

NeutronSparse: Coordinating Heterogeneous Engines for Sparse Matrix Multiplication on NPUs

XIN AI*, School of Computer Science and Engineering, Northeastern University, China

ZEYU LING*, School of Computer Science and Engineering, Northeastern University, China

HAO YUAN, School of Computer Science and Engineering, Northeastern University, China

QIANGE WANG, School of Computer Science and Engineering, Northeastern University, China

YANFENG ZHANG†, School of Computer Science and Engineering, Northeastern University, China

YUTAO PENG, School of Computer Science and Engineering, Northeastern University, China

GE YU, School of Computer Science and Engineering, Northeastern University, China

Sparse matrix–matrix multiplication (SpMM) is a fundamental data operation for large-scale sparse data processing. With NPUs increasingly deployed in data centers for their performance and energy efficiency, accelerating SpMM on these platforms is a natural choice. However, high-performance SpMM on NPUs poses a data management challenge, as irregular sparsity demands efficient data organization and scheduling. On Ascend 910B, the official MindSpore implementation achieves only 36.3% of the performance of GPU-based sparse libraries such as cuSPARSE on NVIDIA A100.

To this end, we conduct an in-depth architectural analysis of SpMM execution on NPUs versus GPU and identify that the key performance bottleneck for SpMM on NPUs lies in the lack of efficient coordination across heterogeneous compute units under tile-based execution model. Therefore, we propose NeutronSparse, a coordination-first SpMM framework for NPUs. NeutronSparse integrates two key techniques: (i) Sparsity-aware coordination of heterogeneous engines, which adaptively partitions and balances workloads between heterogeneous compute units to keep them busy, and (ii) Locality-aware tile orchestrating, which reorganizes and reuses data tiles to reduce redundant computation and data movement overhead. Evaluations on Ascend 910B show that NeutronSparse achieves $1.26\times$ – $7.78\times$ speedup over NPU baselines and $1.03\times$ – $3.07\times$ speedup over leading GPU libraries on NVIDIA A100, revealing untapped potential of NPUs for sparse computation.

CCS Concepts: • **Information systems** → **Data management systems**.

Additional Key Words and Phrases: sparse matrix multiplication, NPU, heterogeneous engine, data scheduling

ACM Reference Format:

Xin Ai, Zeyu Ling, Hao Yuan, Qiange Wang, Yanfeng Zhang, Yutao Peng, and Ge Yu. 2026. NeutronSparse: Coordinating Heterogeneous Engines for Sparse Matrix Multiplication on NPUs. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 281 (September 2026), 27 pages. <https://doi.org/10.1145/3837119>

*Both authors contributed equally to this research.

†Yanfeng Zhang is the corresponding author.

Authors' Contact Information: Xin Ai, aixin0@stumail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Zeyu Ling, lingzeyu@stumail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Hao Yuan, yuanhao@stumail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Qiange Wang, wangqq@mail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Yanfeng Zhang, zhangyf@mail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Yutao Peng, yujiayang@stumail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China; Ge Yu, yuge@mail.neu.edu.cn, School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART281

<https://doi.org/10.1145/3837119>

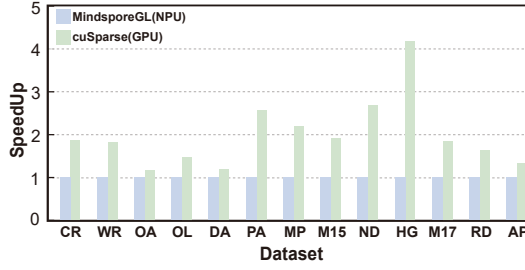


Fig. 1. Comparison of SpMM performance between the Ascend 910B and Nvidia A100.

1 Introduction

Sparse matrix–matrix multiplication (SpMM) is a fundamental kernel in modern data processing and data management systems [8, 10, 14, 16, 31, 51, 52, 57], and is widely used in AI workloads [36, 46, 50]. In particular, SpMM acts as a core data transformation that combines sparse structure with dense features, serving as the computational primitive for graph neural network aggregation [2, 3, 34, 42–44, 58] and sparse attention in large language models (LLMs) [18, 35, 41]. As input sizes grow from billion-scale graphs to LLMs with hundreds of billions of parameters [46, 50, 51], SpMM increasingly stresses both compute and memory bandwidth, making accelerator support essential for scalable performance.

Neural Processing Units (NPUs) have become a cornerstone of modern AI infrastructure, with vendors such as Huawei (Ascend) [22], Google (TPU) [17], Cambricon (MLU) [24], and Intel (Gaudi) [28] deploying their architectures widely across data centers. As demand for AI computation accelerates, NPUs’ market share continues to grow rapidly [55, 56]. These NPUs share two common architectural features. First, they are built around matrix-centric heterogeneous compute units (e.g., matrix, vector, and scalar engines) to support diverse operator and data-access patterns. Second, they follow a tile-centric execution style, where fixed-shape matrix/vector tiles are the basic unit of both computation and data staging. These architectural features have the potential for higher computation capacity [39]. In this work, we focus on the Ascend NPU, which provides mature software support through its Compute Architecture for Neural Networks (CANN).

Efficiently implementing SpMM on NPUs poses a data management challenge. Specifically, NPUs are architected around regular dataflows and high-throughput matrix engines [12, 47], which perform well on dense tensors but struggle with the skew, dynamic sparsity, and irregular memory access that characterize SpMM workloads [10, 16, 36]. As shown in Figure 1, our evaluation on the SuiteSparse dataset [6] reveals that the official SpMM implementation in MindSpore [59] for Ascend 910B is 1.2 to 4.1 times slower than cuSPARSE running on an NVIDIA A100 GPU, despite both offering comparable peak FLOPS and memory bandwidth (910B: 280 TFLOPS, 1.8 TB/s; A100: 312 TFLOPS, 2.04 TB/s). This exposes the significant optimization gap in the current Ascend sparse computation stack and highlights the potential of architecture-aware data organization and execution orchestration for improving SpMM performance on sparse workloads. Given the widespread deployment of NPUs and the central role of SpMM in data-intensive AI pipelines (e.g., GNN aggregation and sparse attention), efficient SpMM on existing NPU platforms is essential to sustain scalable data management and analytics.

Unfortunately, SpMM optimization techniques developed for GPUs cannot be directly applied to NPUs due to fundamental architectural differences. We identify two NPU-specific challenges for SpMM. (1) heterogeneous engines must be explicitly coordinated to avoid load imbalance and idle compute. (2) tile-centric execution can introduce substantial redundant work and data movement overhead under irregular sparsity.

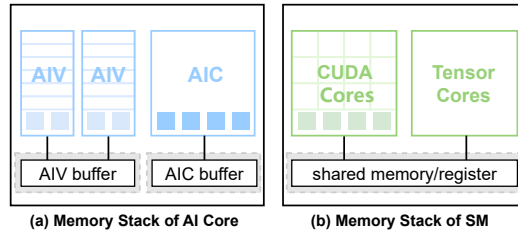


Fig. 2. Resource allocation in NPU AI Cores and GPU SMs.

First, Ascend NPUs integrate a matrix engine (AIC) and a vector engine (AIV) within each AI Core. Since they have independent instruction streams and per-engine local buffers, AIC and AIV can run concurrently with minimal on-chip resource contention (Figure 2 (a)) [7, 22, 55, 56]. This makes coordination a first-order requirement for SpMM on NPUs: workloads must be partitioned and scheduled to keep AIC and AIV making balanced progress, otherwise one engine stalls. By contrast, GPU SpMM optimizations typically alternates CUDA-core and Tensor-Core kernels with limited overlap, because both engines contend for shared SM resources (e.g., registers and shared memory) (Figure 2 (b)) [53]. As a result, GPU approaches typically statically partition sparse and dense regions and execute the corresponding kernels sequentially (e.g., sparse on CUDA cores and dense blocks on Tensor Cores) [21, 38, 46, 50]. Such mapping is fundamentally mismatched to NPUs: it routes work based on sparsity alone, ignoring heterogeneous engine throughput and execution balance. As a result, one engine dominates the critical path while the other idles, limiting effective overlap and parallelism.

Second, Ascend NPUs follow a SIMD-based execution model that combines tile-based computation with software-managed memory [22]. AIC and AIV operate on fixed-size matrix and vector tiles, so SpMM is executed at a coarse, tile-level granularity. They always process tiles of a fixed shape, no matter how nonzeros are distributed inside the tile; when the sparsity pattern is uneven, many tiles contain only a few useful entries, which leads to redundant computation and load imbalance [1, 29, 37]. GPUs instead follow a SIMT model, mapping thousands of lightweight threads to fine-grained work units (e.g., individual nonzeros or short segments) [53, 57]. SIMT-specific techniques such as warp-level NNZ load balancing (thread/warp-per-nonzero mapping) are hard to port to SIMD NPUs [25, 33, 37, 38, 53]. Therefore, SpMM on NPUs calls for tile-centric scheduling that partitions and dispatches work at tile granularity to mitigate redundant work.

These differences introduce challenges for sparse workloads but also suggest an opportunity: SpMM on NPUs must be designed around coordinating heterogeneous engines under SIMD-based, tile-centric execution, rather than simply porting GPU-style kernels. Therefore, we propose NeutronSparse, a coordination-first SpMM framework for NPUs. First, it performs sparsity-aware coordination of heterogeneous engines that goes beyond data-driven partitioning. It derives a sparsity threshold based on hardware execution characteristics, which guides a lightweight row-column extraction that assigns sparse rows and columns to AIV and the remaining denser workloads to AIC. At runtime, NeutronSparse continuously monitors the relative progress of AIV and AIC and adaptively migrates tiles to the faster engine, ensuring that both engines make balanced progress and remain busy throughout the pipeline. Second, NeutronSparse introduces locality-aware tile orchestrating that is specifically designed for SIMD-style NPUs. Rather than pursuing fine-grained NNZ-level reorderings, NeutronSparse reshapes sparsity at tile granularity through a lightweight global-local reordering that forms denser tiles. It further exploits software-visible control over the NPU memory hierarchy to enable hierarchical tile reusing, coordinating reuse across AI Cores via shared cache residency and within each AI Core via buffer-resident tile shaping.

Our contributions are summarized as follows:

- **Cross-Architecture Analysis.** We systematically compare SpMM on GPUs and Ascend NPUs, identifying two NPU-specific bottlenecks: cross-engine coordination and sparsity inefficiency under SIMD-based execution.
- **Sparsity-aware Heterogeneous Coordination.** We propose a coordination mechanism that combines architecture-aware static partitioning with runtime adaptive tile redistribution, balancing AIC/AIV workloads to maximize overlap.
- **Locality-aware Tile Orchestrating.** We propose a tile-granular data orchestrating method, combining global-local reordering with hierarchical tile reuse to reduce redundant computation and data movement overhead.
- **The NeutronSparse Framework.** We implement NeutronSparse, an SpMM framework on Ascend 910B that integrates our coordination mechanisms with optimized AIC/AIV kernels.

We evaluate NeutronSparse on an Ascend 910B processor. The experimental results show that NeutronSparse achieves $1.26\times\text{--}7.78\times$ speedup over the NPU baseline and achieves $1.03\times\text{--}3.07\times$ speedup over state-of-the-art GPU SpMM libraries across various datasets.

The rest of this paper is organized as follows. Section 2 introduces the architectures of GPUs and Ascend NPUs. Section 3 presents our cross-architecture analysis and motivates NPU-specific SpMM design. Section 5 describes sparsity-aware coordination of heterogeneous engines. Section 6 describes our locality-aware tile orchestrating. Section 4 describes the implementation of NeutronSparse. Section 8 evaluates NeutronSparse on real-world workloads. Section 9 discusses related work. Section 11 concludes the paper.

2 Background

2.1 GPU Architecture

Modern GPUs are general-purpose parallel accelerators, commonly used in high-performance computing and AI workloads. They adopt the Single Instruction Multiple Thread (SIMT) programming model, where thousands of lightweight threads execute in parallel, grouped into warps of 32 threads that run in lock-step. Figure 3 shows the overview of the GPU architecture, using an NVIDIA GPU as an example. At the hardware level, a GPU is composed of multiple Streaming Multiprocessors (SMs), each integrating scalar cores and, in some architectures, matrix-oriented cores such as NVIDIA Tensor Cores [10, 13, 31]. These compute units share the SM's fast on-chip storage, including registers and shared memory [10]. To further boost throughput, GPUs use hardware-managed L1/L2 caches and massive multi-threading to hide memory latency [13, 16, 32]. All SMs access a global memory space, enabling inter-SM communication and large-scale data processing [54].

2.2 Ascend NPU Architecture

Ascend NPUs, developed by Huawei, are AI-specialized processors that adopt a heterogeneous design to accelerate deep learning and other compute-intensive tasks [7, 22]. Figure 4 shows the overview of the Ascend NPU Architecture. The core computational component is the AI Core, built on the DaVinci architecture [22, 23], which integrates two primary compute engines: the AI Cube (AIC) and the AI Vector (AIV). AIC is optimized for dense matrix multiplications and features a 3D Cube Unit that processes $16\times 16\times 16$ float16 tiles per cycle. AIV is optimized for vector operations and includes a 2048-bit Vector Unit capable of 128-wide computations per cycle [22]. Both engines are equipped with local buffers and scalar units for control and reduction operations. Each AI Core contains two AIV units and one AIC unit, each loading its own instruction stream to enable decoupled execution of vector and matrix tasks [7, 12, 22, 56]. To facilitate data movement, Ascend provides a dedicated Memory Transfer Engine (MTE), which manages transfers between HBM and

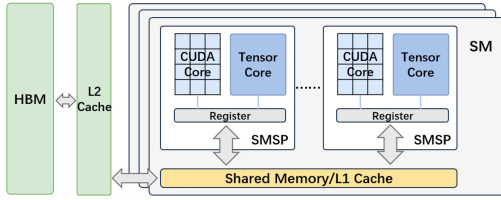


Fig. 3. Overview of the Nvidia GPU architecture.

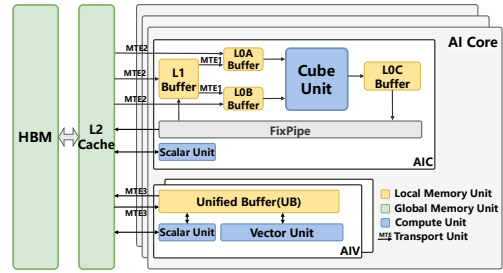


Fig. 4. Overview of Ascend NPU architecture.

various on-chip buffers [22, 56]. In addition, Ascend includes a fixed-function FixPipe datapath that streams tiles between on-chip buffers and global memory for efficient staging and write-back (e.g., draining L0C results to HBM) [22, 56].

Different generations of Ascend processors adopt slightly different AI Core organizations. Earlier Ascend architectures (Ascend 910A and early 910B) adopted a coupled design [22], where the Cube unit can directly feed the Vector unit through Buffer C and the AI Core contains a single scalar control path. In contrast, this paper targets the newer decoupled design used by current mainstream Ascend processors (910B/910C/910D), where Cube and Vector are separated into AIC and AIV, the direct Cube-to-Vector path is removed, and each side has its own scalar unit for independent instruction dispatch and control. Figure 4 follows this newer decoupled design.

3 Comparison and Analysis of GPU and NPU for SpMM

While SpMM has been extensively optimized on GPUs [8, 10, 14, 25, 37, 40, 45, 52, 57], these optimizations do not directly carry over to NPUs due to key architectural differences. We compare GPU and Ascend NPU architectures along two dimensions that fundamentally affect SpMM performance: **Compute Unit Heterogeneity** and **SIMD-based Execution**.

3.1 Compute Unit Heterogeneity

Both GPUs and Ascend NPUs employ heterogeneous compute units, but differ fundamentally in how these units interact and are scheduled. This distinction has a major impact on SpMM execution efficiency. Figure 3 and Figure 4 highlight a key scheduling contrast between GPUs and Ascend NPUs. On GPUs, CUDA cores and Tensor Cores are integrated within the same SM and contend for shared on-chip resources (e.g., registers/shared memory), which limits practical overlap in SpMM execution [13, 53]. On Ascend, AIV and AIC are decoupled with independent pipelines and local buffers, enabling concurrent execution within an AI Core and making cross-engine coordination a primary lever for SpMM efficiency.

Experimental Analysis. To quantify the benefit of concurrent execution across heterogeneous compute engines, we conduct a controlled overlap experiment on both NPUs and GPUs. We co-run two matrix multiplication tasks, assigning one to the vector-oriented engine (AIV/CUDA cores) and the other to the matrix engine (AIC/Tensor Cores), and vary the workload of the latter. We measure concurrency using the Overlap Rate metric [53], which captures the fraction of execution time saved due to overlap (upper-bounded by 50%). As shown in Fig. 5, Ascend NPUs achieve consistently higher overlap (2.1%-32.9%), while GPUs show minimal or even negative overlap (-3.9%-1.3%). This gap indicates that NPUs provide substantially stronger support for cross-engine concurrency, motivating a coordination-first design for SpMM.

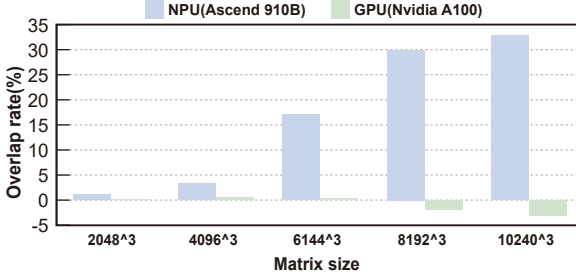


Fig. 5. Effective overlap between heterogeneous compute engines on the NPU and GPU.

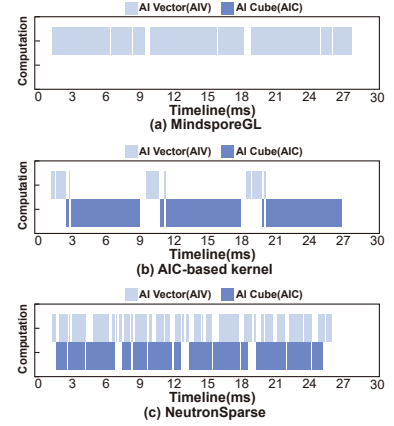


Fig. 6. AIC/AIV utilization timeline on ogbn-arxiv.

Limitation of Existing NPU Baselines. We further analyze how prior Ascend SpMM implementations utilize heterogeneous engines within a fixed execution window on ogbn-arxiv. We profile the execution timeline of AIV and AIC and report their utilization patterns. As shown in Fig. 6, both baselines fail to exploit effective AIC-AIV concurrency. MindSporeGL executes almost entirely on AIV, with AIC utilization below 10%, leaving the primary compute engine largely idle. In contrast, the AIC-based design heavily relies on AIC (over 80% utilization), while AIV is only sporadically invoked, resulting in mostly serialized execution. In both cases, the overlap between AIC and AIV remains minimal, leading to underutilization of available compute resources. This analysis reveals a key limitation of existing designs: they rely on a single engine at a time rather than coordinating both engines. This further motivates a coordination-first approach that jointly schedules AIC and AIV to unlock the full potential of Ascend NPUs.

3.2 SIMD-based Execution

GPUs follow Single Instruction Multiple Threads (SIMT), scheduling thousands of lightweight threads in warps to exploit fine-grained parallelism, whereas Ascend NPUs adopt Single Instruction Multiple Data (SIMD) and execute at a coarse, fixed-shape tile granularity through AIC/AIV [10, 22, 53]. This difference affects how SpMM handles irregular sparsity: SIMT can map work near the NNZ level, while SIMD issues whole tiles, so uneven nonzeros may translate into tile underutilization [1, 29, 37].

Equally importantly for system design, SIMD-style NPUs expose substantially more software-visible control over data movement and residency in the on-chip hierarchy. To sustain SIMD throughput, tiles must be explicitly formed and staged across L2/L1/L0 buffers (e.g., via software-orchestrated transfers and cache controls), so reuse is not only a consequence of access patterns but also a controllable scheduling decision [22]. In contrast, GPUs rely largely on OS-managed caching and warp scheduling; locality is primarily improved by shaping thread-level access, while cache residency and eviction are mostly opaque to the programmer [13, 53]. Therefore, SpMM optimization on NPUs places greater emphasis on tile-centric organization and explicit buffer-aware scheduling to maximize reuse and reduce buffer traffic under the SIMD execution model [1, 29, 37].

Experimental Analysis. To quantify the redundancy induced by tile-granular execution under sparsity, we conduct a controlled measurement on five real-world sparse matrices. We sweep the

Table 1. Fraction of redundant zeros within active tiles under different tile shapes. (TC: Tensor Core)

Dataset	GPU TC	NPU AIC Tiles			
	4×4	16×16	32×32	64×64	128×128
Cora	0.018	0.255	0.672	0.977	0.998
Reddit	0.031	0.392	0.845	0.996	0.998
Flickr	0.001	0.026	0.097	0.304	0.686
Wiki-RfA	0.021	0.275	0.691	0.981	0.999
Mouse_gene	0.153	0.724	0.929	0.986	0.996
Average	0.04	0.33	0.64	0.84	0.92

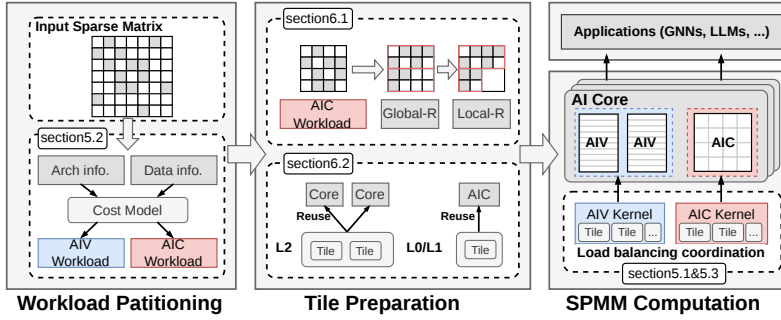


Fig. 7. Workflow of NeutronSparse.

tile size $t \in \{4, 16, 32, 64, 128\}$ and conceptually tile each matrix; following a tile-centric execution abstraction, a tile is marked active if it contains at least one nonzero, and the kernel processes active tiles as whole $t \times t$ units. We then report the fraction of zeros inside all active tiles as a proxy for tile-induced redundant work. As summarized in Table 1, redundancy increases sharply with coarser tiles: averaged over the five datasets, the zero fraction rises from 0.04 at 4×4 to 0.33 (16×16), 0.64 (32×32), 0.84 (64×64), and 0.92 (128×128). This trend highlights an architectural asymmetry: GPUs can exploit smaller tiles (e.g., 4×4 Tensor Core fragments) and fine-grained SIMT work mapping, keeping redundancy low and enabling flexible access to small blocks, whereas on Ascend NPUs the matrix engine operates at a much coarser granularity— 16×16 is the minimum AIC tile, making tile-level redundancy substantial even at the smallest supported tile size. Moreover, larger tiles are often favored on NPUs for better data reuse under SIMD-style execution, which further amplifies the redundancy penalty.

4 The NeutronSparse Framework

We develop NeutronSparse, an efficient SpMM framework on Ascend NPUs with two core functions: sparsity-aware coordination of heterogeneous engines and locality-aware tile orchestrating.

Overall Execution Flow. Figure 7 summarizes the end-to-end workflow of NeutronSparse, which turns an input sparse matrix into a coordinated AIC-AIV execution pipeline through three tightly coupled stages: workload partitioning, tile preparation, and SpMM computation. First, NeutronSparse uses the cost model in Section 5.2 to split the SpMM workload into two engines, routing sparse tiles to AIV and dense tiles to AIC, thus establishing a balanced starting point on the heterogeneous engines. Next, it performs tile preparation to make this split executable under the NPU’s tile-based model: tiles are formed and optimized via global-local reordering (Section 6.1) to better align nonzeros at tile granularity, and tile reuse is explicitly planned across the memory hierarchy (Section 6.2) by staging frequently accessed dense B rows for cross-core reuse while

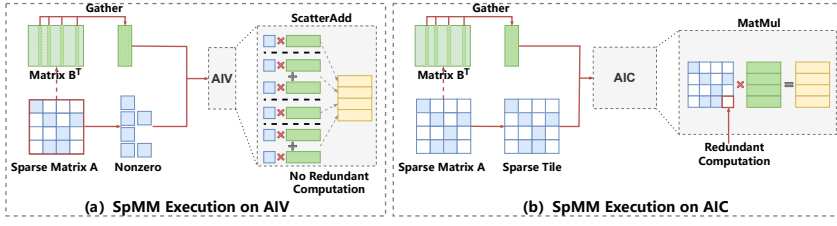


Fig. 8. SpMM execution on heterogeneous AIC and AIV engines: (a) Sparse, vector-oriented execution on AIV, (b) Dense, matrix-oriented execution on AIC.

choosing per-core tile shapes that saturate private buffers and favor efficient transfers. Finally, AIV and AIC kernels (Section 5.1) consume the prepared tile streams concurrently within each AI Core, and NeutronSparse applies the load-balancing coordination in Section 5.3 to overlap their progress and avoid engine idle time.

5 Sparsity-Aware Coordination of Heterogeneous Processing Engines

We propose a sparsity-aware coordination strategy that combines static heterogeneous workload partitioning (Section 5.2) with adaptive runtime coordinated pipelining (Section 5.3), enabling sustained overlap between AIC and AIV.

5.1 Execution Model of SpMM on AIC and AIV

In this section, we establish a high-level execution model for SpMM on Ascend NPUs by characterizing how the heterogeneous AIC and AIV engines process sparse-dense multiplication. They follow fundamentally different execution models, reflecting their architectural specialization for dense and sparse workloads, respectively.

AIV-based SpMM Execution. As illustrated in Figure 8a, AIV executes SpMM in a vector-oriented manner. During execution, AIV performs a *Gather* operation that uses the column indices of the sparse vector to fetch the corresponding row vectors from the dense matrix B . The gathered vectors are stored in the local on-chip buffer (UB) and subsequently accumulated into the output via *ScatterAdd*. By operating directly on vectors after *Gather*, AIV avoids redundant computation on zero entries and is therefore better suited to relatively sparse workloads.

AIC-based SpMM Execution. In contrast, AIC adopts a dense execution model optimized for matrix multiplication, as illustrated in Figure 8b. During execution, the column indices associated with each tile of the sparse matrix A are used to perform a *Gather* operation, which fetches the corresponding feature vectors from the dense matrix B . The gathered data are stored in the local on-chip buffer (L0B), while the corresponding tiles from matrix A are loaded into the local buffer (L0A). Since computation is still performed on tile-level operands, zero entries inside sparse tiles also participate in *MatMul*, introducing redundant work and making AIC more suitable for relatively dense tiles.

5.2 Heterogeneous Workload Partitioning

The goal of heterogeneous workload partitioning is to decompose a sparse matrix into tile-level workloads and assign them to the processing engine best matched to their sparsity characteristics. To this end, we first derive a sparsity threshold using an architecture-aware cost model, and then partition SpMM workloads across AIC and AIV accordingly.

5.2.1 Architecture-Aware Cost Model. Existing partitioning methods for heterogeneous SpMM are largely data-centric, e.g., splitting workloads by sparsity and routing them to the preferred

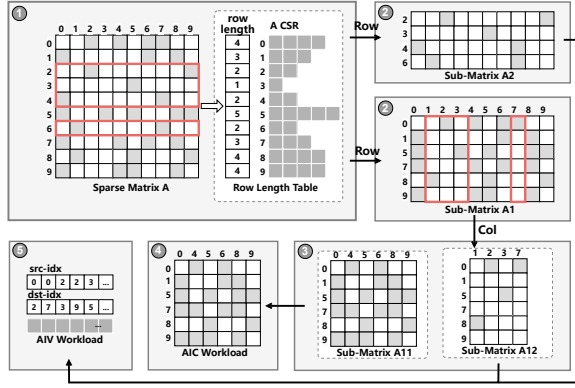


Fig. 9. Heterogeneous workload partitioning. The sparse matrix is first partitioned by rows into a dense part (A_1) and a sparse part (A_2), and then A_1 is further partitioned by columns into A_{11} and A_{12} .

engine [21, 38, 46, 50]. On Ascend NPUs, this is insufficient because partitioning must optimize execution-time balance rather than data characteristics alone: even if AIV receives the sparse part and AIC receives the dense part, overall performance degrades whenever their assigned workloads finish at mismatched pace, leaving one engine idle and reducing overlap. Therefore, we introduce an architecture-aware cost model that estimates the sparsity tile runtime on AIV and AIC from hardware execution characteristics, and uses predicted runtime to guide workload allocation so that AIV and AIC make balanced progress during execution.

Specifically, we derive an architecture-aware sparsity threshold α via a lightweight sparse-aware dry run. Here, $\alpha \in [0, 1]$ denotes a density boundary (NNZ ratio) that separates tiles better handled by AIV from those better handled by AIC: tiles with density below α are assigned to the sparse-friendly AIV, while the remaining tiles are assigned to AIC. The key is that this boundary should reflect real hardware efficiency, i.e., where AIV and AIC deliver comparable effective progress under their respective execution styles.

To obtain α , we run SpMM microbenchmarks on both engines using representative strategies (AIC with GEMM-style dense multiplication and AIV with vectorized sparse-oriented multiplication), and measure their empirical throughputs, P_{AIC} and P_{AIV} . For a matrix of size $M \times K$ with NNZ nonzeros, its density is $NNZ/(M \cdot K)$. We model the execution time on AIV as proportional to the number of nonzeros, and the execution time on AIC as proportional to the full tile volume:

$$\text{Cost}_{AIV}(NNZ) = \frac{NNZ}{P_{AIV}}, \quad \text{Cost}_{AIC} = \frac{M \cdot K}{P_{AIC}}. \quad (1)$$

We then choose the crossover point NNZ^* such that the two engines achieve a *balanced progress rate* according to their architectural capacity ratio r (e.g., $r=2$ since each AI Core contains two AIV units and one AIC unit). Concretely, we find NNZ^* that makes $\text{Cost}_{AIV}(NNZ)/\text{Cost}_{AIC}$ as close as possible to r :

$$NNZ^* = \arg \min_{NNZ} \left(\frac{\text{Cost}_{AIV}(NNZ)}{\text{Cost}_{AIC}} - r \right)^2. \quad (2)$$

Minimizing this objective yields $\frac{NNZ^*}{P_{AIV}} = r \cdot \frac{M \cdot K}{P_{AIC}}$, and thus the sparsity threshold

$$\alpha = \frac{NNZ^*}{M \cdot K} = r \cdot \frac{P_{AIV}}{P_{AIC}}. \quad (3)$$

In the following, we use α to classify tiles by density and construct tile-level workloads for heterogeneous execution.

5.2.2 AIV-AIC Workload Partitioning. Using the density threshold α , we partition SpMM workloads across AIV and AIC via a lightweight two-stage row-column extraction. We choose extraction-based partitioning because it (i) can be implemented by a single linear scan over the sparse structure, (ii) directly targets workload skew dominated by a few long rows/columns, and (iii) yields submatrices that naturally match the data formats and execution paths of AIV (irregular sparse vectors) and AIC (regularized dense tiles).

Extraction Metric and Rule. Let v denote a sparse vector in A (a row or a column, depending on the extraction dimension). We define its nonzero length as

$$Len(v) = |\{x_j \mid x_j \in v, x_j \neq 0\}|, \quad (4)$$

and convert the density boundary α into a length threshold

$$Thres = \alpha \cdot K, \quad (5)$$

where K is the vector dimension. Vectors with $Len(v) \leq Thres$ are treated as sparse workloads and assigned to AIV, while the remaining vectors are assigned to AIC.

Two-Stage Row-Column Extraction. As shown in Figure 9, we apply the above rule in two stages. We first traverse A once to compute $Len(\cdot)$ and extract sparse rows to AIV; the remaining rows form a denser submatrix are routed to AIC. Since row extraction alone may still leave the AIC submatrix with highly sparse columns, we apply the same extraction rule once more along the column dimension within the AIC submatrix, moving short columns back to AIV. This two-stage procedure decomposes A into a dense core handled by AIC and sparse fringes handled by AIV, providing a static, sparsity-aware workload decomposition that is well aligned with heterogeneous execution. We store the AIC-assigned dense core in a BitMap to enable sequential access and tile-based computation, while storing the AIV-assigned sparse fringes in COO to avoid zero storage and support irregular gather/scatter accesses.

5.3 Adaptive AIV-AIC Coordinated Pipelining

Static partitioning (Section 5.2) provides an initial AIV/AIC mapping guided by the hardware-aware threshold α . During execution, however, SpMM on Ascend runs as a concurrent two-engine pipeline: AIV and AIC process their assigned workloads in parallel within an AI Core, and the end-to-end latency of each iteration is determined by the slower engine. Due to irregular sparsity and runtime variation (e.g., skewed rows/columns and cache effects), the two engines may drift out of balance, causing the faster engine to stall and leaving pipeline bubbles. We therefore introduce an adaptive coordinated pipeline that (i) monitors the relative progress of AIV and AIC at coarse granularity and (ii) migrates a small amount of remaining work from the bottleneck engine to the idle engine, thereby sustaining overlap and minimizing idle gaps.

Progress Monitoring at Epoch Granularity. Many target workloads repeatedly invoke SpMM across a large number of iterations/epochs (e.g., GNN training typically runs hundreds of epochs and sparse attention executes similar sparsity patterns across layers and steps). This repetition makes epoch-level adjustment effective: a decision made from the previous epoch's timing feedback remains informative for subsequent epochs, while keeping the monitoring overhead negligible. Accordingly, we execute SpMM in epochs and collect per-epoch execution time for each engine: Δt_{AIC} for AIC and Δt_{AIV} for AIV (Figure 10, top). We quantify imbalance with a dimensionless skew ratio

$$Skew \triangleq \frac{\max(\Delta t_{AIC}, \Delta t_{AIV})}{\min(\Delta t_{AIC}, \Delta t_{AIV})} \geq 1. \quad (6)$$

When $Skew \leq 1 + \epsilon$, the engines are considered sufficiently balanced. We trigger migration only if $Skew > 1 + \epsilon$, where ϵ is a small tolerance (e.g., 0.05) to avoid oscillation. This lightweight mechanism

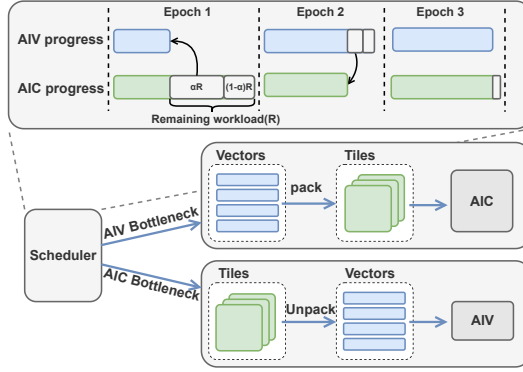


Fig. 10. Load migration under runtime load imbalance.

serves as a short warm-up process after the initial partition, using only epoch-level timing to correct residual imbalance with negligible overhead. Our current target is mainly static SpMM workloads. For dynamic workloads, matrix evolution is often gradual, so the original partition can remain effective for multiple epochs and migration can be re-triggered after sufficient changes accumulate.

Adaptive Workload Migration. When imbalance is detected, we migrate part of the residual workload from the bottleneck engine to the other engine to eliminate pipeline bubbles. Let R denote the remaining work on the slower engine at the moment the faster engine would otherwise become idle (Figure 10, top). Migration follows a sparsity-guided direction consistent with the execution models of AIV and AIC (Figure 10, bottom):

- **AIC is the bottleneck** ($\Delta t_{AIC} > \Delta t_{AIV}$). We preferentially extract sparser tiles from AIC's residual work and offload them to AIV. Concretely, we decompose candidate tiles into sparse vectors (index-value lists).
- **AIV is the bottleneck** ($\Delta t_{AIV} > \Delta t_{AIC}$). We preferentially densify and pack relatively dense sparse vectors from AIV's residual work into tile-aligned matrix blocks and offload them to AIC. Concretely, we merge denser rows/segments to form matrix tiles.

During the local reordering stage (Sec. 6.1), when the matrix is organized into fixed-shape tiles, we simultaneously record the sparsity of each tile. Online migration then directly uses these precomputed sparsity values to identify sparser tiles from the residual workload.

We use the threshold α (Section 5.2.1) as a hardware-aware target split between AIV- and AIC-suited workloads. Given the residual workload R , we partition it into R'_{AIV} and R'_{AIC} to balance execution time across the two engines. Since AIV and AIC follow different cost models (Eq. (1)), the partition is not based on row or tile counts. Instead, we select $R'_{AIV} \subseteq R$ such that its nonzero count and the remaining dense workload assigned to AIC satisfy:

$$\frac{\text{NNZ}(R'_{AIV})}{M(R'_{AIC}) \cdot K} \approx \alpha = r \cdot \frac{P_{AIV}}{P_{AIC}}. \quad (7)$$

where $R'_{AIC} = R \setminus R'_{AIV}$, and $M(R'_{AIC})$ denotes the number of rows in the AIC-assigned workload. This condition aligns the workloads according to the throughput capabilities of the two engines and corresponds to the sparsity threshold α .

6 Locality-Aware Tile Orchestrating

On NPUs, SpMM is fundamentally governed by the tile as the minimum unit of computation and data movement. While many GPU-style reorderings can in principle be applied to NPUs, directly reusing NNZ-/thread-level transformations is often unnecessarily fine-grained for tile execution and

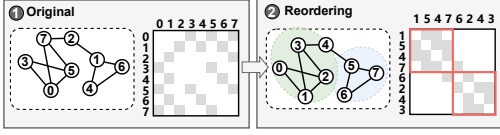


Fig. 11. Global reordering.

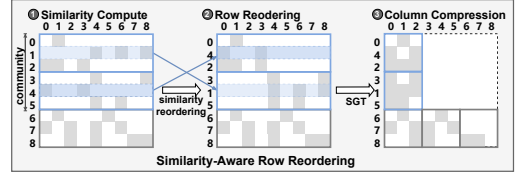


Fig. 12. Local reordering.

can introduce prohibitive preprocessing overhead. This motivates a different optimization principle for NPUs: improve locality at tile granularity, and trade heavy preprocessing for lightweight, hardware-effective transformations. Equally importantly, SIMD-style NPUs expose much more explicit control over tile movement and residency across on-chip buffers. Since tile materialization, staging, and eviction are largely software-orchestrated, performance depends not only on how tiles are formed, but also on where they are kept and how often they are reused under buffers.

Therefore, we adopt a lightweight, tile-centric orchestrating strategy that jointly optimizes tile formation and tile reuse to reduce both redundant tile execution and unnecessary memory traffic. Specifically, we introduce two complementary mechanisms: (i) a global-local tile reordering scheme that coarsely reshapes row/column locality and then cheaply refines tile alignment to form denser, better-aligned tiles, and (ii) hierarchical tile reusing that improves reuse both across AI Cores (by retaining high-frequency dense B tiles in shared cache) and within each AI Core (by choosing tile shapes that maximize buffer residency and enabling aligned write-back). Together, these techniques improve sustained throughput under irregular sparsity.

6.1 Global-Local Tile Reordering

Prior reorderings for GPUs often rely on fine-grained row/column permutations to maximize NNZ-level alignment and clustering [9, 21, 29, 54]. For tile-based NPUs, we instead target tile-level density and alignment with a lightweight two-level global-local reordering scheme. The **global stage** performs a coarse permutation over both rows and columns to form a small number of large clusters, capturing dominant structural correlations with low overhead. The **local stage** then refines each cluster at row-window granularity: it groups rows with similar sparsity patterns to improve intra-tile alignment and to facilitate column compaction during tile construction. Together, the two stages achieve tile-friendly layouts while keeping preprocessing lightweight.

Global Reordering. The global stage aims to expose coarse structural locality by grouping strongly related rows and columns into clusters. Here, rows are considered related if they share similar nonzero-column patterns, while columns are considered related if they share similar nonzero-row patterns. As shown in Figure 11, we model the sparse matrix as a bipartite graph and apply Rabbit Order [5, 29] to reorder the original row and column indices so that related rows and columns become adjacent and form clusters. Unlike aggressive clustering schemes that iterate until convergence [21, 29, 54], we intentionally limit the number of clusters. This reordering makes originally scattered nonzeros fall into more localized row and column ranges, forming denser block-like regions in the reordered matrix. Such a coarse permutation captures the major structural correlations needed by the subsequent local reordering, while keeping preprocessing overhead low and avoiding unnecessary fragmentation for tile-oriented execution.

Local Reordering. After global clustering, residual sparsity still exists inside each cluster. Rather than continuing expensive element-level permutations over both rows and columns, the local stage refines locality at tile granularity. As shown in Figure 12, we partition each cluster into row windows whose height matches the target tile shape (i.e., M , set to 128 in Section 5.2). We then

reorder rows *within each cluster* by grouping rows with similar sparsity patterns into the same window, which increases nonzero alignment across columns and exposes empty column regions that can be compacted during tile construction. Concretely, we use the current row windows as anchors by selecting one representative row per window, and then rebuild windows greedily: for each anchor row, we pick the $(M-1)$ most similar unassigned rows based on Jaccard similarity over their nonzero column sets, form a new window, and remove assigned rows to avoid duplication. Importantly, this refinement only permutes rows and does not alter the global column order, making it much cheaper than full element-level reordering while still producing denser, better-aligned tiles.

In summary, the global-local reordering strategy reflects a deliberate trade-off. The global stage provides a coarse approximation of element-level reordering to capture long-range structure, while the local stage performs low-cost, tile-oriented refinement to align with SIMD execution. This design contrasts with GPU-oriented approaches that pursue maximal fine-grained alignment, and instead tailors reordering to the needs and limits of tile-based NPUs.

6.2 Hierarchical Tile Reusing

Tile reusing on NPUs can be reasoned at two distinct levels: reusing across AI Cores and reusing within a single AI Core.

6.2.1 Inter-core Tile Reusing. After partitioning the sparse matrix A by rows (or row windows), different AI Cores process disjoint subsets of A . However, both AIV and AIC repeatedly perform Gather to fetch rows of the dense matrix B according to the column indices of nonzeros in A . As a result, even when AI Cores work on independent tiles of A , their accesses to B exhibit substantial overlap. This makes B a natural target for inter-core tile reusing.

Unlike GPUs, where L2 cache behavior is largely managed by hardware and the operating system, Ascend NPUs allow software to explicitly control whether memory accesses are cached in L2 [4, 22]. We exploit this capability to orchestrate B as a shared working set across AI Cores. Guided by the global reordering from Section 5.1, we process the tiles cluster by cluster. For each active cluster, we identify the rows of B that are most frequently accessed by tiles within that cluster and proactively stage them into the shared L2 cache. During the processing of this cluster, accesses to these high-frequency rows of B are marked as cacheable, while accesses to sparse tiles of A and less frequently used rows of B bypass L2.

The size of the staged B working set is bounded by the L2 capacity (typically 20–32 MB). In practice, we conservatively cap it at approximately 80% of the available L2 space to tolerate system-reserved entries and runtime fluctuations. This design enables multiple AI Cores to reuse the same B tiles while processing different row windows, improving the reuse efficiency of hot B tiles and reducing redundant HBM traffic induced by repeated Gather.

6.2.2 Intra-core Tile Reusing. Within an AI Core, tile reusing mainly takes the form of tile shaping, where the tile shape determines how much computation can be performed before operands and partial results are evicted from on-chip buffers. By keeping data resident longer, a well-orchestrated tile improves intra-core reuse efficiency. In our design, each tile corresponds to one row window produced by local reordering, with height M and width K in the sparse window, and width N in the associated dense matrix B . Selecting a proper (M, N, K) requires balancing compute throughput, buffer capacity limits, and write-back efficiency.

Architectural Constraints. Ascend 910B provides three on-chip buffers per AIC: L0A and L0B (64 KB each) for fp16 operands of the $M \times K$ A -tile and the $K \times N$ B -tile, and L0C (256 KB) for fp32 accumulation of the $M \times N$ output tile. To overlap computation with data transfer, we employ double-buffer pipelining (see Section 6.2 for details), so a tile can use at most half of each buffer.

Let $C_A=C_B=64$ KB and $C_C=256$ KB denote the capacities of L0A, L0B, and L0C, respectively. Since fp16 and fp32 occupy 2 and 4 bytes per element, the residency constraints are:

$$\underbrace{(MK) \cdot 2}_{A\text{-tile bytes}} \leq \frac{1}{2}C_A, \quad \underbrace{(NK) \cdot 2}_{B\text{-tile bytes}} \leq \frac{1}{2}C_B, \quad \underbrace{(MN) \cdot 4}_{C\text{-tile bytes}} \leq \frac{1}{2}C_C. \quad (8)$$

Substituting $C_A=C_B=64$ KB and $C_C=256$ KB yields:

$$MK \leq 16384, \quad NK \leq 16384, \quad MN \leq 32768,$$

where M , N , and K must also be multiples of 16 to satisfy AIC execution requirements.

Write-back Preference. The output matrix is stored in row-major layout and written back row by row. As a result, consecutive output tiles advance the global write pointer by $N \cdot 4$ bytes. When this stride is aligned to the fixpipe 512-byte transaction granularity, write-back achieves higher bandwidth; otherwise, split transactions may degrade efficiency [22, 56]. This favors tile shapes with N being a multiple of 128.

Maximizing AIC Utilization. AIC throughput is determined by the tile volume MNK . Given the input-buffer constraints $MK \leq 16384$ and $NK \leq 16384$, the maximum achievable MNK is reached when both bounds are tight. One such configuration is (128, 128, 128), which fully utilizes L0A and L0B and satisfies the preferred $N \geq 128$ for efficient write-back.

From this peak-throughput point, we can increase N and reduce K proportionally while keeping $NK=16384$ unchanged. This preserves the same AIC throughput but reduces the A -side footprint MK , thereby improving compute efficiency. As a result, (128, 256, 64) achieves the same AIC throughput as (128, 128, 128) with lower input traffic. Specifically, the fp16 input traffic per tile equals the combined footprints of the A - and B -tiles, i.e., $2(MK + NK)$ bytes. Thus, (128, 256, 64) transfers $2(128 \cdot 64 + 256 \cdot 64) = 48$ KB, whereas (128, 128, 128) transfers $2(128 \cdot 128 + 128 \cdot 128) = 64$ KB.

Final Tile Choice. The above trade cannot be extended indefinitely, since the fp32 output must satisfy $MN \leq 32768$ under double-buffered L0C, which bounds $N \leq 256$ when $M = 128$. Combining AIC utilization, input traffic, buffer residency, and write-back efficiency, we adopt

$$(M, N, K) = (128, 256, 64)$$

as the default tile configuration. This choice maximizes effective compute density while respecting all on-chip constraints, enabling sustained intra-core tile reuse.

7 Implementation

Vector Tiles Merging (AIV-side). Figure 13 illustrates vector tiles merging in the AIV runtime. In sparse SpMM, AIV repeatedly gathers rows of B by `col_idx` and accumulates them to output rows by `row_idx`. When N (the column width of dense matrix B) is not divisible by AIV's 128-wide vector width, each gather-accumulate activates only a fraction of SIMD lanes, wasting both compute and bandwidth. The optimization groups entries sharing the same `row_idx` and packs multiple gathered B rows into one widened vector tile (Figure 13, optimized), concatenating them to reach a 128-aligned shape, so a single AIV step processes a denser tile and then performs one *ScatterAdd* to the target output row, improving lane utilization and reducing fragmented memory accesses.

Row Window List Migration (AIC-side). Figure 14 illustrates row window list migration in the AIC runtime. After tile preparation, computation is organized as a row window list, where each row window corresponds to one input tile and is assigned to an AIC. A naive row-wise assignment can be highly imbalanced because different lists contain different numbers of row windows, leading to long-tail stragglers. The migration step reshuffles row windows across the list to form a balanced list (Figure 14, middle), interleaving heavy and light windows without splitting windows or changing

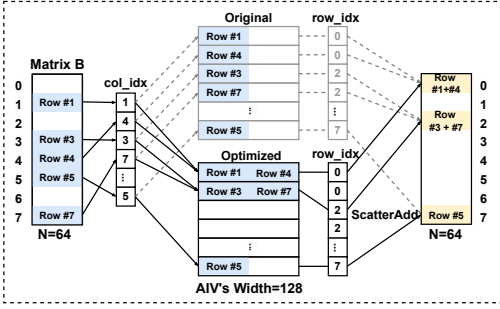


Fig. 13. Vector tiles merging.

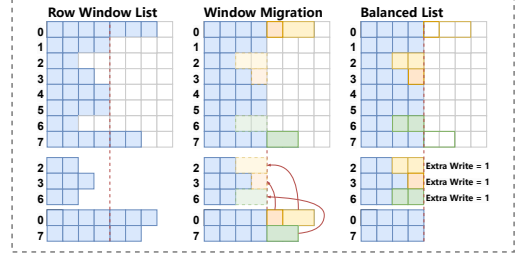


Fig. 14. Row window list reordering.

tile shapes. This incurs one extra write to reconcile the migrated windows with their original output locations, but it substantially reduces stragglers and improves throughput.

Double Buffer Pipelining. We implement double buffer pipelining in both AIC and AIV to overlap computation with data movement. Each engine alternates between two buffer sets, allowing one tile to execute while the next tile is being prefetched into on-chip buffers or written back. This design hides data transfer latency behind computation and enables continuous tile execution, at the cost that each tile may occupy at most half of the local buffer capacity.

8 Evaluation

8.1 Experimental Setup

Environments. The experiments are conducted on two servers: an NPU server equipped with an Ascend 910B (24 AI Cores and 64 GB HBM) enabled with CANN v8.0.1 runtime, and a GPU server equipped with an NVIDIA A100 (Ampere, 108 SMs each consisting of 64 CUDA Cores and 4 Tensor Cores, 80 GB HBM) enabled with CUDA v12.3. The server runs Ubuntu 18.04 with Linux kernel 4.13.0.

Datasets. We use four real-world GNN benchmarks (cora [27], ogbn-arxiv [15], reddit [11], and amazon-product [26]) and sixteen sparse matrices from the SuiteSparse [6]. Table 2 lists the datasets. “Abbr.” denotes dataset abbreviations, “Row&Col” gives the row and column dimensions of the input sparse matrix ($M \times K$), “NNZ” is the number of nonzeros, and “AvgL” indicates the average row length. Table 2 summarizes the datasets and their key structural characteristics. “Abbr.” denotes dataset abbreviations, “Row&Col” gives the matrix dimensions, “NNZ” is the number of nonzeros, “AvgL” is the average row length, “Dens.” reports the matrix density scaled by 10^{-3} , “Skew” measures the fraction of NNZ contained in the top 10% rows ranked by row NNZ, and “Empty” reports the fraction of empty 16×16 tiles before tile reordering.

Competitor baselines. In performance evaluation, we compare NeutronSparse against representative SpMM baselines on both Ascend NPUs and GPUs. On NPUs, we include MindSporeGL [22, 59], which mainly executes sparse kernels on AIV, and an AIC-based SpMM design reimplemented from prior Ascend work [30], which primarily relies on AIC execution. On GPUs, we use cuSPARSE [31], NVIDIA’s vendor-optimized sparse computation library with deeply tuned SpMM and SpMV kernels, DTC-SpMM [9], a SOTA SpMM framework that exploits Tensor Cores, and HC-SpMM [21], a recent GPU SpMM framework that jointly leverages CUDA Cores and Tensor Cores to improve utilization under sparsity. By default, we use $N = 256$ as the column width of the dense matrix B , unless stated otherwise.

Table 2. Datasets

Dataset	Abbr.	Row&Col	NNZ	AvgL	Dens. (10^{-3})	Skew	Empty Tiles
cora	CR	2,708	10,556	3.9	1.44	0.32	0.05
wiki-RfA	WR	11,380	362,053	31.8	2.80	0.39	0.01
dawson5	DA	51,537	1,010,777	19.6	0.38	0.14	0.98
olafu	OL	16,146	1,015,156	62.9	3.89	0.12	0.99
ogbn-arxiv	OA	169,343	2,315,598	13.6	0.08	0.50	0.86
pattern1	PA	19,242	9,323,432	484.5	25.18	0.16	0.73
mip1	MP	66,463	10,352,819	155.8	2.34	0.17	0.99
mycielskian15	M15	24,575	11,111,110	452.1	18.40	0.42	0.89
nd12k	ND	36,000	14,220,946	395.0	10.97	0.12	0.95
human_gene1	HG	22,283	24,669,643	1107.1	49.68	0.24	0.49
F1	F1	343,791	26,837,113	78.1	0.23	0.44	0.98
ML_Laplace	ML	377,002	27,689,972	73.4	0.19	0.10	0.99
Fault_639	FA	638,802	28,614,564	44.8	0.07	0.12	0.99
mouse_gene	MG	45,101	28,967,291	642.3	14.24	0.41	0.50
audikw_1	AU	943,695	77,651,847	82.3	0.09	0.24	0.99
mycielskian17	M17	98,304	100,245,742	1019.8	10.37	0.46	0.94
reddit	RD	232,965	114,615,892	492.0	2.11	0.46	0.01
amazon-product	AP	2,449,029	123,718,152	50.5	0.02	0.45	0.96
mycielskian18	M18	196,607	300,933,832	1530.6	7.79	0.48	0.95
mycielskian19	M19	393,215	903,194,710	2296.9	5.84	0.50	0.96

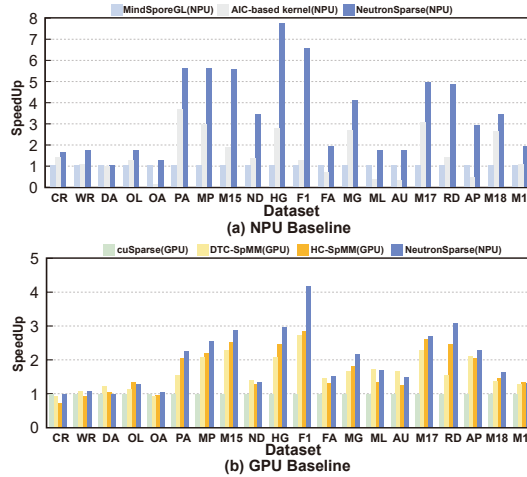


Fig. 15. SpMM performance on Ascend 910B and Nvidia A100

8.2 Overall Comparison

We conduct a comprehensive performance comparison of NeutronSparse, MindSporeGL, cuSPARSE, DTC-SpMM, and HC-SpMM. Figure 15 shows the results: on Ascend NPUs, speedups are normalized to MindSporeGL, while on GPUs, speedups are normalized to cuSPARSE.

Comparison with the NPU baseline. Compared to both MindSporeGL and the AIC-based baseline, NeutronSparse consistently achieves superior performance, with an average speedup of 3.48 $\times$ over MindSporeGL and 3.25 $\times$ over the AIC-based design. The two baselines expose complementary limitations: MindSporeGL relies solely on AIV, leaving AIC underutilized, while the AIC-based design centers on dense-tile execution, incurring substantial redundant computation on zero entries and executing AIC and AIV largely in a serial manner. In contrast, NeutronSparse jointly exploits AIC and AIV through fine-grained partitioning and coordinated execution, enabling

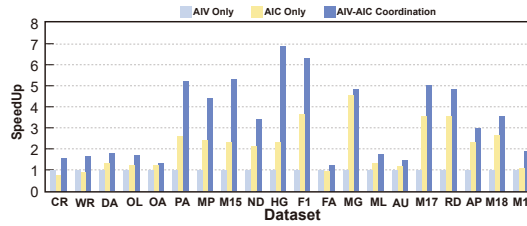


Fig. 16. Performance gain of AIV-AIC Coordination.

both high compute efficiency and effective overlap. The gains are particularly pronounced on higher-density matrices (e.g., human_gene1), where leveraging AIC becomes critical and NeutronSparse achieves up to 7.78 $\times$ speedup.

Comparison with the GPU baseline. Compared to cuSPARSE, DTC-SpMM, and HC-SpMM, NeutronSparse exhibits superior performance across most datasets, achieving an average speedup of 1.98 $\times$ (up to 4.19 $\times$), 1.27 $\times$ (up to 1.99 $\times$), and 1.18 $\times$ (up to 1.47 $\times$), respectively. cuSPARSE and DTC-SpMM mainly rely on a single dominant execution engine (CUDA cores or Tensor Cores), which performs well only on the regions that match its compute pattern. Although HC-SpMM combines CUDA cores and Tensor Cores to cover different workload types, its kernels are still executed in a largely staged manner and cannot concurrently run heterogeneous engines to maximize sustained throughput. In contrast, NeutronSparse can overlap sparse processing and dense tile computation on the NPU's heterogeneous engines via online coordination, thus handling mixed sparsity patterns more effectively. DTC-SpMM and HC-SpMM achieve better performance on cora, wiki-RfA, and dawson5. Because these datasets are small in scale, the benefits of NeutronSparse's coordinating execution are insufficient to outweigh the additional scheduling overhead.

8.3 Analysis of AIV-AIC Coordination

Performance Gain. To evaluate the performance benefits of our proposed AIV-AIC coordinating optimization for SpMM, we use 15 datasets of varying scales and sparsity. We compare three kernels: (1) AIV-only: a baseline kernel that exclusively utilizes AIV; (2) AIC-only: a baseline kernel using only AIC; and (3) AIV-AIC Coordination: the hybrid heterogeneous kernel applied in NeutronSparse. Figure 16 shows the normalized speedup with the AIV-only kernel as the baseline. AIV-AIC Coordination achieves the best performance on all test matrices, with average speedups of 3.71 $\times$ and 1.98 $\times$ over the AIV-only and AIC-only kernels, respectively. The largest speedups are observed on human_gene1 and F1, whose power-law sparsity induces clear separation between sparse and dense regions. This heterogeneity amplifies the benefit of AIV-AIC coordination, allowing NeutronSparse to map sparse work to AIV and dense tiles to AIC more effectively.

Online Workload Migration. Figure 17 shows the online workload migration between AIV and AIC on two datasets. NeutronSparse monitors the real-time progress of the two engines in the coordinated pipeline and migrates tiles from the slower engine to the faster one to eliminate stragglers. Starting from the initial static partition, the AIV/AIC workload ratio quickly converges to a stable range within the first 10 epochs, and the per-epoch time drops accordingly, yielding an average speedup of 3.83 $\times$ over the initial execution time after convergence. The workload ratio obtained via online migration is not strictly optimal, but is already close to the optimal time-balanced solution. Specifically, migration stops once the execution time difference between AIV and AIC falls below a small tolerance (e.g., 5%), which bounds the final ratio within a near-optimal region. Starting from the optimal ratio could further improve performance, but the gain would be limited. Since the final ratio is already tightly bounded and the migration overhead is small

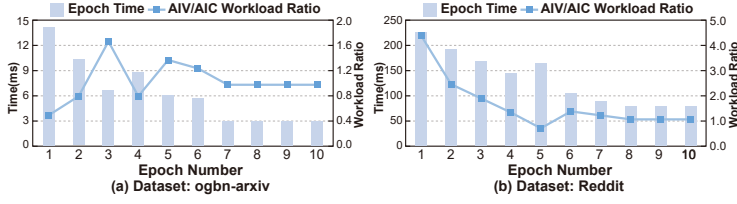


Fig. 17. Dynamic balance AIV-AIC workload.

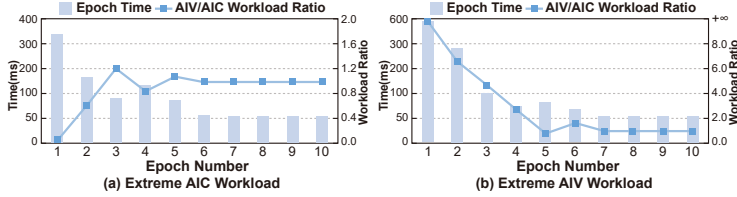


Fig. 18. Convergence of online workload migration under extreme initial-skew cases.

compared to the overall runtime (e.g., 5% among hundreds of epochs), the additional improvement is marginal (up to 10%). By adapting the partition online, NeutronSparse sustains time-balanced progress between AIV and AIC and achieves stable throughput improvements.

We further construct extreme-skew cases from reddit by artificially adjusting the number of nonzeros in each row, so that the initial workload is assigned almost entirely to either AIC or AIV. Even under such extreme imbalance, online migration still converges quickly. This process can be viewed as a bisection-style iterative balancing procedure on the residual imbalance: each round repartitions the remaining workload according to the observed execution-time skew, so the imbalance shrinks geometrically and the number of adjustment rounds grows only logarithmically with the initial skew. As shown in Fig. 18, the adjustment converges within 7 rounds, only slightly slower than on real workloads despite the highly imbalanced initialization. Notably, we observe that the final stable ratio in case (b) is AIC-dominated because AIC provides significantly higher effective compute throughput than AIV, even when the raw sparsity pattern appears more suitable for vector-style execution. As a result, after balancing execution time, a larger fraction of workload is assigned to AIC to fully utilize its higher processing capacity.

Sensitivity to the initial sparsity threshold We further study the sensitivity of heterogeneous workload partitioning to the initial sparsity threshold. In our implementation, this threshold determines the initial AIV/AIC split: rows with density above the threshold are assigned to AIC, while the others are assigned to AIV. Since the densities of the target sparse matrices are mostly on the order of 10^{-3} , we sweep the threshold from 1×10^{-3} to 10×10^{-3} and evaluate the resulting performance on ogbn-arxiv and reddit. As shown in Fig. 19, the default threshold 2×10^{-3} already lies near the optimal-performing region. In particular, thresholds from 1×10^{-3} to 3×10^{-3} lead to only 6.4% performance variation on average, while larger deviations cause more noticeable degradation due to a worse initial workload balance. This result suggests that the lightweight cost model is sufficiently accurate to provide a good starting point, while the remaining mismatch can be corrected by online migration.

8.4 Analysis of Tile Orchestrating

Performance Gain. We quantify the benefit of the two techniques in Section 6 via an ablation study on 15 datasets (Figure 20). The baseline is a plain tile-based SpMM using a fixed tile shape $(M, N, K) = (128, 128, 64)$; we then progressively enable global-local reordering (Baseline+Reorder)

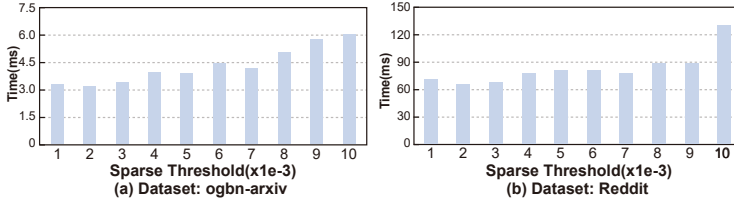


Fig. 19. Sensitivity of SpMM performance to the initial sparsity threshold.

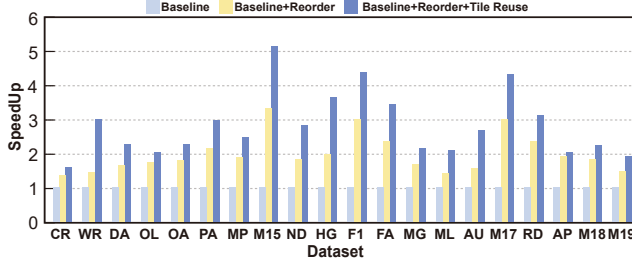


Fig. 20. Performance gain of locality-aware tile scheduling.

and further add hierarchical tile reusing (Baseline+Reorder+Tile Reuse), reporting speedups normalized to the baseline. Overall, Baseline+Reorder achieves an average speedup of 2.17 $\times$ by clustering correlated rows/columns into communities and forming denser row-window tiles, which reduces redundant work on sparsely populated tiles. On top of that, Tile Reuse provides an additional 1.42 $\times$ speedup over Baseline+Reorder by reducing repeated gathers of dense B rows and increasing per-core tile residency so that loaded data is reused across more MAC operations before being evicted. The larger gains appear on datasets such as F1 and M15, which exhibit strong power-law sparsity with pronounced structural skew; this makes our global-local reordering particularly effective at clustering highly correlated rows/columns and producing much denser row-window tiles.

Impact on Tile Density. We analyze the effectiveness of Global-local Reordering on AIC workloads, including Global Reordering (GR) and Local Reordering (LR). We use density improvement as the metric, defined as the multiplicative increase in the nonzero ratio after reordering, i.e., $DensityImprovement = \rho_{after} / \rho_{before}$, where $\rho = NNZ / (M \cdot K)$ is the tile density. We start from an unoptimized AIC workload and progressively enable GR and LR. Figure 21 reports the normalized density improvement. Compared with the baseline, baseline+GR increases tile density by clustering correlated rows/columns via community detection, achieving an average 3.44 $\times$ improvement. Building on baseline+GR, baseline+GR+LR further boosts density by reordering rows within each cluster to better align nonzeros across columns, and then compacting away empty columns during tile construction, achieving an average 10.06 $\times$ improvement. GR and LR show larger improvements on F1, Fault_639, and audikw_1 because these matrices have shorter average row lengths and thus contain more zero-only regions that can be eliminated by clustering and compaction. In contrast, pattern1 benefits less since its higher baseline density leaves limited room for further compression.

Performance with Varying Tile Sizes. Tile shape has a clear impact on overall performance because it directly affects compute efficiency, data reuse, and buffer utilization. To evaluate this effect, we compare our default tile shape (128, 256, 64) with four alternatives: (16, 16, 16), (32, 32, 32), (64, 64, 64), and (128, 128, 128). The results show that (128, 256, 64) consistently achieves the best performance, outperforming these alternatives by 3.39 $\times$, 2.84 $\times$, 1.64 $\times$, and 1.31 $\times$ on average, respectively. Smaller tiles suffer from lower compute intensity and weaker data reuse, while larger

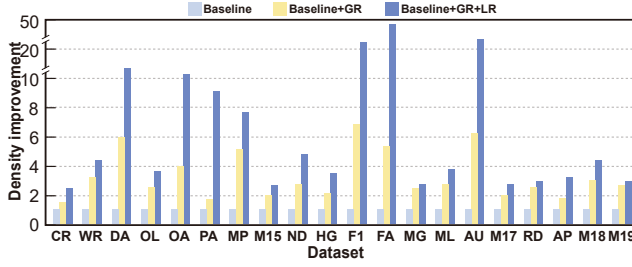


Fig. 21. Density improvement of global-local reordering. "GR": global reordering, "LR": local reordering.

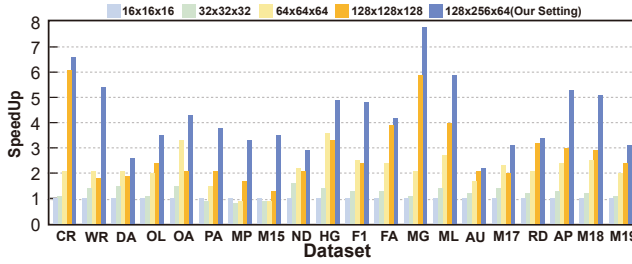


Fig. 22. Performance with varying tile sizes.

symmetric tiles are less effective in balancing on-chip buffer usage and write-back efficiency. These results validate that our default tile shape provides a better trade-off among compute-to-memory ratio, buffer constraints, and output efficiency.

8.5 Scalability Analysis

In this experiment, we compare NeutronSparse with all baselines with different column widths N (set to $\{32, 64, 128, 256, 512\}$) of matrix B over six datasets (PA, MG, F1, RD, AP, M18). Figure 23 shows the results. Across different N sizes, NeutronSparse consistently outperforms the baselines. Specifically, as N increases from 32 to 512, NeutronSparse achieves $1.35\times$ – $4.04\times$ GFLOPs improvement, MindSporeGL $1.18\times$ – $2.09\times$, cuSPARSE $1.26\times$ – $2.38\times$, DTC-SpMM $1.31\times$ – $3.68\times$, and HC-SpMM $1.32\times$ – $3.88\times$. We observe that NeutronSparse, DTC-SpMM, and HC-SpMM exhibit higher GFLOPs improvement than cuSPARSE and MindSporeGL. That is because the larger N raises the compute-to-memory ratio, thereby amplifying the computation bottlenecks of cuSPARSE and MindSporeGL, whose advantages mainly lie in memory access. In contrast, NeutronSparse, DTC-SpMM, and HC-SpMM employ matmul-based units, where the higher compute-to-memory ratio helps alleviate the memory access bottleneck of density computing units.

8.6 Preprocessing Overhead Analysis

We next analyze the preprocessing overhead introduced by NeutronSparse. Since this overhead is mainly determined by dataset scale, we use representative small-, medium-, and large-scale datasets to present the trend clearly.

Amortized Overhead Analysis. We evaluate the amortized overhead of the two preprocessing steps introduced by NeutronSparse, heterogeneous workload partitioning and global-local reordering, under long-running training workloads. Specifically, we run 200 epochs of GCN training and report the runtime breakdown in Table 3. Both preprocessing steps are executed once before training and their costs are amortized across all epochs. On average, heterogeneous workload partitioning accounts for approximately 3.04% of the total runtime, while global-local reordering

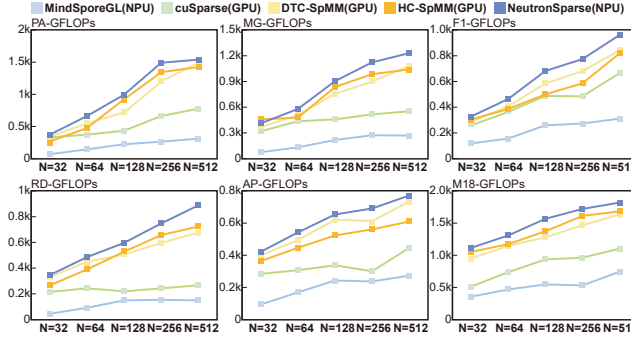


Fig. 23. Scaling performance when varying matrix sizes.

Table 3. Runtime breakdown of 200-epoch GNN training (seconds and percentage of total time).

Dataset	Runtime Breakdown (Ratio)			Total (s)
	Partition	Reorder	GCN Exec.	
cora	0.01 (2.38%)	0.018 (4.29%)	0.392 (93.33%)	0.42
ogbn-arxiv	0.016 (2.35%)	0.015 (2.21%)	0.649 (95.44%)	0.68
reddit	9.34 (3.48%)	6.32 (2.35%)	252.84 (94.17%)	268.5
amazon-product	10.37 (3.12%)	9.62 (2.90%)	311.81 (93.98%)	331.8
mycielskian18	15.88 (3.87%)	10.34 (2.52%)	384.08 (93.61%)	410.3

Table 4. Preprocessing overhead comparison.

Dataset	Rows&Col	Overhead	
		NeutronSparse	DTC-SpMM
cora	2,708	0.4s	1.7s
ogbn-arxiv	169,343	0.7s	12.3min
amazon-product	1,569,960	6.5min	17 h

contributes about 2.85%. In contrast, GNN training is dominated by SpMM execution, which consistently accounts for over 93% of the end-to-end runtime across all datasets. These results indicate that the overhead of our preprocessing is negligible while enabling substantial performance benefits during iterative SpMM execution.

Overhead Comparison. We compare the preprocessing overhead of NeutronSparse and DTC-SpMM on cora, ogbn-arxiv, and reddit. Table 4 presents the results. DTC-SpMM relies on a global row-column reordering over the entire sparse matrix, which scans and permutes all rows and columns and thus incurs substantial preprocessing cost. In contrast, the lower overhead of NeutronSparse stems from two design choices. First, heterogeneous workload partitioning identifies and assigns extremely sparse rows and columns (with very few nonzeros) to the AIV stream in a single pass, completely bypassing reordering for this portion of the matrix. Second, for the remaining AIC workload, reordering is confined to two lightweight stages: a coarse cluster-level grouping followed by row-only reordering within each cluster, rather than a full global row-column permutation. As a result, NeutronSparse consistently incurs significantly lower preprocessing overhead than DTC-SpMM, reducing preprocessing cost by 4.25× up to 156.92×, with the advantage becoming more pronounced on larger datasets where global reordering grows increasingly expensive.

8.7 Power Consumption Analysis

To evaluate energy efficiency, we record the power draw (in watts) every 100 milliseconds over a 10-second time window during SpMM. On Ascend, power draw is collected using `npu-smi`, while

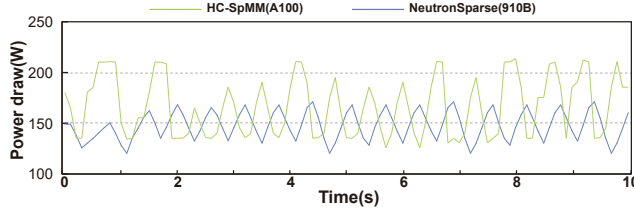


Fig. 24. Power draw (in watts) sampled every 100ms over a 10-second training window (Dataset: reddit).

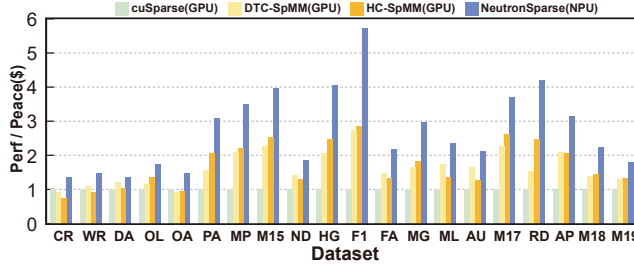


Fig. 25. Cost-normalized SpMM throughput using hourly cloud rental prices.

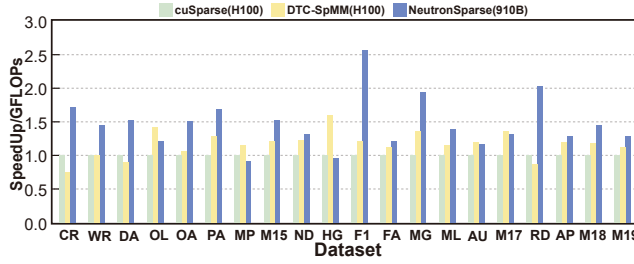


Fig. 26. TFLOPS-normalized SpMM performance comparison on Ascend 910B and NVIDIA H100.

on GPUs, `nvidia-smi` is used. Figure 24 shows the results. NeutronSparse, running on the Ascend 910B, achieves the lower power consumption, with an average power draw of 147.7W and a peak of 171.2W. In contrast, HC-SpMM incurs higher consumption, with an average power draw of 166.31W and a peak of 213.7W. The consistent gap across platforms indicates that NeutronSparse achieves competitive performance and resource efficiency with lower power consumption.

8.8 Cost Efficiency Analysis

We evaluate cost efficiency as throughput normalized by hourly rental price. To reduce potential bias from pricing variability, we collect rental prices from multiple mainstream cloud platforms in the Asia-Pacific region, including Huawei Cloud, AutoDL, Parallel Technology, Microsoft Azure, AWS EC2, and Google Cloud, and use the averaged price for each device type. Figure 25 reports the normalized results. After price normalization, NeutronSparse consistently outperforms GPU-based baselines in performance per dollar. On average, NeutronSparse achieves 2.47 $\times$, 1.49 $\times$, and 1.43 $\times$ higher cost efficiency than cuSPARSE, DTC-SpMM, and HC-SpMM, respectively. These results indicate that the advantage mainly comes from improved execution efficiency rather than pricing differences, suggesting that NPU-aware optimization can translate hardware capability into tangible cost benefits.

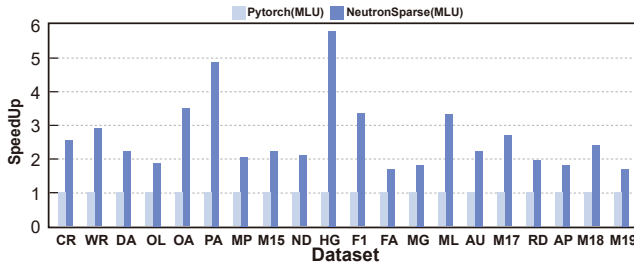


Fig. 27. SpMM performance on Cambricon MLU across all 20 datasets.

8.9 Comparison with a Modern GPU

We further evaluate NeutronSparse against a more recent GPU platform, NVIDIA H100, from a normalized-efficiency perspective. Ascend 910B provides 280 TFLOPS of peak compute throughput and 1.8 TB/s memory bandwidth, whereas H100 provides 1600 TFLOPS and 2.0 TB/s. Given this large gap in raw compute capability, a direct throughput comparison would be less fair. We therefore normalize performance by peak compute throughput (TFLOPS) and compare the resulting efficiency across platforms. As shown in Fig. 26, NeutronSparse on Ascend 910B achieves higher normalized efficiency than both cuSPARSE and DTC-SpMM on H100 for most datasets, with average improvements of 1.64 $\times$ and 1.55 $\times$, respectively.

8.10 Portability to Another NPU Backend

We port NeutronSparse to Cambricon MLU and compare it against the SpMM implementation provided by PyTorch on the same platform. The port preserves the core ideas of NeutronSparse by adapting heterogeneous workload partitioning and tile-aware scheduling to the execution primitives and software stack of MLU. Compared with Ascend 910B, Cambricon MLU also adopts a heterogeneous design with separate vector and matrix processing engines, but differs in its execution abstraction and memory hierarchy, where tensor operations rely more on compiler-managed tiling and shared on-chip buffers, leading to tighter coupling between engines and requiring additional backend-specific memory management and scheduling support for coordinated execution. As shown in Fig. 27, we evaluate all 20 datasets and observe consistent improvements across the full set: NeutronSparse achieves speedups ranging from about 1.71 $\times$ to 5.83 $\times$, with the largest gain on HG and clear improvements on most datasets. These results suggest that, although backend-specific engineering is still required, the main design principles of NeutronSparse are not limited to Ascend and can also deliver substantial benefits on another NPU family.

9 Related Work

SpMM Framework. Many works have sought to accelerate SpMM workloads arising in GNNs [29, 36, 46, 49, 50], graph analytics [29, 49, 60], and sparse attentions [20, 35]. Most GPU-based approaches utilizing a single compute unit within an SM (CUDA cores or Tensor Cores) [9, 10, 14, 16, 19, 21, 48, 51, 57]. For example, ASpT [14] employs column-block reordering to form dense tiles and improve data locality; RoDe [33] applies row decomposition to normalize long and short rows for better load balance; and Acc-SpMM [54] leverages hardware–software co-design to optimize the wmma API for sparse workloads. Recent studies have explored heterogeneous cooperation between CUDA cores and Tensor Cores. For example, HC-SpMM [21] and Libra [38] investigate hybrid execution modes by leveraging both CUDA cores and Tensor Cores on GPUs.

Research on Ascend NPUs. Research on Ascend NPUs is gradually gaining traction in the systems and architecture community [22, 56, 59]. MindSporeGL [59] provides a GNN training framework on

Table 5. Common architectural features leveraged by NeutronSparse across NPU families.

NPU	Heterogeneous Compute Engines	Tile-Based Execution
Ascend 910B [22]	✓ (AIC + AIV)	✓
TPU v4 [17]	✓ (MXU (Systolic Array) + vector/scalar units)	✓
Gaudi2 [28]	✓ (MME (Matrix Math Engine) + vector cores)	✓
MLU370 [24]	✓ (MMU (Matrix Multiply Unit) + vector units)	✓

MindSpore; NeutronAscend [4] proposes a heterogeneous scheduling framework that dispatches light and heavy workloads across units; and Zhou *et al.* present an automatic operator compilation technique alongside a componentized Roofline model for operator performance analysis [55, 56]. XY-Serve [39] proposed an end-to-end LLM service system that efficiently adapts to hardware by decomposing, rearranging, and abstracting dynamic workloads.

10 Future Work

Extension to Other NPUs. Although NeutronSparse is implemented on Ascend, its high-level design is not fully Ascend-specific. It is grounded in two architectural properties that are increasingly common across modern NPUs: **(1) matrix-centric heterogeneous engines** and **(2) tile-based execution**. As summarized in Table 5, mainstream NPUs (e.g., TPU [17], Gaudi [28], and MLU [24]) integrate dedicated matrix engines together with vector/scalar units, and expose fixed-shape tiles as the basic execution granularity. Under sparse workloads, this organization makes performance hinge on (i) coordinating heterogeneous engines to avoid idle resources and (ii) mitigating tile-level inefficiency caused by irregular sparsity. Accordingly, the high-level ideas of heterogeneous workload partitioning and tile-aware scheduling can generalize to other NPUs. In contrast, AIV-AIC coordinated pipelining is more Ascend-specific, since it depends on the decoupled AIC/AIV design. On other NPUs, such as Cambricon MLU and Google TPU, heterogeneous engines may still share and contend for local buffers and on-chip memory resources, so coordinated execution requires additional backend-specific memory management and scheduling support.

Learning-based Initialization. Our current design combines a lightweight initial split with online migration, but the optimal AIV/AIC ratio can still be affected by dynamic factors such as irregular sparsity, memory interference, and input-dependent behavior. A more accurate initial ratio could further reduce the warm-up period, especially for short-running workloads where online adjustment has less time to amortize. A promising direction for future work is learning-based initialization, which predicts a better initial split from matrix features and runtime signals.

11 Conclusion

This paper shows that efficient SpMM on Ascend NPUs requires NPU-specific optimization rather than directly porting GPU designs. We develop NeutronSparse, which addresses two bottlenecks: (i) balancing heterogeneous engines (AIC/AIV) via architecture-aware partitioning and online tile migration, and (ii) mitigating tile-level sparsity inefficiency via locality-aware tile orchestrating (global-local reordering and hierarchical tile reuse). Experiments on Ascend 910B demonstrate substantial speedups over NPU baselines and competitive performance against GPU libraries.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62461146205, U2241212, 625B2041), and the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501).

References

- [1] Khalid Ahmad, Cris Cecka, Michael Garland, and Mary Hall. 2024. Exploring Data Layout for Sparse Tensor Times Dense Matrix on GPUs. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–20.
- [2] Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, and Ge Yu. 2024. NeutronOrch: Rethinking Sample-based GNN Training under CPU-GPU Heterogeneous Environments. *Proc. VLDB Endow.* 17, 8 (2024), 1995–2008.
- [3] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proceedings of the VLDB Endowment* 18, 2 (2024), 173–186.
- [4] Xin Ai, Bing Zhang, Qiange Wang, Yanfeng Zhang, Hao Yuan, Shufeng Gong, and Ge Yu. 2025. NeutronAscend: Optimizing GNN Training with Ascend AI Processors. *ACM Transactions on Architecture and Code Optimization* (2025).
- [5] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 22–31.
- [6] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
- [7] Aritra Dhar, Clément Thorens, Lara Magdalena Lazier, and Lukas Cavigelli. 2024. Ascend-CC: Confidential Computing on Heterogeneous NPU for Emerging Generative AI Workloads. <https://arxiv.org/abs/2407.11888>.
- [8] I. S. Duff. 1977. A Survey of Sparse Matrix Research. *Proc. IEEE* 65, 4 (1977), 500–535.
- [9] 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 (ASPLOS)*, Volume 3. 253–267.
- [10] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. <https://arxiv.org/abs/2006.10901>.
- [11] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
- [12] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU–PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Volume 3. 722–737.
- [13] Khoa Ho, Hui Zhao, Adwait Jog, and Saraju Mohanty. 2022. Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 223–228.
- [14] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 300–314.
- [15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.
- [16] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix–Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
- [17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmamghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24–28, 2017*. ACM, 1–12.
- [18] Hyeoung Ko, Suyeon Lee, Yoonseo Park, and Anna Choi. 2022. A Survey of Recommendation Systems: Recommendation Models, Techniques, and Application Fields. *Electronics* 11, 1 (2022), 141.
- [19] Sureyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayappan. 2020. Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

- Networking, Storage and Analysis*. 1–14.
- [20] Shigang Li, Kazuki Osawa, and Torsten Hoefer. 2022. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.
 - [21] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Yunjun Gao, and Yaofeng Tu. 2025. HC-SpMM: Accelerating Sparse Matrix-Matrix Multiplication for Graphs with Hybrid GPU Cores. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 501–514.
 - [22] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 789–801.
 - [23] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. IEEE, 1–44.
 - [24] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 393–405.
 - [25] Shengbai Luo, Bo Wang, Yihao Shi, Xueyi Zhang, Qingshan Xue, and Sheng Ma. 2024. Sparm: A Sparse Matrix Multiplication Accelerator Supporting Multiple Dataflows. In *IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 122–130.
 - [26] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 43–52.
 - [27] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the construction of internet portals with machine learning. *Information Retrieval* 3, 2 (2000), 127–163.
 - [28] Eitan Medina and Eran Dagan. 2020. Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor. *IEEE Micro* 40, 2 (2020), 17–24.
 - [29] Nikolai Merkel, Pierre Toussing, Ruben Mayer, and Hans-Arno Jacobsen. 2024. Can Graph Reordering Speed Up Graph Neural Network Training? An Experimental Study. *Proceedings of the VLDB Endowment* 18, 2 (2024), 293–307.
 - [30] Salli Moustafa. 2023. Accelerating sparse matrix-matrix multiplication with the Ascend AI core. In *Proc. Workshop Accelerated Mach. Learn.* 33–41.
 - [31] NVIDIA. 2023. CUDA Sparse Matrix Library. <https://developer.nvidia.com/cusparse>. Accessed 2025-05-09.
 - [32] NVIDIA. 2023. Dense Linear Algebra on GPUs. <https://developer.nvidia.com/cublas>. Accessed 2025-05-09.
 - [33] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. 2024. RoDe: A Row Decomposition-Based Approach for Sparse Matrix Multiplication on GPUs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 377–389.
 - [34] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph LLM Neural Networks. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.
 - [35] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2021. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics* 9 (2021), 53–68.
 - [36] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2022. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. <https://arxiv.org/abs/2204.13103>.
 - [37] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. 2025. Flashsparse: Minimizing computation redundancy for fast sparse matrix multiplications on tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 312–325.
 - [38] Jinliang Shi, Shigang Li, Youxuan Xu, Xueying Wang, Rongtian Fu, Zhi Ma, and Tong Wu. 2025. Libra: Synergizing CUDA and Tensor Cores for High-Performance Sparse Matrix Multiplication. *arXiv preprint arXiv:2506.22714* (2025).
 - [39] Mingcong Song, Xinru Tang, Fengfan Hou, Jing Li, Wei Wei, Yipeng Ma, Runqiu Xiao, Hongjie Si, Dingcheng Jiang, Shouyi Yin, et al. 2026. XY-Serve: End-to-End Versatile Production Serving for Dynamic LLM Workloads. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 314–329.
 - [40] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonese, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780.
 - [41] W. Patrick Walters and Regina Barzilay. 2020. Applications of Deep Learning in Molecule Generation and Molecular Property Prediction. *Accounts of Chemical Research* 54, 2 (2020), 263–270.
 - [42] Xinchun Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23.

- [43] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4 (2023), 246:1–246:27.
- [44] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *International Conference on Management of Data, Philadelphia, SIGMOD’22, PA, USA*. ACM, 1301–1315.
- [45] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 338–353.
- [46] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2021. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2021).
- [47] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2022. A Pattern-Based SpGEMM Library for Multi-Core and Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 159–175.
- [48] Yi Xu, Licheng Yu, Hongteng Xu, Hao Zhang, and Truong Nguyen. 2015. Vector Sparse Representation of Color Image Using Quaternion Matrix Analysis. *IEEE Transactions on Image Processing* 24, 4 (2015), 1315–1329.
- [49] Jiale Yan, Hiroaki Ito, Angel Lopez Garcia-Arias, Yasuyuki Okoshi, Hikari Otsuka, Kazushi Kawamura, Thiem Van Chu, and Masato Motomura. 2023. Multicoated and Folded Graph Neural Networks with Strong Lottery Tickets. Manuscript.
- [50] Jianrong Yan, Wenbin Jiang, Dongao He, Suyang Wen, Yang Li, Hai Jin, and Zhiyuan Shao. 2024. RT-GNN: Accelerating Sparse Graph Neural Networks by Tensor-CUDA Kernel Fusion. *ACM Transactions on Architecture and Code Optimization* (2024), 3702001.
- [51] Carl Yang, Aydin Buluc, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. <https://arxiv.org/abs/1803.08601>.
- [52] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (2020), 106848.
- [53] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. 2021. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In *IEEE International Conference on Computer Design (ICCD)*. 290–298.
- [54] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, Yan Zeng, Yangang Wang, and Xuebin Chi. 2025. Acc-SpMM: Accelerating General-Purpose Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 326–338.
- [55] Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu, Yanlin Liu, Chunsheng Li, Ziyang Zhang, Yaoyuan Wang, Bin Zhou, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Squeezing Operator Performance Potential for the Ascend Architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 1156–1171.
- [56] Yuhang Zhou, Zibo Wang, Zhibin Wang, Ruyi Zhang, Chen Tian, Xiaoliang Wang, Wanchun Dou, Guihai Chen, Bingqiang Wang, Yonghong Tian, Yan Zhang, Hui Wang, Fuchun Wei, Boquan Sun, Jingyi Zhang, Bin She, Teng Su, Yifan Yao, Chunsheng Li, Ziyang Zhang, Yaoyuan Wang, Bin Zhou, and Guyue Liu. 2025. Accelerating Model Training on Ascend Chips: An Industrial System for Profiling, Analysis and Optimization. In *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, Deniz Altinbükten and Ryan Stutsman (Eds.). USENIX Association, 1387–1408.
- [57] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 359–371.
- [58] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.
- [59] Zeling Zhu, Bangchuan Wang, Chuying Yang, Rui Zhu, Mingyao Zhou, and Nenggan Zheng. 2023. Performance Evaluation of MindSpore and PyTorch Based on Ascend NPU. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 1826–1832.
- [60] Xiaohan Zou. 2020. A Survey on Application of Knowledge Graph. In *Journal of Physics: Conference Series*, Vol. 1487. 012016.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026