Mentor: A Memory-Efficient Sparse-dense Matrix Multiplication Accelerator Based on Column-Wise Product

XIAOBO LU, School of Computer Science and Technology, National University of Defense Technology, Changsha, China
JIANBIN FANG, School of Computer Science and Technology, National University of Defense Technology, Changsha, China
LIN PENG, School of Computer Science and Technology, National University of Defense Technology, Changsha, China
CHUN HUANG, School of Computer Science and Technology, National University of Defense Technology, Changsha, China
ZIDONG DU, Institute Of Computing Technology, Chinese Academy of Sciences, Beijing, China
YONGWEI ZHAO, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
ZHENG WANG, Northwest University, Xi'an, China

Sparse-dense matrix multiplication (SpMM) is the performance bottleneck of many high-performance and deep-learning applications, making it attractive to design specialized SpMM hardware accelerators. Unfortunately, existing hardware solutions do not take full advantage of data reuse opportunities of the input and output matrices or suffer from irregular memory access patterns. Their strategies increase the off-chip memory traffic and bandwidth pressure, leaving much room for improvement. We present MENTOR, a new approach to designing SpMM accelerators. Our key insight is that column-wise dataflow, while rarely exploited in prior works, can address these issues in SpMM computations. MENTOR is a software-hardware co-design approach for leveraging column-wise dataflow to improve data reuse and regular memory accesses of SpMM. On the software level, MENTOR incorporates a novel streaming construction scheme to preprocess the input matrix for enabling a streaming access pattern. On the hardware level, it employs a fully pipelined design to unlock the potential of column-wise dataflow further. The design of MENTOR is underpinned by a carefully designed analytical model to find the tradeoff between performance and hardware resources. We have implemented an FPGA prototype of MENTOR. Experimental results show that MENTOR achieves speedup by geomean 2.05× (up to 3.98×), reduces the memory traffic by geomean 2.92× (up to 4.93×), and improves bandwidth utilization by geomean 1.38× (up to 2.89×), compared with the state-of-the-art hardware solutions.

X. Lu and J. Fang contributed equally to the paper.
This work is supported by the National Key Research and Development Program of China under Grant No. 2023YFB3001503.
Authors' Contact Information: Xiaobo Lu, School of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China; e-mail: xb.lu@nudt.edu.cn; Jianbin Fang, School of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China; e-mail: j.fang@nudt.edu.cn; Lin Peng (Co-corresponding author), School of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China; e-mail: penglin@nudt.edu.cn; Chun Huang (Co-corresponding author), School of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China; e-mail: chunhuang@nudt.edu.cn; Zidong Du, Institute Of Computing Technology, Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: duzidong@ict.ac.cn; Yongwei Zhao, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, Beijing, China; e-mail: zhaoyongwei@ict.ac.cn; Zheng Wang, Northwest University, Xi'an, Shaanxi, China; e-mail: zwang@nwu.edu.cn.

ACM 1544-3973/2024/11-ART79
https://doi.org/10.1145/3688612

CCS Concepts: • **Hardware → Hardware accelerators**; • **Computer systems organization → Architectures;**

Additional Key Words and Phrases: SpMM, sparse linear algebra

ACM Reference Format:
Xiaobo Lu, Jianbin Fang, Lin Peng, Chun Huang, Zidong Du, Yongwei Zhao, and Zheng Wang. 2024. Mentor: A Memory-Efficient Sparse-dense Matrix Multiplication Accelerator Based on Column-Wise Product. *ACM Trans. Arch. Code Optim.* 21, 4, Article 79 (November 2024), 25 pages. https://doi.org/10.1145/3688612

1 Introduction

Sparse-dense matrix multiplication (**SpMM**) involves the multiplication of a sparse matrix, **A**, with a dense matrix, **B**. This operation is commonly seen in traditional scientific applications [23, 31, 38, 44–46, 49, 52] and emerging machine learning workloads [17, 24, 39, 50, 56, 61]. There is an extensive body of research for accelerating the SpMM kernel on CPUs [11, 40], GPUs [1, 5, 12, 20, 37, 54] and purposed-built hardware accelerators [13, 14, 25, 43, 53]. Since the input matrices of real-life SpMM kernels are often too large to be kept in on-chip buffers on hardware accelerators with MBs of **static RAM** (**SRAM**), memory optimization is vital for SpMM performance.

Prior work for optimizing SpMM typically follows three types of dataflows for matrix computations: *inner*, *outer*, and *row-wise* products [2, 3, 18, 33, 35, 42, 59, 62], as depicted in Figure 1. However, these approaches are inefficient at optimizing off-chip memory access. Specifically, the inner product computes an element of **C** by performing a dot product, summing a row of **A** with a column of **B**. In contrast, the outer product dataflow computes an outer product between a column of **A** and a row of **B** to generate a partial matrix, which is then merged to form **C**. Unfortunately, neither approach effectively exploits the data reuse of both input and output matrices. When processing large matrices, the inner product method results in excessive memory traffic by iterating over the dense matrix **B** multiple passes. In contrast, the outer product method generates a large number of partial matrices, leading to suboptimal performance.

More recent work exploits the row-wise product for matrix multiplications [26, 42, 59]. This approach multiplies each non-zero in a row of **A** with the corresponding row of **B** to form a final row of **C**, where the row in **B** is determined by the column index of the non-zero in **A**. While representing a step forward over inner and outer products, row-wise based SpMM accelerators can suffer from poor data locality since the non-zeros used to locate rows in **B** are irregularly distributed in the sparse matrix **A**. This often leads to poor memory bandwidth utilization.

We present Mentor, a software-hardware co-design approach to accelerate SpMM, built upon the column-wise dataflow. As shown in Figure 1(d), this approach multiplies each element in **B** with the corresponding column of **A** to generate a partial column of the output matrix. Our systematic analysis of the memory access pattern of different dataflows for SpMM suggests that the column-wise approach can effectively overcome the drawbacks of inefficient off-chip memory accesses. While some prior works [13, 57] have leveraged this approach for consecutive SpMM operations, they fail to take full advantage of the column-wise approach on individual SpMM computation.

Mentor is designed to take advantage of the column-wise approach by providing a streaming design at the software and the hardware layers. At the software level, we identify optimization opportunities in conventional memory access patterns and propose a pre-processing technique to reconstruct the input matrices. This enables streaming memory accesses for Mentor to improve memory access efficiency when accessing the matrices. Specifically, for dense matrices, this scheme adopts row-major storage rather than column-major used in other column-wise-related works to harvest the data locality and permits the hardware to overlap the computation with memory

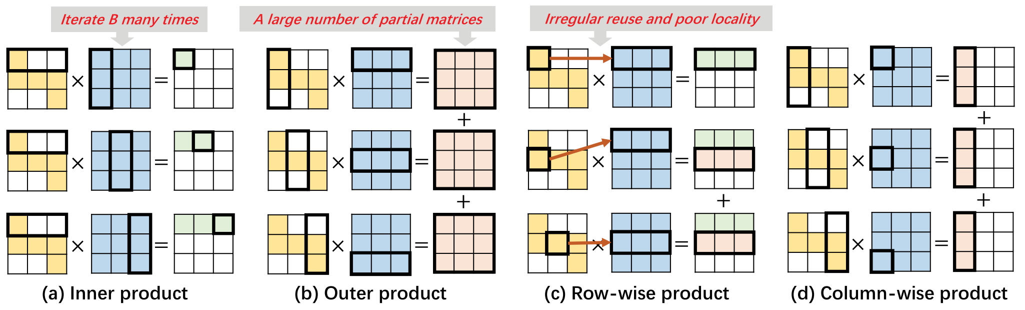

Fig. 1. Four different SpMM approaches.

accesses by an aggressive optimization. For the sparse matrices, the scheme incorporates a novel compressed format, enabling streaming access to sparse matrices and addressing the **read-after-write** (**RAW**) conflict by introducing control elements to the conventional compressed format. At the hardware level, Mentor employs a fully pipelined design to process data in a streaming fashion for enhancing throughput and bandwidth utilization. This includes an efficient memory interface with low hardware complexity and a double-buffered on-chip structure to avoid pipeline stall. The hardware design of Mentor is supported by a performance model to derive hardware parameters (e.g., *#PEs*) for given off-chip memory bandwidths and sparse matrices.

We have implemented a hardware prototype of Mentor on an Xilinx FPGA. We evaluate Mentor across a wide range of input matrices from real-world applications with various sparse patterns and structures. We compare Mentor against four SpMM accelerator baselines based on inner [35], outer [18], row-wise [41, 59], and column-wise [13] dataflows. Compared with the baselines, Mentor achieves a 1.18×–3.98× speedup, with 1.23×–4.93× reduction in the off-chip memory traffic.

This article makes the following contributions:

— It provides a quantitative analysis to demonstrate the advantages of the column-wise approach on SpMM acceleration (Section 2);
— It demonstrates how the column-wise dataflow can be employed at the software level (Section 3) and at the hardware level (Section 4.1) to accelerate SpMM;
— It presents a performance model to help explore the hardware design space for leveraging the column-wise dataflow design for SpMM (Section 4.2).

2 Background and Motivation

2.1 Sparse-Dense Matrix Multiplication

Given a sparse matrix **A** of $M \times K$, a dense matrix **B** of $K \times N$ and a dense matrix **C** of $M \times N$, SpMM performs the following computation,

$$\mathbf{C} = \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}, \tag{1}$$

where $\mathbf{A} \cdot \mathbf{B}$ is typically the performance bottleneck and thus is the main focus of this work.

Since matrix **A** is often sparse with many zero elements, it is typically stored in a sparse matrix storage format such as the **compressed sparse row** (**CSR**) [41, 59] or the **compressed sparse column** (**CSC**) [16] formats. In contrast, **B** is a dense matrix where the non-zeros are stored in row-major or column-major order. As in previous works [30, 59], we refer to each compressed row/column of the sparse matrix **A** as a *fiber* and use *row/column* exclusively when referring to the dense matrix **B**.

Table 1. Memory Access Analysis of Dataflows and their Parallelization Versions, where (M) and (N) Denote the Stationary Dimension. $\underline{M}$ and $\underline{N}$ are the Parallelized Dimensions. Here, We Only Discuss the Parallelized Dimensions, Which are Beneficial to Optimize Memory Access

Dataflow	**Order**	**Input (w/o. P)**	**Input (with P)**	**Partial**	**Reuse (with P)**	**Locality**	**Band. Req.**
Inner (M)	$\underline{\text{M}}$-N-K	$nnz + M \cdot K \cdot N$	$nnz + \frac{M \cdot K \cdot N}{P}$	–	$< \frac{nnz \cdot P}{M \cdot K}$	Moderate	Low
Inner (N)	$\underline{\text{N}}$-M-K	$(nnz + K) \cdot N$	$nnz \cdot \frac{N}{P} + K \cdot N$	–	$= \frac{1}{\frac{1}{P} + \frac{K}{nnz}}$	Moderate	Low
Outer (M)	K-M-N	$nnz + K \cdot N$	$nnz + K \cdot N$	$\geq 2 \cdot nnz \cdot N$	$< \frac{nnz}{2 \cdot nnz + K}$	Good	High
Outer (N)	K-N-M	$nnz + K \cdot N$	$nnz + K \cdot N$	$\geq 2 \cdot nnz \cdot N$	$< \frac{nnz}{2 \cdot nnz + K}$	Good	High
Row	$\underline{\text{M}}$-K-N	$nnz + nnz \cdot N$	$[nnz(1 + \frac{N}{P}),$ $nnz(1 + N)]$	–	$[\frac{1}{\frac{1}{N} + \frac{1}{P}}, \frac{N}{1+N}]$	Moderate	Low
Column	$\underline{\text{N}}$-K-M	$\leq (nnz + K) \cdot N$	$nnz \cdot \frac{N}{P} + K \cdot N$	–	$= \frac{1}{\frac{1}{P} + \frac{K}{nnz}}$	Good	Low

2.2 Memory Access Analysis

Our work is motivated by the observation that existing SpMM optimizations built upon the inner, outer, and row-wise products have drawbacks of inefficient off-chip memory accesses. According to Reference [30], we categorize the dataflows into 6 variants, where (M) and (N) denotes the stationary dimension[1] in the dataflow. We follow the memory modeling approach for the **General Sparse Matrix-Matrix Multiplication (SpGEMM)** [42, 48] to analyze the memory access of these implementations. Table 1 quantifies the amount of memory traffic[2] required for moving input data and storing partial results, the data reuse,[3] spatial locality and bandwidth requirements for six computation dataflows with parallelization, in terms of the number of non-zeros (nnz) in **A** and the matrix sizes (M, K, and N) of matrices **A** and **B**. The number of MAC operations for each SpMM approach is $nnz \cdot N$. We also discuss the data locality and bandwidth requirements of the parallelized dataflows in Table 1.

2.2.1 *Inner Product.*

This dataflow iterates the fibers (i.e., rows/columns) of sparse matrix **A** (which is typically stored in the CSR format) and the columns of **B** to perform vector dot product to obtain a final element:

$$\mathbf{C}[i,j] = \sum_{k=0}^{K} \mathbf{A}[i,k] \cdot \mathbf{B}[k,j] \tag{2}$$

As shown in Table 1, this dataflow has two variants: Inner (M) for M-stationary and Inner (N) for N-stationary. To calculate the entire final matrix **C**, both require negligible off-chip memory accesses to partial sums produced by dot products, which are often small enough to be stored on-chip. But they require a large amount of off-chip memory traffic to access entire **B** M times or entire **A** N times, and thus the memory traffic is $M \cdot K \cdot N$ for Inner (M) and $nnz \cdot N$ for Inner (N). As a result, the data reuse of Inner (M) is $\frac{nnz \cdot N}{nnz + M \cdot K \cdot N} \approx \frac{nnz}{M \cdot K}$. This approximation holds because $nnz \ll M \cdot K \cdot N$. And the data reuse of Inner (N) is $\frac{nnz \cdot N}{(nnz+K) \cdot N} = \frac{nnz}{nnz+K}$.

2.2.2 *Outer Product.*

This dataflow traverses fibers of **A** (which is typically stored in the CSC format) and rows of **B**, performs the outer product ($\otimes$) to produce a partial matrix (multiply stage),

[1] A *stationary* dimension is a matrix dimension that remains in place or "stationary" in the processing element's local memory while other dimensions are moved or streamed through the processing elements.
[2] In this article, the *memory traffic* is defined as the number of matrix elements loaded/stored from/into memory [42].
[3] The *data reuse* of matrices is defined as the number of multiply-accumulate (MAC) operations divided by the number of matrix elements moved from/into memory (i.e., memory traffic), indicating the number of operations performed per element accessed.

$\mathbf{C}^{'}$, which will then be merged to form the output matrix **C** during a merging phase:

$$\mathbf{C}_k^{'} = \mathbf{A}[:, k] \otimes \mathbf{B}[k, :]. \tag{3}$$

This dataflow has two variants: Outer(M) and Outer(N). Both dataflow variants traverse **A** and **B** once but suffer from large memory traffic requirements for partial matrices and poor output reuse. Thus, the data reuse is less than $\frac{nnz \cdot N}{nnz + K \cdot N + 2 \cdot nnz \cdot N}$ for both Outer(M) and Outer(N). For the typical SpMM computations in **deep neural networks** (**DNNs**), matrix **B** often works as a weight matrix, where N is the hidden layer dimension ranging from 10 to 1000. Under such settings, we can estimate that $nnz \ll nnz \cdot N$, and the data reuse is less than $\frac{nnz}{2 \cdot nnz + K}$ (Table 1).

2.2.3 *Row-Wise Product.* This dataflow is a variant of Gustavson's algorithm, which performs intersections, i.e., scalar-vector multiplication between the scalars from fibers of **A** in CSR and corresponding rows from **B** determined by the non-zero coordinates in the fibers, and produces a partial row of the final row of **C**,

$$\begin{aligned} \mathbf{C}_{cid}^{'}[i, :] &= \mathbf{A}[i, cid] \cdot \mathbf{B}[cid, :] \\ \mathbf{C}[i, :] &= \sum_{j=0}^{k} \mathbf{C}_j^{'}[i, :] \end{aligned} \quad . \tag{4}$$

Each intersection takes a scalar and a vector of N elements, i.e., an input traffic of $nnz + nnz \cdot N$. Since the partial results sizing N can be held on-chip, the data reuse of SpMM is $\frac{nnz \cdot N}{nnz + KN} \approx \frac{N}{1+N} \approx 1$.

2.2.4 *Column-Wise Product.* This dataflow is also a variant of the Gustavson algorithms, which performs intersections between scalars from **B** columns and fibers of **A** in the CSC format, and obtains a partial column of **C**,

$$\begin{aligned} \mathbf{C}_{rid}^{'}[:, i] &= \mathbf{A}[:, rid] \cdot \mathbf{B}[rid, i] \\ \mathbf{C}[:, i] &= \sum_{j=0}^{k} \mathbf{C}_j^{'}[:, i] \end{aligned} \quad . \tag{5}$$

Each intersection consumes a scalar from **B** and a fiber from **A**, leading to an input traffic of $nnz \cdot N + K \cdot N$. Note that the non-zeros in **B** is smaller than $N \cdot K$. The partial results sizing K can also be held on-chip; thus, the data reuse for column-wise is larger than $\frac{nnz \cdot N}{(K + nnz) \cdot N} = \frac{nnz}{nnz + K}$. Considering that $\frac{nnz}{K}$ ranges from 10 to 100 [42], the data reuse will also approach 1 (Table 1).

2.3 Advantages of Column-Wise

Given the dense nature of matrix **B** in SpMM, we find that the column-wise dataflow can significantly enhance memory efficiency, which is crucial for the memory-bound SpMM. We now consider parallelization to discuss the drawbacks of other methods regarding memory access. We use $\underline{M}$ or $\underline{N}$ in Table 1 to indicate that the dataflow is parallelized along the M or N dimension with a parallelism degree P. Figure 2 compares the two variants of the Gustavson algorithm, i.e., the row-wise vs. column-wise approach.

❶ **The column-wise approach demonstrates better data reuse.** With parallelization, the column-wise approach reduces iterations for accessing **A** from N to $\frac{N}{P}$ and improves data reuse to $\frac{1}{\frac{1}{P} + \frac{K}{nnz}}$, and Inner(M) can improve reuse to $\frac{nnz \cdot N \cdot P}{nnz \cdot P + M \cdot K \cdot N} \approx \frac{nnz}{M \cdot K} \cdot P$. Thus, the reuse ratio of the Inner(M) to column-wise approach is $\frac{nnz}{M \cdot K} \cdot P \cdot (\frac{1}{P} + \frac{K}{nnz}) = \frac{nnz}{M \cdot K} + \frac{1}{M} \ll 1$. In other words, the column-wise approach exhibits better data reuse compared with Inner(M). The memory access of the outer dataflow does not benefit from parallelization, making the column-wise approach more suitable. The parallelized row-wise approach still suffers from irregular data reuse. Figure 2(a) shows that

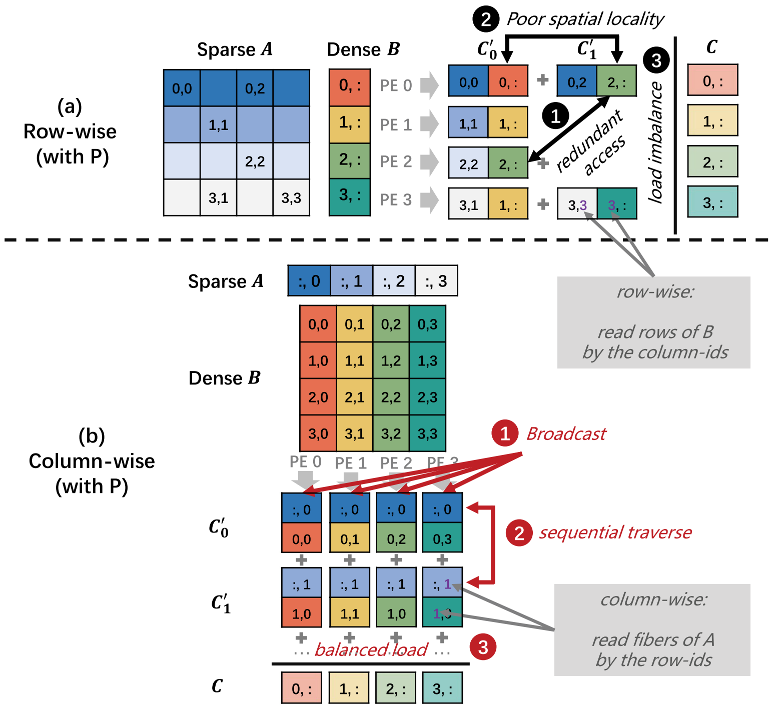

Fig. 2. Comparing parallelized row-wise and column-wise dataflows.

B$[2,:]$ is accessed twice by **A**$[2,2]$ and **A**$[0,2]$. Such irregular and challenging-to-capture reuses significantly increase memory traffic. In the worst case, when there are no identical elements in the parallel fibers, the data reuse of this approach is only $\frac{1}{1+N}$. Figure 2(b) shows that the column-wise approach can benefit from a regular memory access pattern, where **A**$[:,0]$ is read once and subsequently broadcasted to multiple **B** scalars (**B**$[0,0]$, **B**$[0,1]$, **B**$[0,2]$ and **B**$[0,3]$), which maximizes reuse of **A**, thereby reducing input traffic from $(nnz+K)\cdot N$ to $(\frac{nnz}{P}+K)\cdot N$ and increasing data reuse to $\frac{1}{\frac{1}{P}+\frac{K}{nnz}}$.

❷ **The column-wise approach has better spatial locality.** The Inner(M) and Inner(N) dataflows can sequentially traverse **A** and **B**, but its spatial data locality is limited by the index matching between **A** fibers and **B** columns. The outer-product dataflow can avoid this problem and achieve good spatial locality. The row-wise dataflow performs intersections between scalars of **A** and vectors of **B** (**B**$[0,:]$ and **B**$[2,:]$ determined by **A**$[0,0]$ and **A**$[0,2]$ in Figure 2(a)), and the irregular distribution of non-zeros leads to poor locality of **B**. In contrast, the column-wise dataflow guarantees sequential access to the entire **A** by exploiting the continuity of row-ids of non-zeros in columns of **B**. The sequential access begins with **A**$[:,0]$, followed by **A**$[:,1]$ and so forth for subsequent columns as shown in Figure 2(b). This access pattern can significantly benefit memory access. Taking the DDR4 on the UltraScale FPGA as an example, the simple address increment pattern can reach an efficiency of up to 94% [21]. These findings highlight the significant performance impact of the column-wise approach with better spatial locality.

❸ **The column-wise approach requires low off-chip bandwidth.** Although the outer-product dataflow has good spatial locality, it is still limited by the non-parallelizable multiply and merge stages. The merge stage reduces a large number of partial elements and requires high bandwidth

Table 2. The Summary of Related Accelerators and Comparison in Terms of Dataflow, Data Reuse, Locality, Bandwidth Requirement, Load Balancing, and Streaming

Accelerator	**Kernel**	**Dataflow**	**Reuse**	**Locality**	**Band. Req.**	**Load Balance**	**Streaming**
SIGMA [35]	SpGEMM	Inner(M&N)	Poor	Good	Low	✗	✓
Spaghetti [18]	SpGEMM	Outer(M)	Moderate	Good	Low	✓	✓
GAMMA [59]	SpGEMM	Row-wise	Good	Enhanced	High	✗	✗
Sextans [41]	SpMM	Row-wise	Poor	Enhanced	High	✓	✓
HyGCN [53]	SpMM&GEMM	Row-wise&Inner(M)	Poor	Moderate	Low	✓	✗
AWB-GCN [13]	SpMM	Column-wise	Moderate	Good	Low	✓	✗
GCNAX [25]	SpMM	Outer(M)	Moderate	Good	High	✗	✗
Mentor	**SpMM**	**Column-wise**	**Good**	**Good**	**Low**	✓	✓

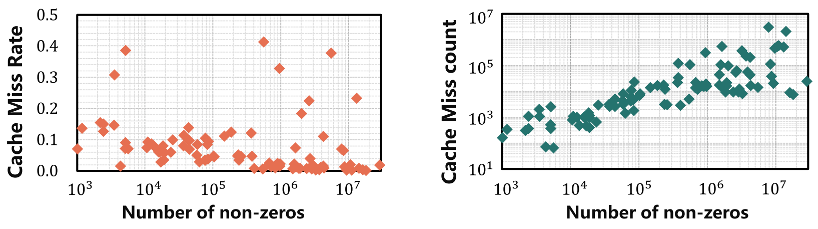

Fig. 3. FiberCache performance on 100 sparse matrices.

support to minimize overall computational latency. The inner products and Gustavson variants can hold partial results on-chip to reduce the bandwidth requirement.

We also identify that it is easier for the column-wise dataflow to achieve load balancing than the row-wise dataflow. Figure 2(a) shows that the computation load assigned to each processing engine (PE) depends on the distribution of non-zeros in **A** for the row-wise dataflow. In contrast, broadcasting ensures even loads among PEs for the column-wise dataflow. This advantage will simplify software or hardware design to balance computational efficiency.

2.4 Prior Accelerators

Table 2 presents a comparison of related accelerators, including specialized works for SpGEMMs (SIGMA, Spaghetti, GAMMA) and GCNs (HyGCN, AWB-GCN, and GCNAX). We consider GCNs because the two stages of a single GCN layer can be organized as two consecutive SpMMs. We discuss the inefficiencies in their dataflows and specific designs.

On the one hand, prior accelerators make critical tradeoffs to overcome the inefficiencies of their dataflows. SIGMA [35] utilizes a bitmap format (using binary to mark non-zero positions) to map sparse matrices to the MAC array and avoid index matching. Still, this format is space-inefficient on highly sparse matrices, leading to decreased input reuse. Spaghetti [18] is the state-of-the-art outer-product-based accelerator, which supports a streaming access pattern to improve output reuse and reduce bandwidth requirement. Unfortunately, it introduces significant overhead of sort-mergers, sacrificing input reuse. To address the issues of irregular reuse and poor data locality, GAMMA [59] employs cache-based on-chip structures to capture irregular reuse of **B**, reducing memory traffic and enhancing the data locality. However, the performance of the cache-based design heavily depends on the non-zero distribution of the sparse **A**. Figure 3 shows an emulation of cache behaviors using the same configuration as the FiberCache [59] (3MB, 16-way set-associative) on 100 sparse matrices. We see significant variations in cache performance, with a maximum miss rate exceeding 41% and a minimum of 0.08%. Therefore, this approach requires high bandwidth support to mitigate the impact of cache misses. Sextans [41] uses multi-level memory optimizations to buffer partitioned blocks of **B** on-chip. But it ruins input reuse of **A** and fails to overlap the memory access by computation. Thus, Sextans suffers from significant memory traffic and high bandwidth requirement. AWB-GCN [13] is designed for consecutive SpMMs, which leverages the column-wise approach and maps the computation of a final column to multiple parallel PEs. This strategy sacrifices the data reuse of matrix **A**.

On the other hand, the previous non-streaming accelerators suffer from not fully utilizing off-chip bandwidth and PEs. Streaming is a design that can receive, process, and emit data elements at a consistent rate in continuous clock cycles, ensuring efficient data flow [63]. This means that a streaming design has two main features: ❶ A streaming access pattern, which requires continuous transfer of off-chip data with low access latency [27, 41]); ❷ An optimized data path, which needs to achieve a consistent processing rate between the components in the architecture to ensure high **processing element** (**PE**) utilization [18, 19]. As a streaming example, Sextans adopts non-zero scheduling and multi-level memory optimization for efficient memory access. Despite being limited by the inherent disadvantages of the row-wise dataflow, Sextans provides an effective solution by streaming design to accelerate SpMM. However, previous non-streaming designs either do not support streaming access patterns or experience reduced PE utilization due to potential pipeline stalls. Specifically, GAMMA is constrained by the irregular access pattern determined by the distribution of non-zero elements within fibers, and cache misses can cause pipeline stalls in the PEs. HyGCN [53] uses fine-grained parallel execution but does not support a streaming access pattern due to graph partitioning and sparse elimination. Additionally, the tandem engine results in the underutilization of PEs due to workload imbalance between the two execution stages. AWB-GCN [13] maps the partitioned matrix **A** to multiple parallel PEs, preventing **A** from being streamed. Furthermore, its auto-tuning unit needs to rebalance the workload after the completion of each column, leading to pipeline stalls and limited parallelism. GCNAX [25] focuses on dataflow optimization about loop order and loop fusion but does not offer a streaming design.

In a nutshell, our Mentor utilizes a column-wise approach without compromising memory accesses. This is achieved by implementing streaming at the software level to enable streaming memory accesses (Section 4) and designing a fully-pipelined hardware architecture at the hardware level (Section 3) to adapt to the created data streams for near-optimal performance.

3 Streaming Construction

To achieve streaming memory accesses, we propose a novel preprocessing scheme to reconstruct the input dense and sparse matrices.

3.1 Streaming Construction for the Dense Matrix

Figure 4(a) shows the limitations of the conventional access pattern. We assume that each fiber of **A** contains two non-zeros, meaning that each scalar-vector intersection takes two clock cycles with pipelined **float point units** (**FPUs**). The matrix **B** is shown in transposed form and accessed by two 128-bit (4 floating-point elements) memory channels. During cycle 1, four elements are transferred to the PE-0 and PE-4 using channel-1 and channel-2, respectively. In the next cycle, the 2nd and 6th columns of **B** are traversed, presenting a poor spatial locality. Moreover, this pattern limits PE utilization because half of the PEs remain idle while the other half is overloaded during cycle 2. Increasing bandwidth can alleviate this issue but may lead to lower bandwidth utilization.

To this end, we propose two optimization strategies to reconstruct dense matrices to achieve more efficient memory accesses.

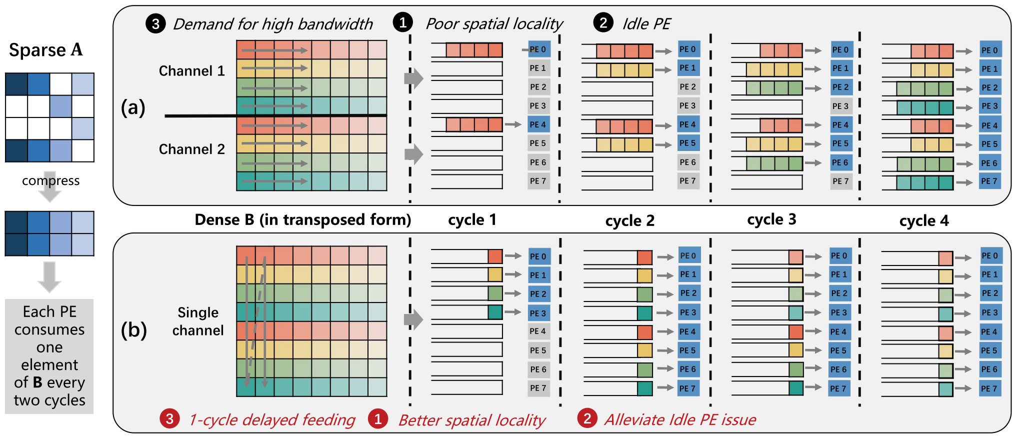
Sparse A
compress
Each PE consumes one element of B every two cycles
(a)
(b)
Channel 1
Channel 2
Single channel
3 Demand for high bandwidth
1 Poor spatial locality
2 Idle PE
Dense B (in transposed form)
cycle 1
cycle 2
cycle 3
cycle 4
3 1-cycle delayed feeding
1 Better spatial locality
2 Alleviate Idle PE issue

Fig. 4. Comparison of conventional and streaming access pattern in consecutive cycles.

Strategy 1: Row-major traversal and matrix partitioning: We first adopt the row-major storage for **B**, allowing the architecture to traverse and distribute elements to the PEs. We then partition each **B** row into segments with a length of #PE (the number of PEs) and store each block with a dimension of $K \times \#PE$ in consecutive address spaces. Doing so can ensure a better spatial locality when performing computations on the blocks of **B**. In Figure 4(b), during cycles 1 and 2, the elements in the first row are evenly distributed to all PEs with a single memory channel. This access pattern enables streaming access to blocks of **B** as all rows are sequentially traversed and alleviates the idle PE issue by feeding each PE with at most one element in one clock cycle. In implementation, we also adopt this strategy when writing the matrix **C**, which means **C** will be stored in the same form as **B**. This improves the spatial locality in the writing phase, and **C** can be directly applied as an operand in consecutive SpMMs. More details will be introduced in Section 4.1.4.

Strategy 2: c-cycle delayed feeding: In the column-wise approach, each element of **B** participates in a scalar-vector product $\mathbf{A}[:, rid] \cdot \mathbf{B}[rid, i]$. Thus, the minimum required cycle is equal to the number of non-zeros in $\mathbf{A}[:, rid]$ (assuming each PE employs one multiplier). Figure 4 shows that each scalar-vector product takes two cycles, enabling feeding one **B** element to each PE every two cycles while maintaining high PE utilization. Therefore, we introduce a delay parameter, c, and propose a c-cycle delayed feeding strategy to allow for feeding each PE with one $\mathbf{B}[rid, i]$ every $c + 1$ cycles. Figure 4(b) presents the 1-cycle delayed strategy, where each PE is fed with one element per two cycles. This strategy significantly reduces the bandwidth budget for accessing **B** and achieves high bandwidth utilization. Note that it is an aggressive optimization method because a larger c value enables a smaller bandwidth requirement but increases the risk of performance degradation. Further analysis will be detailed in Section 4.2.

3.2 Streaming Construction for the Sparse Matrix

CSC and CSR are commonly adopted for storing sparse matrices. Figure 5(a) shows that both formats use *idx* and *val* arrays to hold the positions and values of non-zeros, respectively, and use a pointer array, *ptr*, to indicate the start address of each column/row. We argue that the traditional processing dataflow has three issues. (1) It adopts burst transfer to access **A** fibers without delay, but this method is less efficient than streaming access. (2) It requires additional control logic to separate the **A** fibers and complete the computation of a **C** row; (3) It can incur RAW conflicts. Figure 5(a) presents a sparse **A** as an example. The first fiber will be multiplied with $\mathbf{B}[0, i]$ to

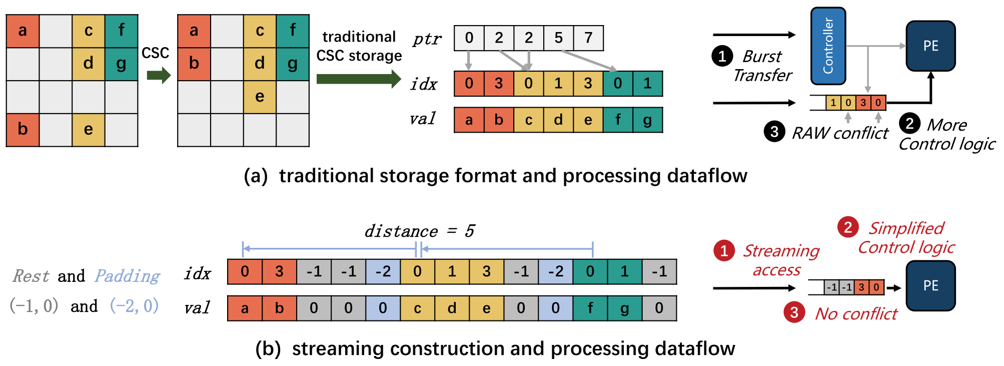

Fig. 5. CSC storage format and streaming construction for sparse matrices.

produce a partial column containing two non-zeros of $a \cdot \mathbf{B}[0, i]$ and $b \cdot \mathbf{B}[0, i]$. The two partial results will be accumulated according to their row-ids, i.e., 0 and 3, which are determined by a and b. Similarly, multiplying $\mathbf{B}[2, i]$ with $\mathbf{A}[:, 2]$ produces partial elements with row-ids of 0, 1, and 3. Since FPUs require multiple clock cycles for floating-point additions, two accumulation operations with the same row-id may incur an RAW conflict. Assuming a RAW dependence distance of d, the distance between two **A** non-zeros with the same row-id must not be less than d.

The streaming construction scheme tackles these challenges by incorporating a novel storage format, which introduces three control elements.

control element 1 – Rest: The proposed format discards the *ptr* array and instead inserts a *Rest* element with row-id -1 into the *idx* array (correspondingly insert 0 into the *val* array) at the end of a fiber. In Figure 5(b), we insert four *Rest* elements after the element-b, -e and -g. Continuous occurrences of two *Rest* indicate a fiber without non-zeros. In this case, the accelerator can access *idx* and *val* in a streaming fashion, and identify the end of each fiber from the streaming data without additional control logic.

control element 2 – Padding: The RAW conflicts occur when the distance between two non-zeros with the same row-id is less than d. We insert *Padding* elements between the conflicting non-zeros to increase the distance. Assuming $d = 5$, streaming construction traverses the *idx* array and checks whether each element conflicts with its following five elements, then inserts several *Padding* elements (row-id -2 and value 0) before the conflicting element once a conflict is detected (lines 21 to 23 in Algorithm 1). Figure 5(b) shows streaming construction first detects a conflict between element-a and element-c and then inserts a *Padding* before element-c. Similarly, we insert a *Padding* before element-f to avoid RAW conflicts with element-c.

control element 3 – Blocking: As discussed in Section 2.2.4, the column-wise approach needs to hold the partial sums with size of K on-chip. However, practical applications may involve large-scale sparse matrices, leading to a large on-chip memory requirement. Therefore, we introduce *Block* with row-id -3 and value 0 to partitioned multiplicand **A** into several sub-matrices, tightening the on-chip memory budget.

Algorithm 1 provides a detailed implementation of streaming construction for sparse matrices. We prioritize the insertion of *Rest* (line 3) and *Block* (line 8) elements, as both can increase the distance between non-zeros, reducing the number of *Padding* elements inserted in line 16.

4 Mentor Architecture

This section introduces the Mentor architecture and our analytical model for design space exploration to provide an optimized data path and achieve near-optimal performance.

ALGORITHM 1: Streaming Construction for Sparse Matrices

```
Input  :(sparse matrix $A$ in CSC): $ptr, idx, val, K, NNZ, D, B$
Output:(processed martrix $A$): $pidx, pval$
$init(pidx, idx)$;
$init(pval, val)$;
//Insert the Rest elements
for $i = 1$ to $K + 1$ do
    $insertRest(pidx, pval, ptr[i] + i - 1, 1)$;
end
$pidx[NNZ + K] = -4$;
// Insert the Block elements
$r = 0$;
while $idx[r]! = -4$ do
    if $!isControlElement(r)$ && $!isControlElement(r + 1)$ then
        $insertBlock(pidx, pval, r, (pval[r + 1]/B) - (pval[r]/B))$;
    end
    $r + +$;
end
// Insert the Padding elements
$r = 0$;
while $pidx[r]! = -4$ do
    if $!isControlElement(pidx[r])$ then
        for $i = r + 1$ to $r + D + 1$ do
            if $pval[r] == pval[i]$ then
                $insertPadding(pidx, pval, i, D - (r - i))$;
            end
        end
    end
    $r + +$;
end
```

4.1 Column-Wise Pipeline Design

Our Mentor architecture design has four stages: read, multiplication, accumulation, and write.

4.1.1 Phase I—Read. The *Read B* module streams the constructed matrix **B** into the architecture. Following the delayed feeding strategy discussed in Section 3.1, the number of traversed **B** elements in each cycle is less than #PE. Thus, the *Read B* module will distribute the **B** elements to each PE in a round-robin manner. Simultaneously, the *Read A* module streams both the *idx* and *val* arrays, and broadcasts the non-zeros of **A** to all PEs, as explained in the second advantage of the column-wise approach in Section 2.3. Thanks to the column-wise approach, each PE will independently access several columns of **B** rather than the entire matrix. And we avoid expensive bus control and memory interface by streaming access pattern. Thus, the *Read B* module employs two dependent crossbars for transferring data to four PEs with two channels, reducing the hardware overhead (Figure 6).

4.1.2 Phase II—Multiplication. Each PE employs two FIFOs for the streamed elements from Phase I and a fully-pipelined multiplier, performing scalar-vector product, $\mathbf{A}[:, rid] \cdot \mathbf{B}[rid, i]$. In each cycle, one non-zero $\mathbf{A}[m, rid]$ is consumed for producing a partial product with a value of

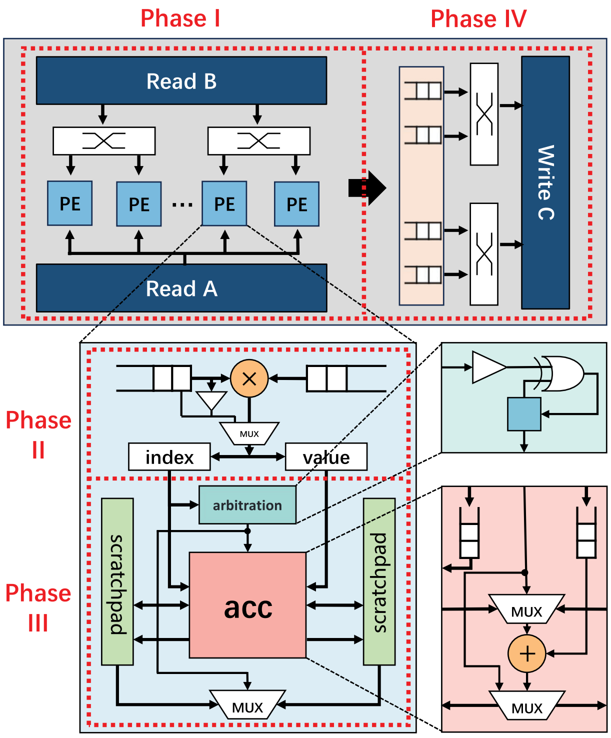

Fig. 6. The micro-architecture of Mentor.

$\mathbf{A}[m, rid] \cdot \mathbf{B}[rid, i]$ and row-id of m. The row-ids will be sent to the FIFO in the accumulation unit and a comparator, which triggers a completion signal when the consumed element is either *Rest* or *Blocking*, indicating the completion of the scalar-vector product. This signal is connected to the FIFO storing **B** elements and the multiplier to control the starting of the following scalar-vector product and the output of the multiplier. Whenever the multiplier receives the completion signal, it produces a result with a value of 0 and a row-id of -1, indicating an invalid output. In addition, the *Padding* element triggers a skipping signal for the multiplier to produce a result with a value of 0, avoiding RAW conflicts.

4.1.3 Phase III—Accumulation. The accumulation phase consumes a row-id and a value computed in Phase II. The row-id m is used to fetch the $\mathbf{C}[m, i]$ from the scratchpad memory, and the fully-pipelined accumulator then performs $\mathbf{C}[m, i] += \mathbf{A}[m, rid] \cdot \mathbf{B}[rid, i]$. After the completion of an entire column $\mathbf{B}[:, i]$, the $\mathbf{C}[:, i]$ stored in the scratchpad memory will be streamed out to the *Write C* module. This process will stall the following accumulation operations. Inspired by the double-buffer technique, we avoid the pipeline stall by employing a double-scratchpad structure to alternately store the partial columns and stream out a final column of **C** to DRAM. This on-chip structure is controlled by an arbitration signal, which flips (XOR with the comparator output) when the arbitration unit detects a *Rest* or *Blocking* element. The accumulation unit employs two multiplexers to determine the destination scratchpad for fetching and writing back and the source scratchpad for streaming out final rows.

4.1.4 Phase IV—Write. During the accumulation phase, a column of **B** is consumed, and a corresponding column of **C** is produced. To ensure a balanced throughput in a fully-pipelined design, the *Write C* module is equipped with the same bandwidth as the *Read B* module and also follows the streaming fashion. Thus, we also adopt a c-cycle delayed strategy for writing the final **C**. Specifically, the *Write C* module concurrently collects $\#PE/(c+1)$ elements from #PE FIFOs in a round-robin manner and streams the elements to the off-chip memory. In every $c+1$ clock cycles, the *Write C* module can stream #PE elements, which forms a segment of a row of **C**. Therefore, the final C is also stored in blocks with row-major storage, and the dimension of each block is $M \times \#PE$. By adopting this strategy, we ensure a synchronization between the consumption of **B** and the production of **C**. Figure 6 shows a design resembling Phase I, where two crossbars are employed to write the results of four FIFOs back to DRAM.

4.2 Analytical Model

Our analytical model is motivated by the delayed feeding strategy adopted in streaming construction. First, given the definition of the delay parameter c,

$$c = \frac{P}{E_b} - 1, \tag{6}$$

where P represents #PE, and E_b is the number of transferred **B** elements per cycle. A larger c indicates a longer interval between two consecutive feeds for each PE, reducing the required bandwidth but lowering PE utilization. Thus, c presents a tradeoff between resource consumption and performance in Mentor. To explore the design space and determine the optimal value of c, we introduce an analytical model before configuring Mentor.

Section 3.1 has introduced that the execution time of $\mathbf{A}[:, rid] \cdot \mathbf{B}[rid, i]$ varies with the number of non-zeros in $\mathbf{A}[:, rid]$. Thus, it is necessary to obtain a reliable estimate of the execution time, denoted as T, based on features of the input matrix. The most conservative estimate considers the minimum number of non-zeros in rows of **A**, maximizing the PE utilization in Mentor but significantly limiting throughput. As a compromise, we calculate the average number of non-zeros per row, denoted as *npr*, and utilize the function f to make a more conservative estimate:

$$T = f(npr) = 2^{\lfloor \log_2 npr \rfloor}, \quad npr = \frac{nnz}{m}, \tag{7}$$

$f(x)$ is defined with the consideration that the preferred #PE is a power of 2, which can reduce the sensitivity of #PE to data features, enabling excellent performance across input data with varying features. The optimal value of the delay parameter c can be determined as $T-1$. By referring to Equation (6), we can draw the following two conclusions:

$$P = f(npr) \cdot E_b, \quad E_b = \frac{P}{f(npr)}. \tag{8}$$

Finally, we employ the following two equations to estimate #PE configured on Mentor and the off-chip bandwidth of the entire system:

$$\begin{aligned} P &= f\left(\frac{nnz}{m}\right) \cdot E_b \\ bandwidth &= (E_a + E_b + E_c) \cdot W \\ &= \left(2 + 2 \cdot \frac{P}{f(\frac{nnz}{m})}\right) \cdot W \end{aligned}, \tag{9}$$

where W is the memory width. The entire system allocates bandwidth for the following three modules: *Read A* module, which only reads one non-zero of **A** per cycle, consisting of an index and a value, *Read B*, and *Write C* modules, which transfer E_b and E_c elements per cycle. The balanced

Table 3. Resource Utilization on xczu15eg-ffvb-2-i

	Resource	Used	Available	Util.
Mentor	LUT	119,631	341,280	35.05%
	LUTRAM	16,871	184,320	9.15%
	FF	159,093	682,560	23.31%
	BRAM	485	744	65.19%
	DSP	476	3528	13.49%

Table 4. Properties of 125 Sparse Matrices and the GCN Benchmarks

	min	**max**		Dimension		Density		
Row / Column	72	84,414		**Node**	**Feature**	**A**	**X**	**W**
NNZ	1,012	28,715,634	Cora	2708	1433	0.18%	1.27%	100%
Density	0.098%	19.52%	Citeseer	3327	3703	0.11%	0.85%	100%
Memory (KB)	8.2	224411.2	Pubmed	19717	500	0.028%	10%	100%

throughput makes $E_b = E_c$. Overall, the analytical model utilizes *npr* to determine an optimal value for c, thereby achieving a good tradeoff between hardware resources and throughput.

5 Evaluation Setup

5.1 Mentor Implementation

Mentor is implemented on the `xczu15eg-ffvb-2-i` board through `Xilinx Vitis v2022.2` with the High-Level Synthesis tool. Its maximum off-chip bandwidth is 9600MB/s. The achieved post-implementation frequency is 214 MHz according to the post-implementation timing report from `Xilinx Vivado v2022.2`. We configure the capacity of each scratchpad and FIFO as 16KB and 128B, respectively. The number of PE is set to 32. Table 3 shows the resource utilization. For the *Read A*, *Read B*, and *Write C* modules, we have assigned AXI slave interfaces with data widths of 64-bit, 128-bit, and 128-bit, respectively. Note that Mentor adopts a 7-cycle delayed feeding.

5.2 Benchmarks

Table 4 summarizes that we randomly selected 125 sparse matrices with varying size, sparsity, and structures from the SuiteSparse Matrix Collection [7]. We construct ten corresponding dense matrices for each sparse matrix by setting N from 32 to 1024 and perform ten iterations of single-precision floating-point SpMM operations. Note that $\alpha = 1.0$ and $\beta = 0.0$.

5.3 Baselines

We use two state-of-the-art SpMM-dedicated accelerators, i.e., Sextans [41] and AWB-GCN [13].
Sextans (row-wise) [41]: This is a software-hardware co-design system based on the row-wise approach. We leverage the non-zero scheduling algorithm on the host and then implement Sextans on the same FPGA, where the number of multipliers is scaled to 32.
AWB-GCN (column-wise) [13]: This accelerator is designed for consecutive SpMMs based on the column-wise approach. We model its performance on a single SpMM operation with the same hardware resources (i.e., 32 multipliers, 0.5MB on-chip memory for accumulation buffer array, and 0.5MB for scratchpad memory) as other baselines and matrix blocking optimization, where t is set to 4. Since the density of benchmarks is less than 25%, we use TDQ-2 (task-distribution-queue) to distribute tasks to PEs and 1-hop distribution smoothing.

Given that there are very few SpMM-dedicated accelerators, we modify and fine-tune three SpGEMM accelerators (i.e., SIGMA, Spaghetti, and GAMMA) to support SpMM. To compare these architectures regardless of their underlying platforms, we build cycle-accurate simulators to evaluate their performance.

SIGMA (inner) [35]: This accelerator uses MAC arrays with distribution networks and reduction networks to process bitmap-compressed sparse matrices. SIGMA is a variant of the inner-product approach, which uses the **B**-stationary, **A**-streaming dataflow (referred to as inner(N) in Table 1). Due to **B**'s dense nature, generating source-destination pairs in SIGMA can be simplified to store non-zeros' positions in the fibers of **A**. For the hardware configuration, we use a 4×8 MAC array with an accumulator buffer of 1MB and the same off-chip bandwidth as that of MENTOR. Following the original work, the total latency is obtained by *Loading+Streaming+Add*.

Spaghetti (outer) [18]: We reproduce the pattern-aware scheduling to tile and segment the multiplicand **A**. Due to the Pareto-optimal configuration of N_m (the number of multipliers) being 16 in Spaghetti, we use 32 multipliers and double the bandwidth of the original work to maintain a fair comparison while leveraging the advantages of outer products. We configure the sort-merger depth to be 8K to ensure the same on-chip SRAM budget.

GAMMA (row-wise) [59]: This design features a FiberCache to capture irregular data reuse in a row-wise approach. We simplify its merge component since the elements in the rows of **B** are inherently ordered. We implement the affinity-based row reordering to preprocess **A**. The FiberCache is scaled to 1MB for hardware, and the size of the cache line is fixed to 32. We use the original 128GB/s bandwidth to support 32 64-radix PEs in our model.

6 Experimental Results

This section compares MENTOR to state-of-the-art SpMM approaches and shows the efficacy of our analytical model. It also discusses how MENTOR performs in GCN and its overhead.

6.1 Comparing to State-of-the-Art

6.1.1 Overall Performance. We partition the 1250 SpMMs into 6 groups according to problem size (i.e., the number of floating-point operations, defined as $2 \cdot nnz \cdot N$). Figure 7(a)–(c) shows the results normalized to SIGMA regarding speedup, traffic reduction, and bandwidth utilization. We see that MENTOR achieves a geomean of 2.05× speedup, 2.92× memory traffic reduction, and 1.38× bandwidth utilization improvement against the state-of-the-art. This is because of three factors.

Firstly, the fully-pipelined design of MENTOR avoids the potential pipeline stalls and achieves a near-optimal performance. MENTOR performs 3.06× and 1.94×, 1.31×, 3.98× and 1.18× better than SIGMA, Spaghetti, GAMMA, Sextans and AWB-GCN. MENTOR achieves an impressive measured peak throughput of 13.63 GFlops/s, close to its theoretical peak of 13.65 GFlops/s. SIGMA employs the bitmap storage for index matching, which has a non-negligible impact when processing sparse matrices. On 125 matrices, this operation accounts for 33% of the total latency. Spaghetti is limited by the merge phase for partial sums. Although it attempts to pipeline the computation and merge phase by matrix tiling, the sort-merger latency averages 3× that of the computation phase. GAMMA and Sextans use on-chip storage structures to overcome the irregular data reuse of the row-wise approach. GAMMA suffers from the increasing cache miss rate on large-scale problems. Sextans fails to hide the latency of memory accessing with computation, and the FPGA board's low bandwidth significantly limits its performance. AWB-GCN demonstrates the benefits of column-wise by hardware auto-tuning and employs a scratchpad to cache **A** as much as possible. These strategies deliver good performance on small-scale computations. However, the workload needs to be re-balanced after each tile is calculated, limiting the PE utilization on large-scale computations.

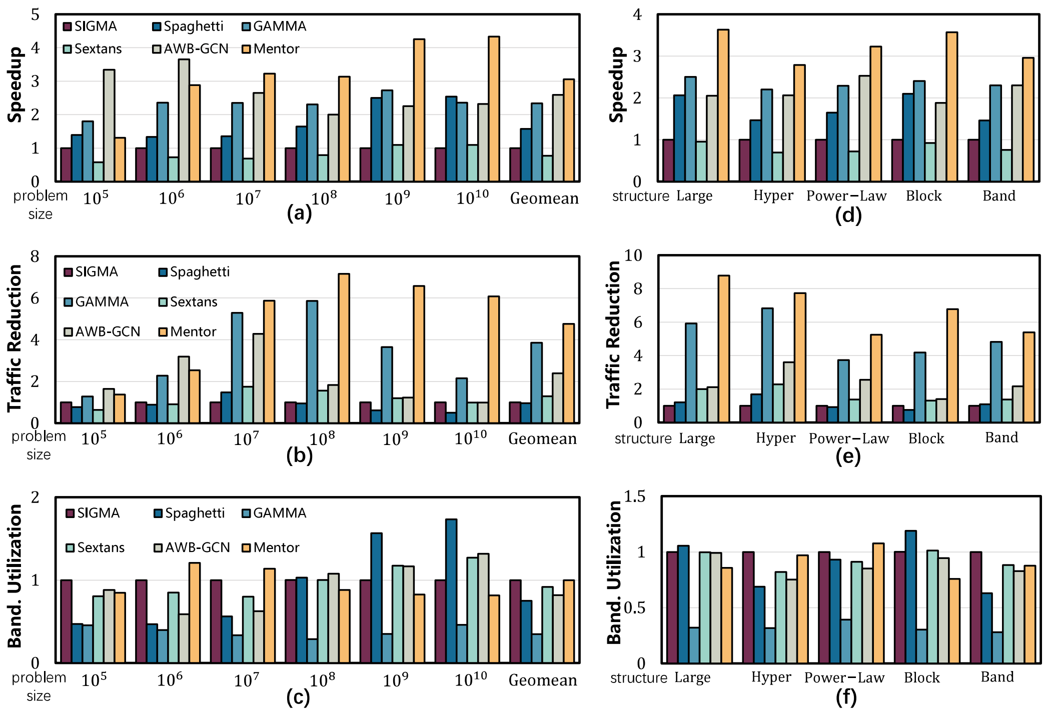

Fig. 7. Performance of Mentor over SIGMA, Spaghetti, GAMMA, Sextans, and AWB-GCN. The results in (a)–(c) are grouped by the problem size, and (d)–(f) shows the performance on matrices with special structures, namely Large (large matrices), Hyper (hyper-sparse matrices), Power-Law (matrices following power-law distribution), Block (block-sparse matrices), and Band (band matrices).

Secondly, column-wise's inherent advantages result in a substantial reduction in memory traffic. Mentor exhibits geomean reductions of 4.76×, 4.93×, 1.23×, 3.68× and 1.99× over state-of-the-art. On problems larger than 10^8, the baselines suffer heavy memory traffic, but Mentor achieves reductions ranging from 1.84× to 9.98×. SIGMA uses the Inner(N) dataflow, but the bitmap is space-inefficient on sparse matrices and accounts for an average of 66% of the total traffic. Spaghetti limits the merge of partial sums on-chip, which sacrifices good input reuse of multiplicand **A**. GAMMA can hold the entire small matrix with the FiberCache and perform well on problems smaller than 10^8. But it struggles with large matrices, and exhibits 2.82× more memory traffic over Mentor on problems larger than 10^{10}. Sextans sacrifices the reuse of **A**, resulting in 5.34× more traffic over Mentor on large problems. AWB-GCN can cache the entire **A** on problems smaller than 10^7 and achieves 1.27× reduction against our method. But this advantage diminishes as the problem scale increases, leading to 0.2× traffic reduction against Mentor on large problems.

Thirdly, the streaming access pattern reduces the average memory access latency and improves bandwidth utilization of 1.0×, 1.33×, 2.89×, 1.08×, and 1.21×. SIGMA shows comparable utilization because 33% of the total latency is spent searching for source-destination pairs by accessing bitmaps sequentially. The critical path of Spaghetti is sort-mergers, which leads to low utilization on small-scale problems. For GAMMA, most memory accesses occur when there are cache misses, and it can exhibit better utilization on small-scale computations. Sextans can exploit an enhanced data locality and demonstrate improved bandwidth utilization when the problem size exceeds 10^8. Unfortunately, it cannot overlap memory access and computation, resulting in minimal efficiency when dealing with small-scale problems. The AWB-GCN trend resembles that of Sextans, with access to **A** dominating memory traffic in large-scale problems.

6.1.2 The Impact of Matrix Structure. We further discuss the impact of special structure on the overall performance in Figure 7(d)–(f). We include five types of matrices, where large matrices (M > 10K) and hyper-sparse matrices (density < 0.01 [18]) represent the matrix scales, and power-law, block-sparse, and band matrices represent the distribution of non-zeros in **A**. For power-law matrices, the non-zeros are usually clustered in some fibers. For block-sparse matrices, the non-zeros are usually clustered in some matrix blocks and form many dense blocks. For band matrices, the non-zeros are primarily distributed along the diagonals, and adjacency fibers exhibit similar access patterns.

Firstly, Mentor achieves a 2.06×–2.31× geomean speedup across the five structures, with a 2.31× on block-sparse matrices and a 2.28× on large matrices. Mentor can benefit from the dense blocks on block-sparse matrices, and introduce fewer control elements in streaming construction. As the baselines are all dedicated to sparse computation, they struggle to leverage the hardware performance on dense blocks. In particular, AWB-GCN performs worst on this structure due to the dense blocks alleviating the workload imbalance. Only GAMMA performs well because the similar access pattern exhibited by the dense blocks can avoid cache misses. For large matrices, as previously discussed, all baselines suffer from significant memory access overhead. In contrast, our method mitigates this issue by optimizing the access pattern.

Secondly, Mentor reduces memory traffic by 3.11×–4.77×. The better performance can be observed on block and large matrices. On hyper-sparse and band matrices, Mentor's traffic reduction is comparable to GAMMA, which is 1.13× and 1.12×, respectively. On the two structures, the streaming construction scheme introduces more padding elements to avoid RAW conflicts, leading to more redundant computations. The FiberCache in GAMMA can benefit from these features, resulting in a low miss rate.

Thirdly, Mentor improves bandwidth utilization by 0.94×–1.45×. On hyper-sparse and power-law matrices, Mentor improves utilization by 1.45× and 1.39× compared with the baselines. This is because both matrices exhibit poor spatial locality of non-zero, and Mentor can effectively mitigate this issue by streaming construction. Only on block-sparse matrices, Mentor exhibits poor bandwidth utilization, which can be attributed to the good locality of dense matrix blocks. Spaghetti performs well because the adopted matrix tiling enhances locality when accessing dense matrix blocks.

6.2 Comparing to CPU and GPU

We then compare Mentor with state-of-the-art SpMM implementations on CPUs and GPUs. Specifically, we compare Mentor against MKL on CPUs and cuSPARSE on GPUs regarding energy efficiency, defined as the throughput divided by the total power consumption. We utilize an Intel Core-i5 12600K CPU with 10 cores and 16 threads to execute *mkl_sparse_s_mm* for SpMM, measuring power using the RAPL interface [6]. We use an NVIDIA RTX 3060 GPU with a 360GB/s bandwidth of GDDR6 and measure the execution time of *cusparseSpMM* in cuSPARSE. Additionally, we monitor power consumption using *nvidia-smi*. Note that we measure the real-time power consumption of CPU and GPU for comparison rather than **thermal design power** (**TDP**), with geomean values of 61.7W and 134.8W, respectively. In contrast, we follow the common practice of reporting the total power of Mentor in the power analysis from the implemented netlist, with a value of 6.4W, including static and dynamic power consumption. Figure 8 shows that the geomean energy efficiency of MKL, cuSPARSE, and Mentor are 5.68×10^8, 9.38×10^8 and 1.53×10^9 FLOP/J, respectively. When dealing with large-scale problems, general-purpose processors are limited by

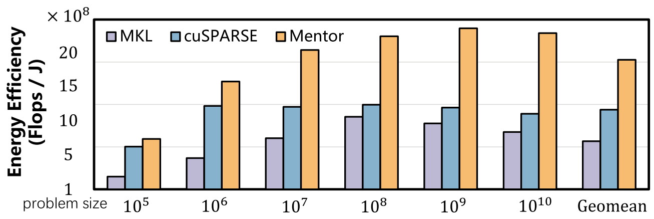

Fig. 8. Energy efficiency of Mentor over software implementations on CPUs and GPUs.

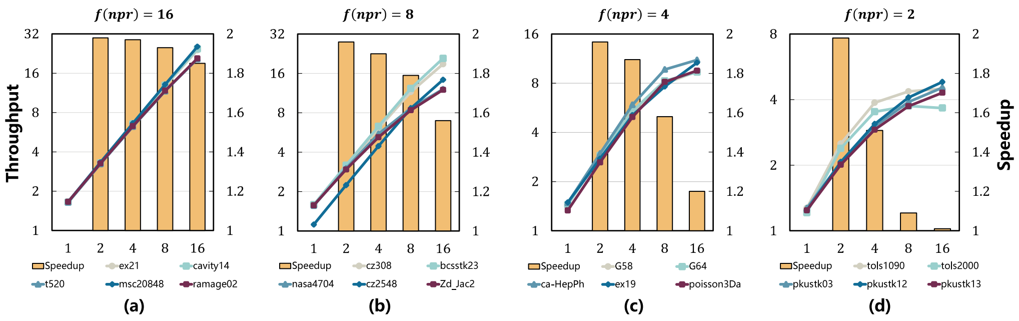

Fig. 9. Scalability with varying delay parameters derived from the analytical model of Mentor.

substantial memory access overhead. By optimizing memory access, Mentor demonstrates 2.49× and 1.86× better energy efficiency than MKL and cuSPARSE for problems larger than 10^7.

6.3 Efficiency of Analytical Model

We perform SpMM operations on 20 sparse matrices with varying features to observe the performance variations with different values of c and demonstrate the efficiency of the analytical model. Note that the off-chip bandwidth remains constant at 9600MB/s and the interface width of *Read B* is set to 128-bit. Thus, we calculate that $E_b = E_c = 4$.

Each subfigure of Figure 9 shows the throughput (lines) of five matrices and the geomean speedup (bars) after doubling #PE with an upper limit of 2. The x-axis denotes c' ($=\#PE/E_b$). Taking Figure 9(a) as an example, for the five matrices with $f(npr)$ larger than 16, our model predicts that $c' = 64$. We see that Mentor exhibits promising speedups, ranging from 1.85× to 1.98× when doubling c', and it is smaller than 64. We observe the same trend in Figure 9(c), where the predicted estimate of c' is 16. Here, the speedups are 1.96× and 1.87× when c' does not exceed 16. But when c' increases from 16 to 32, the speedup declines from 1.87× to 1.58×, indicating that using more hardware resources leads to diminishing performance improvements. Similarly, Figure 9(d) shows that the speedup decreases to 1.01× when c' ranges from 8 to 16. But the accelerator can still achieve 1.98× speedup with $c' = 2$. To summarize, when c' is smaller than our estimate, doubling #PE can yield an average speedup of 1.92×. When using excessive PEs and if c' is larger than our estimate, the speedup will decrease to 1.3×. This demonstrates that our analytical model can provide significant guidance for optimizing the architecture to better balance hardware resources and performance across different problems.

6.4 Case Study: GCN Layer

Since SpMM is a fundamental kernel, Mentor is equally applicable in SpMM-based applications such as graph convolutional networks (GCN). Thus, we evaluate Mentor with the GCN workloads

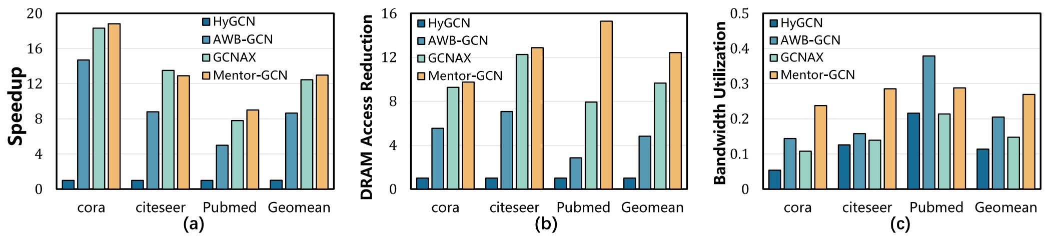

Fig. 10. Performance of Mentor-GCN over three GCN baselines.

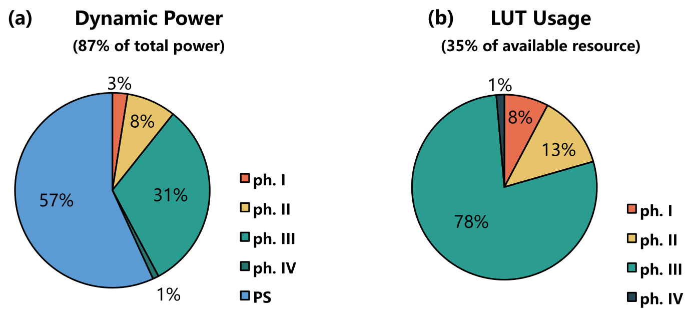

Fig. 11. Dynamic power (a) and LUT (b) usage breakdown.

with a chain of SpMM, i.e., $\mathbf{X}^{(l+1)} = \sigma(\mathbf{A}(\mathbf{X}^{(l)}\mathbf{W}^{(l)}))$, where $\mathbf{A}$ denotes the sparse adjacency matrix, $\mathbf{X}^{(l)}$ and $\mathbf{W}^{(l)}$ represent the sparse feature map and the dense weight matrix of the lth layer. Table 4 shows the structure and sparsity of the three benchmarks. In the SpMM chain, the dense result calculated by $\mathbf{X}^{(l)}\mathbf{W}^{(l)}$ can be directly streamed into the *Read A* module without being written back to the off-chip memory. We reconfigure the architecture as Mentor-GCN with 16 parallel PEs, and compare Mentor-GCN against three state-of-the-art GCN accelerators: HyGCN [53], AWB-GCN [13], and GCNAX [25]. Following the configuration in Reference [25], all baselines use a 1×16 MAC array with a data width of 64-bit and a memory bandwidth of 128GB/s.

We first present speedup with normalized execution cycles in Figure 10(a). On three datasets, Mentor achieves 5-14.7×, 1.28-1.8×, and 0.96-1.15× speedup over HyGCN, AWB-GCN, and GCNAX, respectively. On Citeseer, Mentor only achieves 0.96× speedup over GCNAX. In Mentor-GCN, #PE=16 and $E_b = 4$, thus Mentor-GCN runs with a 3-cycle delayed feeding according to Equation (9). But the estimate of c is 1 because $npr = 2.74$ in the adjacency matrix $\mathbf{A}$ of Citeseer. In other words, the 3-cycle delayed feeding is slightly aggressive, leading to decreased PE utilization and slight performance degradation on Citeseer.

In Figure 10(b) and (c), Mentor exhibits 9.75-15.28×, 1.76-5.36×, and 1.05-1.93× traffic reduction, and 2.38-2.88×, 0.76-1.81×, and 1.35-2.20× higher bandwidth utilization. HyGCN and AWB-GCN employ inefficient execution orders and involve redundant computations and DRAM traffic. GCNAX utilizes an optimized outer product dataflow similar to Spaghetti, which also suffers from a large overhead of partial matrices and exhibits poor performance on Pubmed.

6.5 Overhead Analysis

6.5.1 Power and Hardware Resource. We present the dynamic power and LUT breakdown of Mentor in Figure 11, extracted from the place-and-route results of Vivado 2022.2. The dynamic power consumption is 5.56W, which accounts for 87% of the total power consumption, with a majority (57%) attributed to the **Processing System** (**PS**) side, particularly the DDR.

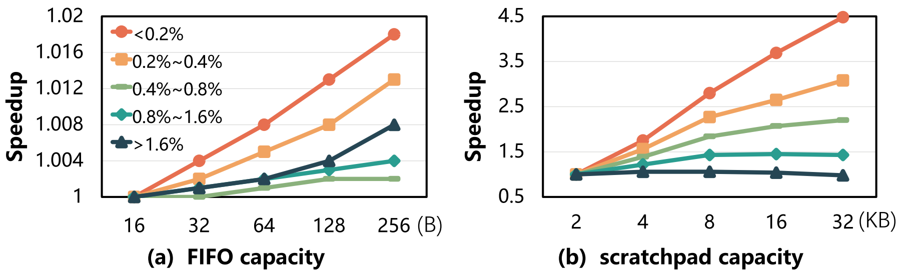

Fig. 12. Performance of Mentor w.r.t the variation of (a) FIFO and (b) scratchpad capacity.

For a fully-pipelined architecture, the accumulation phase involves a tailored on-chip structure, resulting in 31% dynamic power and 78% LUT count requirement. Both phases I and IV exhibit minimal power and resource overhead, indicating that the crossbar does not impose a heavy burden.

6.5.2 Hardware Resource. We further conduct experiments to evaluate Mentor with different on-chip resource support. Figure 12 shows the speedups as FIFO and scratchpad capacities vary. Each line represents the benchmark with various levels of sparsity. We observe that the FIFO capacity has minimal impact on performance, with a maximum speedup of 1.018×, demonstrating that each phase's throughput is approximately equal, thus proving the efficiency of our full-pipelined design on the hardware level. In contrast, we can observe a significant impact of scratchpad capacity on performance from Figure 9. For matrices with a density below 0.2%, configuring the scratchpad capacity to 32KB yields a speedup of 4.48×. This is because the stream construction scheme partitions the matrix **A** into smaller blocks to fit the decreased scratchpad capacity, introducing more *Block* elements. But for matrices with a density exceeding 1.6%, the impact of *Block* elements is minimal, and increasing the scratchpad capacity can not yield considerable performance gains.

6.5.3 Overhead of Streaming Construction. The streaming construction introduces three control elements, increasing the memory footprint of **A**. Our experiments on 125 matrices show that stream construction incurs an average increase of 11% (with a minimum of 0.2%) compared with CSC/CSR. We implement this algorithm in Python and compare its running time with the non-zero scheduling algorithm in Sextans and the affinity-based row reordering in GAMMA. Our approach is 2.47× and 11.51× faster than Sextans and GAMMA, respectively. The construction accounts for 35% of the end-to-end time taken for 10 SpMM iterations.

6.6 Limitations

Mentor has demonstrated the advantages of column-wise-based SpMM accelerators through software-hardware co-design. Here, we discuss two limitations of our approach. (1) **Significant on-chip memory requirements**. Both Mentor and AWB-GCN require substantial on-chip memory to buffer the partial results. On large sparse matrices, the size of the partial results can even exceed 1M, making it challenging to implement an accelerator with tight-budgeted on-chip memories. Matrix partitioning can help alleviate this issue, but this approach involves a tradeoff between hardware resources and data reuse. (2) **Limited scalability on skinny matrices**. Mentor utilizes parallelism in the N-dimension to take advantage of column-wise benefits. In real-world scenarios, N may be smaller than #PE of Mentor, especially for skinny matrices. In this situation, additional hardware resources do not significantly improve Mentor's performance. Therefore, we recommend deploying Mentor with several tens of PEs as a lightweight SpMM core. By utilizing multiple on-chip cores, we can support parallel processing on multiple workloads without incurring significant on-chip communication overhead.

7 Related Work

Hardware accelerators for Sparse Matrix Multiplication. In addition to the sparse accelerators listed in Table 2, there are also sparse algebra accelerators focused on accelerating sparse matrix multiplication with ASIC or FPGA implementations. For example, ALRESCHA [3] and ASCELLA [2] are accelerators based on inner-product, both of which propose custom storage formats to address index-matching and support input streaming. OuterSPACE [33] is the first architecture based on outer-product, which serializes the multiply and merge phases and uses single-program-multiple-data processing units for SpGEMM and SpMV. Following OuterSPACE, SpArch [62] provides a more efficient design with condensed matrix representation and a Huffman tree scheduler to improve output reuse. Apart from these, MatRaptor [42] and InnerSP [4] also recognize the advantages of row-wise dataflow and provide feasible solutions through either custom storage format for enabling streaming fashion (MatRaptor) or on-chip hash table for addressing memory bloating (InnerSP). However, these designs do not exploit the advantages of the column-wise product.

Additionally, some works attempt to accelerate various computation problems using a single dataflow. For example, Tensaurus [43] is a versatile architecture for various sparse-dense tensor computations, proposing the SF^3 pattern based on the inner product and performing well in several computation problems. Recent work has emerged that integrates inner, outer, and row-wise dataflows within a framework [28, 30] to accelerate a single computation problem, dynamically matching the most suitable dataflow according to input data features and demonstrating the suitability of each dataflow varies across different problems. However, these works do not focus on the drastically different sparsity levels of the two operands and fail to deliver better performance on SpMMs.

Furthermore, designs from many other domains have also contributed to the problem of sparse matrix computation. In addition to the mentioned GCN accelerators, there are also many designs specifically tailored towards neural networks [8–10, 15, 22, 34, 35, 47, 55, 60], where the computations can be permuted as matrix operations. GoSPA [10] globally optimizes sparse **convolutional neural networks** (**CNNs**), especially the convolutional computation. SpAtten [47] leverages the sparsity opportunities for improving the performance of the attention mechanism, which involves both sparse and dense computations.

Software techniques for acceleration. Introducing pre-processing at the software level to further enhance the performance of accelerators is a critical technique widely adopted in the domain-specific architecture field. The affinity-based reordering algorithm in GAMMA [59] improves the data locality and has been proven to boost the performance by 18%. The pre-processing algorithm in Sextans [41], a significant reference for our work, avoids RAW conflict and balances loads with the help of non-zero scheduling. Additionally, many works utilize matrix partitioning and customized storage formats, including ALRESCHA, ASECLLA, and MatRaptor. Refs. [32, 51] carefully study the limitations of traditional tiling approaches in sparse tensor algebra and propose new tiling strategies and apply them to existing accelerators for improving the performance. Software algorithms are also widely applied in other workloads involving SpMM, such as sparse attention. Sanger [29] utilizes software pruning to predict the attention mask and rearrange the unstructured sparse patterns and proposes a custom architecture supporting SpMMs. DOTA [36] introduces low-rank linear transformations to learn a mask that detects weak connections in attention maps. ViTCoD [58] separates sparse and dense computation patterns from sparse attention maps through pruning and polarization, enhancing hardware performance. The success of these designs demonstrates that hardware performance can be further enhanced through software innovation.

To summarize, Mentor stands out as a software-hardware co-design approach based on column-wise to accelerate SpMM. Mentor implements streaming at the software level to enable streaming memory accesses and designs a fully-pipelined hardware architecture at the hardware level to adapt to the created data streams for near-optimal performance.

8 Conclusion

We propose a novel SpMM accelerator named Mentor. Built upon the column-wise product, Mentor eliminates random accesses and reduces memory traffic. Mentor further enhances memory and computation efficiency with a pre-processing technique at the software level and a fully-pipelined on-chip design at the hardware level. We also provide an analytical model to explore the design space. Our results with an FPGA prototype Mentor on 1250 SpMM operations show the efficacy of our approach.

Acknowledgments

We appreciate all the anonymous reviewers for their careful evaluations.

References

[1] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In *SpringSim (HPS'15)*. 75–82.

[2] Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim. 2020. Ascella: Accelerating sparse computation by enabling stream accesses to memory. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE'20)*. IEEE, 318–321.

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. Alrescha: A light-weight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 249–260.

[4] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT'21)*. IEEE, 116–128.

[5] Roberto L. Castro, Diego Andrade, and Basilio B. Fraguela. 2022. Probing the efficacy of hardware-aware weight pruning to optimize the SpMM routine on Ampere GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 135–147.

[6] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. 189–194.

[7] 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.

[8] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 749–763.

[9] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K. Parhi, Xuehai Qian, and Bo Yuan. 2018. PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 189–202.

[10] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, 1110–1123.

[11] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2020. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14629–14638.

[12] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[13] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 922–936.

[14] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. SPADE: A flexible and scalable accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[15] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–49.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[17] Mikael Henaff, Joan Bruna, and Yann LeCun. 2015. Deep convolutional networks on graph-structured data. arXiv:1506.05163. Retrieved from https://arxiv.org/abs/1506.05163

[18] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. Spaghetti: Streaming accelerators for highly sparse gemm on fpgas. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 84–96.

[19] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. 2019. A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 6 (2019), 1272–1285.

[20] 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*. IEEE, 1–12.

[21] Advanced Micro Devices Inc. 2022. UltraScale Architecture-Based FPGAs Memory IP Product Guide (PG150). Retrieved from https://docs.amd.com/v/u/en-US/pg150-ultrascale-memory-ip

[22] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 600–614.

[23] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. A high-performance parallel algorithm for nonnegative matrix factorization. *ACM SIGPLAN Notices* 51, 8 (2016), 1–11.

[24] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

[25] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. 2021. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 775–788.

[26] Shiqing Li, Shuo Huai, and Weichen Liu. 2023. An efficient gustavson-based sparse matrix–matrix multiplication accelerator on embedded FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4671–4680.

[27] Shiqing Li and Weichen Liu. 2023. Accelerating Gustavson-based SpMM on embedded FPGAs with element-wise parallelism and access pattern-aware caches. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE'23)*. IEEE, 1–6.

[28] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 747–761.

[29] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[30] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient DNN processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 252–265.

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, and Misha Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. arXiv preprint arXiv:1906.00091 (2019).

[32] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Fletcher W. Christopher. 2023. Accelerating sparse data orchestration via dynamic reflexive tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 3, 18–32.

[33] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse

matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 724–736.

[34] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 27–40.

[35] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 58–70.

[36] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. Dota: Detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 14–26.

[37] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*. IEEE, 256–266.

[38] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Umit V. Çatalyürek. 2015. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing* 76, C (2015), 106–119.

[39] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 593–607.

[40] Haihao Shen, Hengyu Meng, Bo Dong, Zhe Wang, Ofir Zafrir, Yi Ding, Yu Luo, Hanwen Chang, Qun Gao, Ziheng Wang, Guy Boudoukh, and Moshe Wasserblat. 2023. An efficient sparse inference software accelerator for transformer-based language models on CPUs. arXiv preprint arXiv:2306.16601 (2023).

[41] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[42] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 766–780.

[43] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 689–702.

[44] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*. 353–364.

[45] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A high performance sparse tensor algebra compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, IEEE, 27–38.

[46] Elias Trommer, Bernd Waschneck, and Akash Kumar. 2021. dCSR: A memory-efficient sparse matrix representation for parallel neural network inference. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD'21)*. IEEE, 1–9.

[47] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 97–110.

[48] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys* 55, 12 (2023), 1–36.

[49] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. *Advances in Neural Information Processing Systems* 29 (2016), 2074–2082.

[50] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2020), 4–24.

[51] Zi Yu Xue, Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2023. Tailors: Accelerating sparse tensor algebra by overbooking buffer capacity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1347–1363.

[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[53] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 15–29.

[54] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*. Springer, 672–687.

[55] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I.-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. 2019. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*. 236–249.

[56] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7370–7377.

[57] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. 2022. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. IEEE, 460–474.

[58] Haoran You, Zhanyi Sun, Huihong Shi, Zhongzhi Yu, Yang Zhao, Yongan Zhang, Chaojian Li, Baopu Li, and Yingyan Lin. 2023. Vitcod: Vision transformer acceleration via dedicated algorithm and accelerator co-design. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*. IEEE, 273–286.

[59] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[60] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[61] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: A comprehensive review. *Computational Social Networks* 6, 1 (2019), 1–23.

[62] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 261–274.

[63] Marcela Zuluaga, Peter Milder, and Markus Püschel. 2016. Streaming sorting networks. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 21, 4 (2016), 1–30.

Received 7 March 2024; revised 24 June 2024; accepted 23 July 2024