of 0.01553. Specifically, the R^2 value for τ_c exceeds that of τ_r, indicating a stronger explanatory power for predicting τ_c. The observed difference in prediction accuracy between τ_c and τ_r aligns well with our design strategy. During the parameter optimization phase (Bayesian optimization), τ_c is determined first, serving as the primary parameter influencing SpMV performance. Once τ_c is fixed, τ_r is subsequently optimized based on the previously established τ_c. This sequential optimization inherently introduces conditional dependence and greater variability into the determination of τ_r.

Using ML-predicted parameters (cruxes^{ML}), MatXtract achieves substantial speedups on the test set (Figure 12). On A100 with FP64, speedups reach up to 4.43× (2.16× average) over cuSPARSE, 8.07× (1.61× average) over DASP, 3.59× (2.04× average) over Merge-SpMV, and 4.35× (2.39× average) over CSR5. For FP16, speedups are up to 8.83× (1.81× average) over cuSPARSE and 2.97× (1.39× average) over DASP. On RTX4070 with FP16, speedups reach up to 9.75× (1.87× average) over cuSPARSE and 6.49× (1.56× average) over DASP. MatXtract exhibits robust performance generalization across diverse sparse matrices, outperforming cuSPARSE on 96.64% of the test cases on average across both FP16 and FP64 precision. To demonstrate MatXtract's performance advantage on large matrices, Figure 13 presents the results for 32 sparse matrices commonly used in previous SpMV optimization work [45, 53, 73] on the A100 platform. The results show that MatXtract consistently outperforms other methods across nearly all test cases.

6.4 Performance Analysis

Memory bandwidth utilization is the primary metric for evaluating hardware efficiency for SpMV. Table 4 presents the memory bandwidth utilization comparison. Across the 32 sparse matrices shown in Figure 13, MatXtract achieves average bandwidth utilizations of 69.4% and 58.4% for FP64 and FP16, respectively, outperforming cuSPARSE's 62.7% and 42.1%. Although FP16 mode delivers higher GFLOPS, its bandwidth utilization is consistently lower than FP64. This disparity stems from the diminished proportion of effective payload: in CSR-based SpMV, while value sizes shrink under FP16, column indices remain unchanged, diminishing the proportion of useful floating-point data per memory transaction and increasing the relative overhead of index metadata.

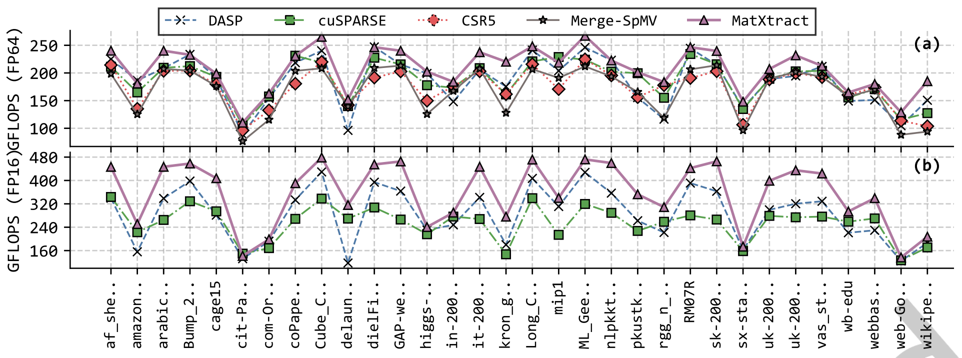
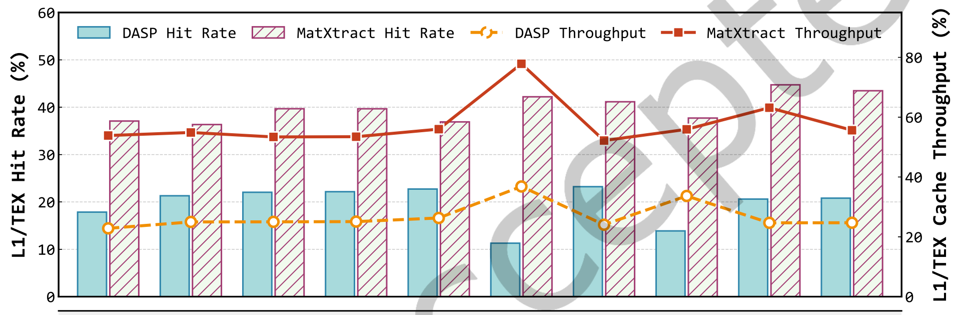


Fig. 13. SpMV Performance on 32 representative matrices.



Matrix	uk-2005	it-2004	sk-2005	GAP-web	arabic-2005	cage15	ML_Geer	coPapersDBL	Bump_2911	af_shell10
τ_c	0.66	0.72	0.56	0.65	0.57	0.40	0.66	0.85	0.00	0.00
τ_r	0.41	0.51	0.45	0.42	0.49	0.30	0.41	0.70	0.00	0.00

Fig. 14. Memory access behavior (measured by NVIDIA Nsight Compute) with cruxes.

Additionally, matrices such as cit-Patents and web-Google exhibit notably lower utilization, attributable to their highly irregular sparsity patterns that induce severe load imbalance and poor spatial/temporal cache locality.

Figure 14 details the cruxes^{ML} values and memory access characteristics of the matrices in Figure 13. MatXtract exhibits superior cache throughput and hit rates compared to the TCU-only DASP method. This is attributable to improved cache utilization for hot regions when accessing the input vector $\mathbf{x}$ and coalesced memory access for cold regions when accessing the sparse matrix $\mathbf{A}$.

MatXtract's speedup stems not only from improved cache locality through algorithmic CDE transformations, but also from TCU's superior computational throughput for dense operations. The hot regions identified by Bayesian optimization are typically small (area$<$ 5%), dense (nnz$>$ 60%), making them ideal candidates for TCU acceleration. While applying permutation directly to the CUDA core improves hot region cache locality, it sacrifices the opportunity to leverage TCU computational throughput (Figure 15 Stage I). Our performance breakdown (Figure 15 Stage I $\longrightarrow$ II) confirms that TCUs (SE-TC) can leverage their raw computational power and outperform CUDA cores (Algorithm 2). This further demonstrates that ***MatXtract effectively identifies and extracts regions well-suited for TCU acceleration to enhance overall SpMV performance***.

Table 4. Bandwidth utilization on A100. BW denotes the effective bandwidth, and Util. represents the ratio of effective bandwidth to the peak theoretical bandwidth of the hardware.

Matrix	FP64			FP16		
	Our BW (GB/s)	Our Util. (%)	cuSPARSE Util. (%)	Our BW (GB/s)	Our Util. (%)	cuSPARSE Util. (%)
af_shell0	1510.3	**78.1**	68.6	1069.8	**71.9**	55.3
amazon-2008	1343.3	**69.4**	63.2	795.5	**46.3**	41.1
arabic-2005	1526.4	**78.9**	68.7	832.4	**72.6**	43.0
Bump_2911	1449.4	**74.9**	68.3	1016.4	**73.2**	52.5
cage15	1295.5	**66.9**	65.0	944.4	**67.6**	48.8
cit-Patents	912.6	**47.2**	45.0	585.1	**28.7**	30.2
com-Orkut	998.1	**51.6**	49.9	514.3	**31.3**	26.6
coPapersDBLP	1424.9	**73.6**	73.8	825.0	**52.9**	42.6
Cube_Coup_dt0	1601.7	**82.8**	68.3	1014.4	**74.2**	52.4
delaunay_n24	1163.8	**60.1**	55.5	988.0	**59.9**	51.1
dielFilterV3real	1509.4	**78.0**	72.0	937.8	**71.7**	48.5
GAP-web	1503.9	**77.7**	69.8	826.7	**74.6**	42.7
higgs-twitter	1276.4	**66.0**	58.0	674.3	**38.8**	34.8
in-2004	1254.8	**64.8**	61.1	919.7	**49.9**	47.5
it-2004	1510.3	**78.1**	68.6	842.5	**72.6**	43.5
kron_g500-logn20	1347.7	**69.6**	51.0	448.8	**43.5**	23.2
Long_Coup_dt6	1484.8	**76.7**	68.4	1009.7	**72.5**	52.2
mip1	1325.2	**68.5**	71.7	647.2	**53.2**	33.4
ML_Geer	1639.3	**84.7**	71.0	976.0	**74.5**	50.4
nlpkkt240	1391.7	**71.9**	65.4	894.1	**73.4**	46.2
pkustk12	1233.2	**63.7**	63.2	693.4	**55.6**	35.8
rgg_n_2_23_s0	1222.6	**63.2**	53.4	844.2	**52.1**	43.6
RM07R	1504.1	**77.7**	73.9	853.0	**69.5**	44.1
sk-2005	1500.9	**77.6**	69.8	826.2	**74.7**	42.7
sx-stackover	995.0	**51.4**	46.8	520.0	**29.5**	26.9
uk-2002	1371.7	**70.9**	64.8	905.8	**67.0**	46.8
uk-2005	1488.4	**76.9**	67.3	868.0	**71.2**	44.9
vas_stokes_4M	1347.3	**69.6**	66.7	865.8	**68.6**	44.7
wb-edu	1271.7	**65.7**	62.1	955.6	**56.0**	49.4
webbase-2001	1289.1	**66.6**	63.1	935.6	**60.8**	48.3
web-Google	1004.5	**51.9**	46.6	471.6	**26.2**	24.4
wikipedia-20070206	1257.3	**65.0**	44.7	566.6	**35.6**	29.3

6.5 Overhead Analysis

MatXtract incurs both offline and online computational overhead. The offline overhead is executed once per hardware platform across the entire training set of sparse matrices. The online overhead, also known as sparse matrix preprocessing, is performed once for each individual matrix. This overhead is easily amortized by the cumulative benefits of subsequent iterations in iterative solvers such as Krylov subspace methods.

In the offline phase, Bayesian optimization efficiently converges to optimal crux^{BO} parameters within 12 SpMV kernel execution times on 32 commonly used sparse matrices. When considering online prediction overhead alone, MatXtract achieves superior end-to-end execution time compared to the Merge-SpMV baseline (which requires no preprocessing) after an average of ~5 SpMV iterations. This demonstrates that our proposed Sparsity-Aware Predictor is both lightweight and practical, effectively balancing prediction efficiency with overhead across diverse sparsity patterns. When other online overhead (matrix reorganization and compression costs) are included, the break-even point extends to 15-48 iterations, depending on specific matrix characteristics. This overhead is

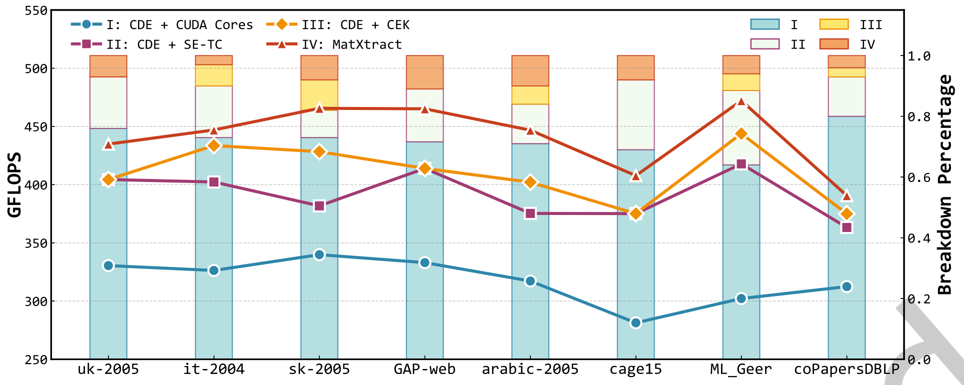


Fig. 15. Performance breakdown of MatXtract.

worthwhile since SpMV operations in Krylov subspace methods typically iterate hundreds of times, allowing the costs to be effectively amortized.

Uniformly distributed matrices—akin to the Eight Queens problem—resist dense block extraction, as row-column reordering cannot alter their nonzero distribution. For some matrices with small sizes, exclusive reliance on either Tensor Cores or CUDA cores often yields optimal performance. Consequently, preprocessing overhead for sparse matrices with small matrix sizes and very uniform non-zero distributions (e.g., Bump_2911 and af_shell10 in Figure 14) becomes very lightweight, requiring only prediction cost without matrix reorganization. The worst-case scenario for preprocessing will occur oppositely, because crux typically guides a complete preprocessing process that includes prediction, reorganization, and compression.

7 Related Work

Matrix reordering for cache locality optimization. Many heuristic reordering algorithms have been proposed to improve the cache locality of SpMV. These algorithms can be categorized into four primary groups [67]: 1) bandwidth and profile reducing orderings, such as Cuthill-McKee (CM) [13] and Reverse Cuthill-McKee (RCM) [46], which cluster nonzeros near the diagonal to improve data locality; 2) fill-in reducing orderings, including Approximate Minimum Degree (AMD) [2] and Nested Dissection (ND) [39], primarily designed to minimize fill-in during sparse factorization; 3) partitioning-based orderings utilizing graph partitioning (GP) [39] or hypergraph partitioning (HP) [81] to minimize edge-cut and communication; and 4) microarchitecture-optimized approaches like Gray ordering [79], which improves branch prediction. While effective in specific scenarios, these heuristic methods are inherently *static best-effort* strategies. According to Trotter et al.'s comprehensive evaluation [67], we can see that these static strategies lack adaptability to hardware characteristics and sparse matrices, and often introduce substantial reordering overhead. Our compute density extraction differs from traditional reordering in two key aspects. ***First***, it is co-designed with Bayesian Optimization for identifying effective reordering termination coordinates and two-stage cascaded compression for handling hierarchical sparsity, enabling robust generalization across platforms and matrices. ***Second***, it targets globally dense regions to benefit both TCUs and CUDA cores while maintaining low preprocessing overhead, thus fully utilizing heterogeneous hardware with minimal amortization requirement.

Manual SpMV on CUDA cores. Traditional SpMV optimization on CUDA cores primarily focuses on designing hand-tuned storage formats and computational kernels. These formats often reflect the design strategy of the current method. Building upon CSR, Coordinate (COO), and ELLPACK (ELL), many advanced formats have been proposed to improve workload balance [11, 33] and memory access locality [1, 23]. Notable derivatives of CSR include PCSR [36], CSX [40], ACSR [4], CSR-Scalar [30], CSR-Vector [22], CSR-Stream [32], CSR-Adaptive [14], CSR5 [45], Merge-SpMV [51], and LightSpMV [47]. COO-based advancements encompass BCCOO [74], SCOO [16], and BRO-COO [66]. Variants of the ELL format include SELL [52], SELL-C-σ [41], ESB, AdELL [50], and ELL-R [68]. Some hybrid approaches, like the Hybrid (HYB) format [5] and TileSpMV [55], leverage the strengths of different formats to enhance performance.

Manual SpMV on Tensor Cores. With the rapid progress of deep learning, specialized units designed to accelerate matrix multiplication, such as TCUs, are increasingly integrated into emerging processors, providing acceleration opportunities for many workloads, including euclidean distance calculations [12], stencil computations [35, 78], sparse matrix-matrix multiplication [26, 61], scan algorithm [15], as well as other applications [27, 34, 38, 42, 43, 49, 70]. DASP [48] and Spaden [9] target the SpMV TCU-only acceleration. They both force all sparse computations into dense accelerators. DASP groups matrix rows and designs efficient computation kernels tailored to each group. Spaden introduces a bitmap-based format and kernel to accelerate relatively dense sparse matrices.

SpMV auto-tuners on CUDA cores. To improve performance generalization, machine learning has emerged as a viable approach for selecting optimal formats [29, 75]. SMAT [44, 65] uses decision tree-based learning to automatically select optimal formats by extracting structural features from over 2,000 matrices. Other ML-based SpMV auto-tuners [7, 24, 62] leverage a variety of models, including decision trees [54, 72], multiclass support vector machines [6, 54], multilayer perceptron [54], and CNNs [80], to predict and select the most suitable method for a given sparse matrix on GPUs.

Workload partitioning in heterogeneous systems. The central challenge in heterogeneous computing is matching architectural capabilities to application characteristics to maximize performance. Shen et al. [63] formalize this placement problem by modeling workload partitioning as a multi-dimensional function over hardware configurations, kernel types, and input datasets. Beyond coarse-grained distribution, modern integrated architectures enable fine-grained collaboration. FinePar [77] transforms the input irregular kernel to use both the CPU and GPU, and builds performance models for partitioning the workload, achieving both device-level and thread-level load balance. To eliminate the overhead of offline profiling, Cho et al. [10] propose on-the-fly partitioning, shifting the placement decision to runtime. These works establish that effective processor mapping depends on a synergistic understanding of both hardware-specific capabilities and data-dependent workload characteristics [8].

8 Conclusion

This paper introduces MatXtract, a sparsity-aware transformation system that significantly enhances the efficiency of TCU-accelerated SpMV. To address the limitations of TCU-only approaches, MatXtract incorporates three components. First, Compute Density Extraction optimizes memory hierarchy utilization and generates computation substrates well-suited for Tensor-CUDA cores. Second, Cascaded Execution Kernel further leverages the *hierarchical sparsity* on CDE-identified hot regions through a two-stage cascaded compression and employs memory-coalesced CUDA kernels for uniform cold regions. Third, the Sparsity-Aware Predictor ensures consistent performance across diverse sparsity patterns on fixed dense accelerators. Comprehensive experiments on modern GPUs show that MatXtract achieves substantial speedups over Tensor and CUDA core-based SpMV implementations across most sparse matrix scenarios. MatXtract's design of partitioning sparse workloads by heterogeneous unit characteristics and enabling their independent execution provides a foundation for two promising directions. First, it can exploit enhanced parallelism between Tensor Cores and CUDA Cores in advanced hardware (e.g., Hopper's WGMMA) to further unlock the acceleration potential of tensor-scalar hybrid algorithms. Second, this design enables exploration of other heterogeneous collaborations, such as CPU-GPU systems. Leveraging

MatXtract's existing partitioning logic, highly irregular tasks can be offloaded to the CPU via asynchronous scheduling to hide communication and synchronization overhead.

References

[1] Christie Alappat, Achim Basermann, Alan R. Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. 2020. A Recursive Algebraic Coloring Technique for Hardware-efficient Symmetric Sparse Matrix-vector Multiplication. *ACM Trans. Parallel Comput.* 7, 3, Article 19 (June 2020), 37 pages. doi:10.1145/3399732

[2] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* 30, 3 (Sept. 2004), 381–388. doi:10.1145/1024074.1024081

[3] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P. Sadayappan. 2014. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 781–792. doi:10.1109/SC.2014.69

[5] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* 1–11. doi:10.1145/1654059.1654078

[6] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP).* 496–505. doi:10.1109/ICPP.2016.64

[7] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2018. BestSF: A Sparse Meta-Format for Optimizing SpMV on GPU. *ACM Trans. Archit. Code Optim.* 15, 3, Article 29 (Sept. 2018), 27 pages. doi:10.1145/3226228

[8] Marcos N. L. Carvalho, Alkis Simitsis, Anna Queralt, and Oscar Romero. 2024. Workload Placement on Heterogeneous CPU-GPU Systems. 17, 12 (Aug. 2024), 4241–4244. doi:10.14778/3685800.3685845

[9] YuAng Chen and Jeffrey Xu Yu. 2024. Bitmap-Based Sparse Matrix-Vector Multiplication with Tensor Cores. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24).* Association for Computing Machinery, New York, NY, USA, 1135–1144. doi:10.1145/3673038.3673055

[10] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. 2018. On-the-fly workload partitioning for integrated CPU/GPU architectures *(PACT '18).* Association for Computing Machinery, New York, NY, USA, Article 21, 13 pages. doi:10.1145/3243176.3243210

[11] Genshen Chu, Yuanjie He, Lingyu Dong, Zhezhao Ding, Dandan Chen, He Bai, Xuesong Wang, and Changjun Hu. 2023. Efficient Algorithm Design of Optimizing SpMV on GPU. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23).* Association for Computing Machinery, New York, NY, USA, 115–128. doi:10.1145/3588195.3593002

[12] Brian Curless and Michael Gowanlock. 2025. Fast and Scalable Mixed Precision Euclidean Distance Calculations Using GPU Tensor Cores. In *Proceedings of the 54th International Conference on Parallel Processing (ICPP '25).* Association for Computing Machinery, New York, NY, USA, 288–298. doi:10.1145/3754598.3754636

[13] Elizabeth H. Cuthill and John M. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69.* https://api.semanticscholar.org/CorpusID:18143635

[14] Mayank Daga and Joseph L Greathouse. 2015. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *2015 IEEE 22nd International conference on high performance computing (HiPC).* IEEE, 64–74.

[15] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19).* Association for Computing Machinery, New York, NY, USA, 46–57. doi:10.1145/3330345.3331057

[16] Hoang-Vu Dang and Bertil Schmidt. 2013. CUDA-enabled Sparse Matrix-Vector Multiplication on GPUs using atomic operations. *Parallel Comput.* 39, 11 (Nov. 2013), 737–750. doi:10.1016/j.parco.2013.09.005

[17] Timothy A. Davis. 2023. Algorithm 1037: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 49, 3, Article 28 (Sept. 2023), 30 pages. doi:10.1145/3577195

[18] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. doi:10.1145/2049662.2049663

[19] Yangdong (Steve) Deng, Bo David Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '09).* Association for Computing Machinery, New York, NY, USA, 539–546. doi:10.1145/1687399.1687501

[20] Jack Dongarra and Francis Sullivan. 2000. Guest Editors' Introduction: The Top 10 Algorithms. *Computing in Science and Engg.* 2, 1 (Jan. 2000), 22–23. doi:10.1109/MCISE.2000.814652

[21] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: generating high performance SpMV codes directly from sparse matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 66, 15 pages.

[22] István R eguly and Mike Giles. 2012. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *2012 Innovative Parallel Computing (InPar)*. 1–12. doi:10.1109/InPar.2012.6339602

[23] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *2017 46th International Conference on Parallel Processing (ICPP)*. 292–301. doi:10.1109/ICPP.2017.38

[24] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. BASMAT: bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 423–424. doi:10.1145/3293883.3301490

[25] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. doi:10.1145/3620666.3651378

[26] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 243–260. doi:10.1145/3689031.3717481

[27] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 37, 13 pages. doi:10.1145/3458817.3476157

[28] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages.

[29] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-matrix Multiplication. *ACM Comput. Surv.* 55, 12, Article 244 (March 2023), 36 pages. doi:10.1145/3571157

[30] Michael Garland. 2008. Sparse matrix computations on manycore GPU's. In *2008 45th ACM/IEEE Design Automation Conference*. 2–6.

[31] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. 2024. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1012–1028. doi:10.1109/HPCA57654.2024.00081

[32] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 769–780. doi:10.1109/SC.2014.68

[33] Jihu Guo, Rui Xia, Jie Liu, Xiaoxiong Zhu, and Xiang Zhang. 2024. CAMLB-SpMV: An Efficient Cache-Aware Memory Load-Balancing SpMV on CPU. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 640–649. doi:10.1145/3673038.3673042

[34] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2019. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 47, 11 pages. doi:10.1109/SC.2018.00050

[35] 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 *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 355–368. doi:10.1145/3710848.3710897

[36] Guixia He, Jiaquan Gao, et al. 2016. A novel CSR-based sparse matrix-vector multiplication on GPUs. *Mathematical Problems in Engineering* 2016 (2016).

[37] Zhengding Hu, Jingwei Sun, Zhongyang Li, and Guangzhong Sun. 2024. PckGNN: Optimizing Aggregation Operators with Packing Strategies in Graph Neural Networks. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2–13. doi:10.1109/IPDPS57955.2024.00010

[38] Zhuoran Ji and Cho-Li Wang. 2022. Efficient exact K-nearest neighbor graph construction for billion-scale datasets using GPUs with tensor cores. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. doi:10.1145/3524059.3532368

[39] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20 (1998), 359–392. https://api.semanticscholar.org/CorpusID:3628209

[40] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: an extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 247–256. doi:10.1145/1941553.1941587

[41] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (Jan. 2014), C401–C423. doi:10.1137/130930352

[42] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. 2019. BSTC: a novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 38, 30 pages. doi:10.1145/3295500.3356169

[43] 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 *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 90–102. doi:10.1109/CGO51591.2021.9370335

[44] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 117–126. doi:10.1145/2491956.2462181

[45] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[46] Wai-Hung Liu and Andrew H. Sherman. 1976. Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices. *SIAM J. Numer. Anal.* 13, 2 (April 1976), 198–213. doi:10.1137/0713020

[47] Yongchao Liu and Bertil Schmidt. 2015. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 82–89.

[48] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 73, 14 pages. doi:10.1145/3581784.3607051

[49] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 52, 16 pages. doi:10.1109/SC41406.2024.00058

[50] Marco Maggioni and Tanya Berger-Wolf. 2013. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In *2013 42nd International Conference on Parallel Processing*. 11–20. doi:10.1109/ICPP.2013.10

[51] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 678–689.

[52] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers* (Pisa, Italy) *(HiPEAC'10)*. Springer-Verlag, Berlin, Heidelberg, 111–125. doi:10.1007/978-3-642-11515-8_10

[53] Naveen Namashivayam, Sanyam Mehta, and Pen-Chung Yew. 2021. Variable-sized blocks for locality-aware SpMV. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 211–221. doi:10.1109/CGO51591.2021.9370327

[54] Israt Nisa, Charles Siegel, Aravind Sukumaran Rajam, Abhinav Vishnu, and P. Sadayappan. 2018. Effective Machine Learning Based Format Selection and Performance Modeling for SpMV on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1056–1065. doi:10.1109/IPDPSW.2018.00164

[55] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 68–78. doi:10.1109/IPDPS49936.2021.00016

[56] NVIDIA. 2017. *NVIDIA Volta GPU Architecture Whitepaper.* Technical Report. NVIDIA. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf Accessed on Oct 24, 2025.

[57] NVIDIA. 2023. *NVIDIA ADA GPU ARCHITECTURE* . Technical Report. NVIDIA. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf Accessed on Oct 24, 2025.

[58] NVIDIA. 2025. *CUDA C++ Programming Guide.* Technical Report. NVIDIA. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf Accessed on Nov 20, 2025.

[59] NVIDIA. 2025. *Parallel Thread Execution ISA Version 9.0.* Technical Report. NVIDIA. https://docs.nvidia.com/cuda/pdf/ptx_isa_9.0.pdf Accessed on Nov 20, 2025.

[60] NVIDIA Corporation. n.d.. CUDA cuSPARSE Library Documentation. https://docs.nvidia.com/cuda/cusparse/. Accessed on Mar 24, 2025.

[61] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High Performance Unstructured SpMM Computation Using Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 54, 14 pages. doi:10.1109/SC41406.2024.00060

[62] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 99–108. doi:10.1145/2751205.2751244

[63] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. 2016. Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.* 27, 9 (Sept. 2016), 2766–2780. doi:10.1109/TPDS.2015.2509972

[64] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/2442516.2442530

[65] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture. *ACM Trans. Math. Softw.* 44, 4, Article 46 (Aug. 2018), 25 pages. doi:10.1145/3218823

[66] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. 2013. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 26, 12 pages. doi:10.1145/2503210.2503234

[67] James D. Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 31, 13 pages. doi:10.1145/3581784.3607046

[68] F. Vázquez, J. J. Fernández, and E. M. Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput.: Pract. Exper.* 23, 8 (June 2011), 815–826. doi:10.1002/cpe.1658

[69] 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 *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. doi:10.1145/2851141.2851145

[70] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 107–119. doi:10.1145/3503221.3508408

[71] 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 *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 149–164. https://www.usenix.org/conference/atc23/presentation/wang-yuke

[72] Guoqing Xiao, Tao Zhou, Yuedan Chen, Yikun Hu, and Kenli Li. 2024. Machine Learning-Based Kernel Selector for SpMV Optimization in Graph Analysis. *ACM Trans. Parallel Comput.* 11, 2, Article 11 (June 2024), 25 pages. doi:10.1145/3652579

[73] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: efficient vectorization of SpMV on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 149–162. doi:10.1145/3168818

[74] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 107–118. doi:10.1145/2555243.2555255

[75] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 329–341. doi:10.1145/3572848.3577506

[76] Xin You, Changxi Liu, Hailong Yang, Pengbo Wang, Zhongzhi Luan, and Depei Qian. 2023. Vectorizing SpMV by Exploiting Dynamic Regular Patterns. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 53, 12 pages. doi:10.1145/3545008.3545042

[77] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. 2017. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 27–38. doi:10.1109/CGO.2017.7863726

[78] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 53, 17 pages. doi:10.1109/SC41406.2024.00059

[79] Haoran Zhao, Tian Xia, Chenyang Li, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2020. Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon. *2020 IEEE 38th International Conference on Computer Design (ICCD)* (2020), 601–609. https://api.semanticscholar.org/CorpusID:229375676

[80] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 94–108. doi:10.1145/3178487.3178495

[81] Ümit V. Çatalyürek and Cevdet Aykanat. 1999. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Trans. Parallel Distributed Syst.* 10 (1999), 673–693. https://api.semanticscholar.org/CorpusID:2954155

Received 23 November 2025; revised 7 January 2026; accepted 18 January 2026