FPGA-Based Sparse Matrix Multiplication Accelerators: From State-of-the-Art to Future Opportunities

YAJING LIU, Fuzhou University, Fuzhou, China
RUIQI CHEN, Vrije University Brussel, Brussel, Belgium
SHUYANG LI, Fudan University, Shanghai, China
JING YANG, Fuzhou University, Fuzhou, China
SHUN LI, Fuzhou University, Fuzhou, China and VeriMake Innovation Lab, Nanjing, China
BRUNO DA SILVA, Vrije University Brussel, Brussel, Belgium

Sparse matrix multiplication (SpMM) plays a critical role in high-performance computing applications, such as deep learning, image processing, and physical simulation. Field-Programmable Gate Arrays (FPGAs), with their configurable hardware resources, can be tailored to accelerate SpMMs. There has been considerable research on deploying sparse matrix multipliers across various FPGA platforms. However, the FPGA-based design of sparse matrix multipliers still presents numerous challenges. Therefore, it is necessary to summarize and organize the current work to provide a reference for further research. This article first introduces the computational method of SpMM and categorizes the different challenges of FPGA deployment. Following this, we introduce and analyze a variety of state-of-the-art FPGA-based accelerators tailored for SpMMs. In addition, a comparative analysis of these accelerators is performed, examining metrics including compression rate, throughput, and resource utilization. Finally, we propose potential research directions and challenges for further study of FPGA-based SpMM accelerators.

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

Additional Key Words and Phrases: Field-Programmable Gate Array (FPGA), Sparse Matrix Multiplication, Compress Ratio, Accelerator

ACM Reference format:
Yajing Liu, Ruiqi Chen, Shuyang Li, Jing Yang, Shun Li, and Bruno da Silva. 2024. FPGA-Based Sparse Matrix Multiplication Accelerators: From State-of-the-Art to Future Opportunities. *ACM Trans. Reconfig. Technol. Syst.* 17, 4, Article 59 (November 2024), 37 pages.
https://doi.org/10.1145/3687480

Yajing Liu and Ruiqi Chen contributed equally to this research.

Authors' Contact Information: Yajing Liu, Fuzhou University, Fuzhou, China; e-mail: 221127076@fzu.edu.cn; Ruiqi Chen (corresponding author), Vrije University Brussel, Brussel, Belgium; e-mail: ruiqi.chen@vub.be; Shuyang Li, Fudan University, Shanghai, China; e-mail: 20307130373@fudan.edu.cn; Jing Yang, Fuzhou University, Fuzhou, China; e-mail: 221127109@fzu.edu.cn; Shun Li, Fuzhou University, Fuzhou, China and VeriMake Innovation Lab, Nanjing, China; e-mail: 211127144@fzu.edu.cn; Bruno da Silva, Vrije University Brussel, Brussel, Belgium; e-mail: bdasilva@etrovub.be.

ACM 1936-7414/2024/11-ART59
https://doi.org/10.1145/3687480

1 Introduction

Sparse Matrix Multiplication (SpMM) plays[1pc] a crucial role in numerous scientific and **High-Performance Computing (HPC)** applications, such as image processing [2, 3], deep learning [49], physics simulation [45], molecular dynamics [27], and several other fields.

Currently, numerous studies focus on accelerating SpMM on platforms such as CPU, GPUs, **Field-Programmable Gate Arrays (FPGAs)**, and **Application-Specific Integrated Circuits (ASICs)** [14].

CPU, as a general-purpose computing platform, is often the first choice for deploying and accelerating SpMM computations. Advanced CPUs have a few high-performance cores, with advantages in executing complex control flows, supporting a wide range of instruction sets, and handling multitasking. Several optimizations for accelerating SpMM on CPUs have been proposed. Gu et al. [18] developed the PB-**Sparse Matrix-Matrix Multiplication (SpGEMM)** algorithm based on the outer product, which can saturate memory bandwidth by using propagation blocking techniques and performing in-cache sorting and merging. Chen et al. [7] designed a partitioning and parallelization method for **Compressed Sparse Row (CSR)**-based SpGEMM tailored to the Sunway architecture. Elliott and Siefert [15] proposed a single-channel OpenMP variant based on the Gustavson algorithm to handle matrix squaring computations.

GPUs consist of over thousands stream processors and are well-suited for data-intensive parallel computing. Benefiting from the excellent parallel computing capabilities, GPUs have evolved from initially accelerating general matrix multiplication operations to becoming the focus of specific SpMM optimizations [31]. Liu et al. [41] proposed three register-aware optimization techniques to enhance SpGEMM performance, covering representative parallel primitives, namely sorting, merging, and hashing. Deveci et al. [12] described a data placement method and a block-based algorithm that utilizes the multiple memory spaces present in each hardware platform. Parger et al. [50] proposed a lightweight, multilevel matrix analysis that dynamically selects and adjusts the best-fitting algorithm for each row of the matrix.

ASICs are often regarded as a candidate to accelerate SpMM, too. For instance, OuterSPACE [48] proposed a reconstructed memory hierarchy that reduces traffic to the main memory. Gamma [70] utilized dynamically scheduled **Processing Elements (PEs)** with efficient high-radix merge capabilities to achieve high throughput. SpArch [71] designed a stream-based merging approach to optimize data reuse. MatRaptor [59] introduced a new sparse storage format, **Cyclic Channel Sparse Row** [C^2SR], enhancing memory bandwidth utilization through vectorization and stream access.

However, the irregular distribution of nonzero elements in sparse matrix limits the performance and energy efficiency of SpMM on general-purpose computing platforms such as CPUs and GPUs. Specifically, the access pattern to nonzero elements is irregular, necessitating a highly optimized and efficient storage and access mechanism. For CPUs and GPUs with fixed memory hierarchies, it is challenging to optimize this type of irregular and intensive task. Moreover, SpMM requires handling a significant amount of intermediate results during computation, which leads to frequent off-chip memory accesses. This further increases computational latency and limits overall computational performance. As for ASICs, once the design is complete, the architecture is fixed. Consequently, ASICs lack flexibility and cannot be redeveloped or modified, making their deployment difficult for small-scale or rapidly changing applications [1].

The success of FPGAs in HPC applications has made them an increasingly popular choice for hardware acceleration platforms [9]. Compared to traditional CPUs, GPUs, and ASICs, FPGAs offer several advantages, including flexible data paths, low latency, reconfigurability, and higher energy efficiency [29]. Developers can customize data paths to precisely match the data access and

processing patterns of sparse matrices. By optimizing these data paths and minimizing unnecessary data access, computational efficiency is significantly enhanced. Additionally, the reconfigurability of FPGAs provides high adaptability to the rapid evolving parameters and algorithms. As in the most common application of SpMM [31], neural network inference, new quantization methods, compression formats, and model architectures are being proposed. It suggests designers need to fine-tune data paths and microarchitectures to accelerate the SpMM computations in accordance to these methods [22, 54].

Therefore, FPGAs are a suitable candidate for hardware acceleration of SpMM. For instance, Spaghetti [22] introduced an FPGA-based SpMM hardware generator and a deployable prototype, offering high input reuse of nonzero values. Sextans [58] is another SpMM architecture based on outer product implementation, maximizing the use of PEs and improving load balancing. eSSpMV [5] targets symmetric sparse matrices, enhancing random memory access efficiency through an improved **Symmetric Compressed Sparse Row (SCSR)** format in conjunction with a data access module. Serpens [57] implemented a general-purpose **Sparse Matrix-Vector Multiplication (SpMV)** accelerator on an FPGA with **High-Bandwidth Memory (HBM)**.

Although the primary focus of this article is to review FPGA-based SpMM accelerators, it is also essential to highlight the foundational role that SpMV accelerators play in this field. SpMV can be considered a special case of SpMM, where the inner and outer product computations are essentially SpMV operations. Therefore, the study of SpMV reveals the intrinsic challenges of sparse matrix computations, such as load imbalance and memory access efficiency issues, and provides a wealth of strategies and techniques to overcome these challenges. Many optimization techniques developed for SpMV, including innovations in storage formats, load balancing, and improvements in memory access patterns, offer valuable insights and inspiration for the design of SpMM accelerators. As a result, while presenting SpMM accelerators, this article also reviews relevant SpMV accelerators. This dual focus aims to establish a robust theoretical and practical foundation for better understanding and advancing FPGA-based SpMM accelerators. By examining the relationship between SpMV and SpMM, and the ways in which SpMV research informs SpMM accelerator design, we aim to provide a comprehensive context for the discussions and findings presented in this article.

Accelerating SpMM on FPGAs, however, is not a straightforward task. Due to the irregularity of sparse data, customizing FPGA-based accelerators for SpMM encounters the following challenges:

— *Dataflow Selection*: The selection of dataflow determines the computation method for SpMM. It further necessitates designers to consider matrix storage formats, memory access design, and parallel computing architecture based on hardware resource constraints.
— *Matrix Storage Efficiency*: Sparse matrices are characterized by a significant amount of zero elements. Storing sparse matrices in the same way as regular matrices (dense matrices), where every element is stored, leads to significant memory wastage. Therefore, it's essential to compress sparse matrices for storage. Effective data compression formats can reduce memory overhead.
— *Memory Access Efficiency*: The random distribution of nonzero elements in sparse matrices necessitates continual access to discontinuous, random memory addresses during parallel matrix multiplications. Therefore, enhancing the access efficiency to the nonzero elements of sparse matrices is crucial in processing sparse matrices.
— *Parallel Computing Efficiency*: Sparse matrices often exhibit significant variations in the count of nonzero elements across their rows and columns. This irregularity results in uneven **Digital Signal Processor (DSP)** loads during parallel computing, which in turn lowers the total

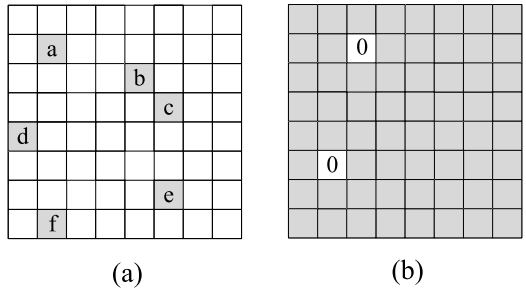

Fig. 1. (a) Sparse matrix; (b) dense matrix.

system throughput. Therefore, balancing the workload across different DSPs is a key factor in determining the efficiency parallel computational.

This article provides an overview of the **State-of-the-Art (SOTA)** research on FPGA-based SpMM accelerators, framed from the perspective of addressing these challenges. Firstly, we introduce the background and computational methods of SpMM. Then, we analyze how different FPGA-based SOTA SpMM designs handle these challenges. Finally, we discuss future development opportunities and offer potential directions for subsequent research efforts. This article aims to benefit researchers dedicated to designing FPGA-based SpMM accelerators. The contributions of this work are summarized as follows:

(1) We characterize common problems in SpMM computing and classify them into four types of challenges.
(2) We review FPGA-based SpMM accelerators, detailing how they address various challenges. Additionally, we conduct a horizontal performance comparison of these accelerators.
(3) We provide a series of potential directions for subsequent FPGA-based SpMM acceleration designs. To the best of our knowledge, this is the first comprehensive survey of FPGA-based SpMM accelerators.

The rest of this article is organized as follows. In Section 2, we provide necessary background information about SpMM and its typical computational methods. Section 3 introduces our survey and search methods. Next, in Section 4 we provide an overview of previous work in FPGA-based SpMM accelerator designs and connect them with the four challenges, and in Section 5, we make a comparison between these SOTA accelerator designs. Section 6 highlights the future directions for FPGA-based SpMM accelerators. Finally, Section 7 concludes the article.

2 Background

2.1 Preliminaries

A sparse matrix is a matrix in which most of the elements are zero. This contrasts with a dense matrix, where most of the elements are nonzero. Sparse matrices are particularly useful in areas of computational mathematics and computer science, including numerical analysis, optimization, and the representation of data or network graphs. As shown in Figure 1, the gray areas represent nonzero values, while the blank areas are all zeros, in contrast to a dense matrix.

SpMM refers to the process of multiplying two matrices in which one or both of them are sparse matrices. SpMMs are commonly encountered in practical applications. For instance, the matrix corresponding to pixel points of an image may be sparse in image processing [62]. As network analysis, due to the sparse connections between a large number of nodes, network topologies can

Table 1. Notations in SpMM

Notation	Description
A	First matrix
B	Second matrix
C	Result matrix
m	The mth row of the first matrix
n	The nth column of the second matrix
k	The kth column of the first matrix and the kth row of the second matrix
nnz	Number of nonzero values

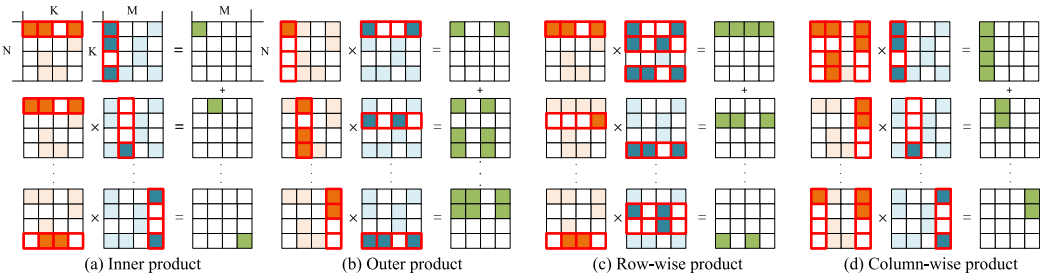

Fig. 2. Examples of four matrix multiplication methods: (a) inner product; (b) outer product; (c) row-wise product; (d) column-wise product. Nonzero elements in the two input matrices are shown in orange and blue, nonzero elements in the output matrix are shown in green, and matrix elements involved in the computation are highlighted with black borders.

be represented by sparse matrices [55]. Furthermore, in machine learning, some feature matrices also exhibit sparse characteristics [38].

The computation of SpMM follows the same principles as regular matrix multiplication, with four main methods: inner product, outer product, row-wise product, and column-wise product. Effective reuse of data reflects the efficiency of these computational methods, and there is variability in data reusability among different methods. We outline the processes of these distinct matrix multiplication methods. Table 1 summarizes the symbols used to represent sparse matrices. Moreover, Table A1 provides a reference for the main abbreviations used in this paper For simplicity, we assume that $n = m = k$. In other words, the matrices involved in the computation are n-dimensional square matrices.

The inner product is the most commonly used method for matrix multiplication. The operation involves a series of dot products between rows of the first matrix, A, and columns of the second matrix, B, to compute elements of the final product matrix, C, as shown in Equation (1):

$$C[m,n] = \sum_{k=0}^{N} A[m,k] * B[k,n]. \tag{1}$$

A single inner product calculation yields one element of the result matrix, necessitating iteration over the rows m of A and the columns n of B. As illustrated in Figure 2(a), index matching k is required to identify elements used for **Multiplication and Accumulation (MAC)**. For SpMM, effective output is generated only when there is an intersection of nonzero value indices in k.

The inner product computation in sparse matrices typically exhibits good output reusability but poor input reusability. The two input matrices require different storage formats: matrix A primarily organized by rows, and matrix B by columns. Inner product calculations involve iterating

over all elements of rows and columns, making it more suitable for dense matrices. However, for extremely sparse matrices, the iteration overhead becomes relatively high. MAC operations can only be executed with index matching, making data organization relatively complex. However, this method has lower requirements for on-chip memory.

The outer product method multiplies a column of the first matrix, A, with the corresponding row of the second matrix, B, producing a partial matrix for the result, as shown in Equation (2):

$$C[:,:] = \sum_{k=0}^{N} A[:,k] * B[k,:]. \tag{2}$$

A single outer product computation generates a complete $m \times n$ matrix. By combining k partial output matrices, the final output matrix is formed, as illustrated in Figure 2(b). Therefore, the computation is divided into two parts: multiplication operations to generate partial product matrices, followed by merging operations that accumulate these partial matrices into the final result.

Compared to the inner product, the outer product achieves good reuse of input matrices by sequentially reading A and B once, without the need for index matching between input matrices. However, as the outer product generates a large number of intermediate results, it typically requires access to off-chip memory, thus being limited by poor output reusability. The outer product is generally more advantageous for sparser input matrices. Similarly, the two input matrices usually require different storage formats, with matrix A primarily organized by columns and matrix B by rows.

The Gustavson method, which includes the row-wise product and the column-wise product (transposed case of the row-wise product) [20], is depicted in Equations (3) and (4):

$$C[m,:] = \sum_{k=0}^{N} A[m,k] * B[k,:]. \tag{3}$$

The row-wise product involves multiplying nonzero elements of a row in matrix A with corresponding nonzero elements in the same row of matrix B. The column indices of the selected row in A are matched with the row indices in B to produce intermediate rows of the result matrix. Figure 2(c) illustrates the process of the row-wise product algorithm, where a row in the result matrix C is the cumulative sum of intermediate rows. The row index of the intermediate rows is determined by the row in matrix A, thus computing the output matrix one row at a time.

The row-wise product is characterized by sequential row-oriented access for matrices, allowing the use of the same storage format for both input and output matrices. Compared to the previously mentioned methods, the row-wise product avoids extreme dataflow scenarios but has lesser reusability for individual values and lower on-chip data reuse rates. Unlike the inner product, it doesn't require hardware-intensive index matching operations between elements of input matrices. Compared to the outer product, it simplifies operations for merging partial sums in sparse matrices. As each computation outputs a single row, it requires less on-chip memory compared to the outer product.

The column-wise product, also known as the column Gustavson algorithm, differs from the row-wise product in that it multiplies all nonzero elements of a certain column in matrix B with corresponding nonzero elements in the same column of matrix A, as shown in Equation (4):

$$C[:,n] = \sum_{k=0}^{N} A[:,k] * B[k,n]. \tag{4}$$

The column indices of B are matched with the row indices of the selected column in A, and the final results are accumulated into the corresponding columns of the output matrix. The computational process is illustrated in Figure 2(d).

2.2 Applications

Efficient SpMM is crucial for many scientific applications, and it has received extensive attention over the past decade. Research related to SpMM has been conducted across multiple fields.

2.2.1 Artificial Intelligence Generative Content (AIGC). In the field of AIGC, models such as **Generative Adversarial Networks (GANs)** increasingly require more efficient SpMM computations as data volumes continue to grow [44]. For example, NVIDIA's Ampere architecture features a sparsity function that introduces structured sparsity to optimize SpMM, enhancing the performance of deep learning models in both inference and training. This improvement provides higher efficiency in tasks such as image and video generation [46]. Specifically, NVIDIA has implemented support for 2:4 structured sparsity on its Ampere GPUs, allowing two out of every four elements to be 0. This approach significantly reduces the computational burden while leveraging the hardware acceleration provided by Sparse Tensor Cores, thereby enhancing the computational efficiency of generative models such as GANs [52].

2.2.2 Natural Language Processing (NLP). In the field of NLP, Transformers are the most popular models whose self-attention mechanisms often generate large sparse matrices when handling cross-sequence dependencies [39]. Accelerating the multiplication of these sparse matrices is particularly crucial. Huawei's Ascend AI processor, especially the Ascend 910, has significantly improved the efficiency of Transformers in tasks such as machine translation and speech recognition by optimizing sparse matrix operations. This enhancement has promoted the widespread adoption and application of Transformers in various fields [68]. NVIDIA's Hopper architecture has further accelerated related computations by adding the FP8 Transformer Engine to its Sparse Tensor Cores [52].

2.2.3 Convolutional Neural Networks (CNNs). In computer vision, CNNs often process large amounts of image data, which become sparse after feature extraction and multiple layers of processing [31]. SpMM accelerators can leverage this sparsity to enhance computational speed. Google's Tensor Processing Unit-like architecture [21] has demonstrated significant advantages in CNN model computations by accelerating SpMM, thereby improving inference and training efficiency [30].

2.2.4 Recommendation Systems. In the field of recommendation systems, **Graph Neural Network (GNN)** models rely on graph-structured data, often involving sparse matrix operations during processing. NVIDIA's **Deep Graph Library (DGL)** is a deep learning framework specifically designed for GNNs, supporting various types of GNN models [63]. DGL provides specialized API functions to encapsulate SpMM operations, optimizing the storage and manipulation of sparse matrices internally. By integrating efficient SpMM acceleration techniques, DGL enhances the computational efficiency of neural networks on graphs. This is particularly beneficial for applications such as large-scale social networks and recommendation systems, where SpMM accelerators can significantly boost the training and inference speed of GNN models.

3 Survey Methodology

To systematically review and analyze the latest research in the field of SpMM, we conduct a detailed literature search and selection process. Given the rapid development in HPC and hardware-based accelerated computation, numerous new methods and technologies are proposed each year.

Table 2. Search Strategy

	Search engines	Search strategy
Initial search	IEEE Xplore	(“ALL Metadata”: SpMM) OR (“ALL Metadata”: SpMV) AND (“ALL Metadata”: FPGA) Filters Applied: 2020–2024
	ACM digital libraries	[All: SpMM] OR [All: SpMV] AND [All: FPGA] AND [E-Publication Date: (01 January 2020 TO 30 May 2024)]
	Springer Link	(SpMM) OR (SpMV) AND (FPGA) AND Custom dates 2020–2024
	arXiv	(All fields: SpMM) OR (All fields: SpMV) AND (All fields: FPGA) AND date_range: from 2020 to 2024
Update search	IEEE Xplore	(“All Metadata”: sparse matrix multipli) AND (“All Metadata”: FPGA) AND Filters Applied: 2020–2024
	ACM digital libraries	[All: “sparse matrix multipli”] AND [All: FPGA] AND [E-Publication Date: (01 January 2020 TO 30 May 2024)]
	Springer Link	(sparse matrix multipli) AND (FPGA) AND Custom dates 2020–2024
	arXiv	(ALL fields: sparse matrix multipli) AND (ALL fields: FPGA) AND date_range: from 2020 to 2024
Additional search	Google Scholar	Related citations Academic profiles of frequently cited authors

Therefore, we choose to include literature from 2020 onward to ensure that all references reflect the latest technological advancements and research findings. This approach not only ensures that the review content is timely and relevant but also maximizes the utilization of recent innovations.

We use the snowball search strategy, conducted in three rounds as shown in Table 2. The initial round was performed in the following major digital libraries: IEEE Xplore, ACM Digital Libraries, Springer Link, and arXiv. It is important to note that, although the research quality on arXiv varies, it contains many cutting-edge studies, so we do not overlook it. On these datasets, we used “SpMM” and “SpMV” as the main keywords for search. However, initial results indicated that these keywords were not ideal, as we only obtained approximately 13 relevant articles. This is because many related papers do not explicitly tag these abbreviations as keywords. Therefore, we’ve adjusted our retrieval strategy to use the more general keyword “sparse matrix multipli” for new searches, accounting for different word forms and expressions by specifically inputting “multipli.” This change allowed us to retrieve over 350 relevant articles. After retrieving the set of literature, we used Google Scholar to analyze the references and citation lists of the retrieved papers, further expanding the literature set step by step. Additionally, we focused on the academic profiles of frequently cited authors within the literature set to supplement and enhance our database. Through these steps, we have

Table 3. Comparison of the Characteristics of Different SpMM Methods

Methods	Intermediate results	Data reusability	On-chip storage	Index matching
Inner product	An element of the result matrix	Output data reuse	Minimum	Strong
Outer product	Partial matrix	Input data reuse	Maximum	Weak
Row-wise product	A row of the result matrix	Medium	Medium	Medium
Column-wise product	A column of the result matrix	Medium	Medium	Medium

obtained a total of 481 articles. Finally, we have made a filtering, as we aim to present the latest and most significant research in this field. We focus on top-tier computer architecture conferences (ISCA, MICRO, HPCA, ASPLOS), FPGA conferences (FPGA, FPL, FCCM, FPT), electronic design automation conferences (DAC, DATE, ICCAD, ASP-DAC), and high-impact journals (TC, TCAD, TRETS, TVLSI, and so on) because these publications represent the forefront of this field [19]. After all these steps, we discussed a total of 26 FPGA-based SpMM designs in this survey.

4 Existing FPGA-Based SpMM Accelerator to Handle Three Challenges

4.1 Dataflow Selection

For SpMM, the inner product, outer product, and Gustavson algorithms are different methods used to perform multiplication computations. Their main differences lie in how they handle the arrangement and access of nonzero elements, that affect the number of memory accesses and the speed (parallelism) of computation in SpMM. It is crucial to select an appropriate computation method based on the sparsity, matrix size, and hardware resource limitation. Table 3 shows the characteristics of different SpMM methods.

4.1.1 The Inner Product. As introduced in Section 2, the inner product method performs computations on each rows of the input matrix A with the column of matrix B. Its parallel computation process can almost be considered as SpMV. For sparse matrices, only computations on nonzero elements are effective and contribute to the final result. Additionally, the irregular distribution of nonzero elements in sparse rows can cause load imbalance issues during parallel computation. Therefore, designers need to pay attention to both matrix storage and load imbalance.

Serpens [57] targets SpMV computation by leveraging the high bandwidth and multichannel characteristics of HBM to design efficient data access mechanism. By replicating sparse elements and dense vectors multiple times, they avoid read conflicts and enable inter-row parallel computation. They achieve load balancing by reordering non-zero elements. Graph-OPU [4] addresses SpMM inner product computation by proposing an optimized **Packet-Level Column-Only Coordinate (PCOO)** format that unifies the storage format for sparse data. Similar to Serpens, Graph-OPU utilizes HBM's high bandwidth and multichannel characteristics, replicating sparse elements and dense vectors multiple times to avoid read conflicts and achieve inter-row parallel computation. To tackle load imbalance issues, Graph-OPU employs a pipelined parallel design and an adder tree design with flexible data paths to mitigate load imbalance. SkeletonGCN [64] also introduces an improved compression storage format, **Compact PCOO (CPCOO)**, which reduces storage overhead. Additionally, SkeletonGCN features a data distribution module to control the dataflow required by the **Multiply-Accumulate (MACC)** array, enabling efficient row data reads from the left matrix. SkeletonGCN addresses load imbalance through a ping-pong buffer design, ultimately enhancing DSP efficiency.

4.1.2 The Outer Product. For the outer product computation, each column of matrix A and each row of matrix B computation results in a portion of the resulting matrix. Therefore, the outer product requires new space to store these intermediate results. This is particularly significant

when dealing with large-scale data, as substantial storage resources are needed to buffer these intermediate results.

Due to the limited on-chip memory resources, FPGA-based SpMM accelerator design with the outer product computation method is not common. LW-GCN [60] proposes an innovative compression format called PCOO, which unifies data storage. Additionally, LW-GCN designs a multiport memory system and reduces data conflicts further through data replication and row grouping. By employing tiling techniques, it processes a portion of SpMM at one time. Sextans [58] balances the load among PEs by partitioning matrix A. It fully leverages the large capacity and data bandwidth advantages of HBM. Sextans designs an HBM access module paired with a fully pipelined computing unit to avoid the limitations of storage resources for intermediate results.

4.1.3 Gustavson. Gustavson is one with significant parallel computing potential, as it allows for the parallel computation of multiple rows within matrix. Each computation in Gustavson results in row or column, so the required output buffer size is approximately the size of one row or column. However, the irregular distribution of nonzero elements leads memory access conflicts. Additionally, it also can lead to workload imbalances among the computational units.

Currently, several FPGA-based accelerators accelerate SpMM with Gustavson. To address memory access conflicts, Li et al. [33, 36] proposed a novel access pattern-aware cache scheme called SpCache, executing the Gustavson algorithm in an element-parallel manner. Gao et al. [16] achieved load balancing by partitioning sparse data equally and proposed vertex clustering optimization to reduce global data transfers. Building on this research, Gao et al. [16] implemented an adaptive dataflow scheduling strategy based on the Gustavson algorithm to mitigate sparsity and explore regular parallelism during computation. FSpGEMM [61] employs the Gustavson algorithm to avoid zero output calculations and reduce the synchronization overhead of computing partial products. They also customized a buffering scheme tailored for the Gustavson algorithm to enhance the reuse of input matrices, thereby reducing off-chip memory accesses. Dynasparse [69] proposed an innovative unified accelerator design on FPGA, leveraging dynamic sparsity to accelerate GNN inference. When computing under SpMM mod, Gustavson's computation can efficiently skip zero elements in both input matrices.

4.1.4 Summary. As the above introduction and review, the design challenges corresponding to dataflow selection can be classified into Matrix Storage Efficiency, Memory Access Efficiency, and Parallel Computing Efficiency. In other words, these challenges are intertwined with dataflow selection. Therefore, in the following sections, we will provide a more detailed introduction to the efforts made by previous research to address these three challenges.

4.2 Matrix Storage Efficiency

4.2.1 General Sparse Matrix. In sparse matrices, where a majority of elements are zero, storing them in the same manner as dense matrices results in unnecessary memory resource consumption. Utilizing standard compression formats (such as **Coordinate (COO)**, CSR, and **Compressed Sparse Columns (CSC)**) enhances memory efficiency. Additionally, to address both compression and access efficiency [56], recent years have seen the development of novel compression formats, tailored to meet diverse computational requirements and hardware platforms. Following, we delve into several general sparse matrix compression formats. Figure 3 exemplifies these formats using matrix A, providing a detailed introduction to common compression methodologies in the context of sparse matrices.

COO: COO [53] is a straightforward compression format that stores nonzero elements by their coordinates. As shown in Figure 3(a), it uses three arrays: *val*, *row*, and *col*, to store the values, row indices, and column indices of nonzero elements in a sparse matrix, respectively. In the COO, the

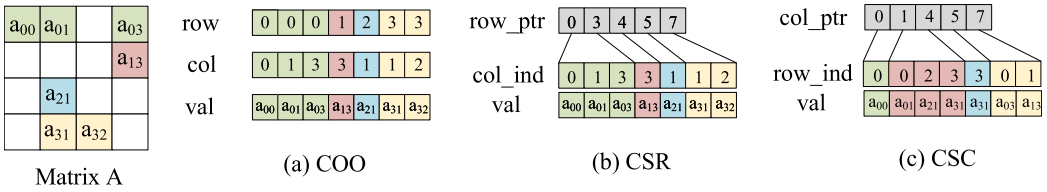

Fig. 3. Three common compression formats of sparse matrices: (a) COO; (b) CSR; (c) CSC.

corresponding three array elements can be obtained without additional operations or sorting of nonzero elements. However, COO may lead to increased memory consumption for matrices with a lower density of non-zero elements.

CSR: CSR [53] is the most commonly employed compression format and optimizes matrices by compressing them row-wise. It also consists of three arrays: *val*, *col_ind*, and *row_ptr*. The *val* stores nonzero values in a row-major sequence, with its size equal to the number of non-zero elements (*nnz*). The *col_ind* indicates the column indices of these nonzero elements, also sized to *nnz*. The *row_ptr*, crucial for row demarcation, points to the start and end of each row in the matrix. For a matrix of size $N \times N$, *row_ptr* has $N + 1$ elements, where the count of nonzero elements in the ith row can be determined by $row_{ptr(i+1)} - row_{prt(i)}$. Figure 3(b) illustrates an example employing the CSR format. Relative to COO, CSR format offers reduced memory consumption for storing. Yet, its efficiency drops notably when column-wise traversal or random access of matrix elements is required. This necessitates the integration of additional data lookup mechanisms to identify nonzero elements' positions. Consequently, employing CSR involves weighing the tradeoff between memory conservation and the extra resource expenditure involved.

CSC: CSC [53] format compresses the matrix column-wise and also comprises three arrays: *val*, *col_ptr*, and *row_ind*, as shown in Figure 3(c). The *val* represents the values of nonzero elements, the col_ptr indicates the start and end pointers of each column, and the *row_ind* corresponds to the row indices of these nonzero elements, aligned with the values in the *val* array. The CSC format simplifies matrix operations involving column-wise actions, offering direct access to the respective columns. However, its efficiency is lower when it comes to row-wise operations.

In addition to the aforementioned common compression formats, specialized formats designed for FPGA-based accelerators have been proposed. These formats cater to various types of sparse matrices and architectural designs, differing in memory consumption and accessing efficiency.

C^2SR: Hosseinabady and Nunez-Yanez [23] introduced an improved compression format based on CSR, termed modified CSR, "removing" (MCSR). This format utilizes the count of nonzero elements per row instead of row pointers to indicate the row information of nonzero elements. This concept is similar to the C^2SR format [59]. For clarity and to distinguish from other compression formats introduced later, this article refers to it as C^2SR. The format comprises three arrays: *val*, *col_ind*, and *row_length*, as shown in Figure 4. *val* and *col_ind* store the nonzero values and column indices of the sparse matrix, respectively, while *row_length* records the number of nonzero values in each row.

MCSR: MCSR [51] is specifically designed for SpMV on multicore hardware accelerators. Its uniqueness lies in compressing not only the nonzero elements of the sparse matrix but also the elements of the vector to be multiplied. MCSR is represented by three arrays that store relevant data of the sparse matrix and the vector: $MatrixValue$, $FinishFlag$, and $VectorValue$. The $MatrixValue$ array stores the nonzero values of the matrix, the $FinishFlag$ indicates whether an element is the last nonzero element in its row, and the $VectorValue$ array contains the corresponding elements of the vector to be multiplied, as shown in Figure 5(a). The sequence of vector elements stored in the

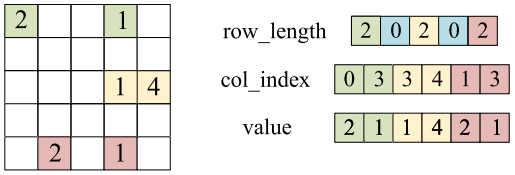

Fig. 4. C^2SR format of the sparse matrix.

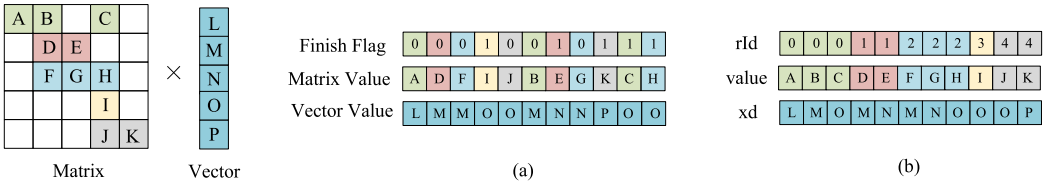

Fig. 5. (a) MCSR format of the sparse matrix; (b) ReMCOO format of the sparse matrix.

VectorValue array aligns with the column indices of nonzero elements. MCSR's storage pattern is distinct from row-wise or column-wise. It begins by storing the first nonzero element of each row, iteratively assessing all rows in a manner where only one nonzero element per row is stored per iteration. This access continues until every row's nonzero elements are included in the array.

Redundant Modified Coordinate (ReMCOO): ReMCOO [43], targeting SpMV implemented on FPGAs, introduces a novel format to represent sparse matrices and vectors for efficient SpMV. ReMCOO is represented by three arrays, as illustrated in Figure 5(b). The *rId* array holds the row indices of nonzero elements in matrix A, *value* stores the nonzero values of matrix A, and *xd* contains elements of the input vector x pointed to by the column indices of nonzero elements in the sparse matrix A. The length of each array is equal and corresponds to the number of nonzero elements in the respective sparse matrix.

PCOO-List: PCOO [60] introduces the concept of data packets into data compression formats. As shown in Figure 6(a), the PCOO format organizes each row as a unit, treating all nonzero elements in a row as a data packet. The first three bits act as the packet header: the first two bits specify the row information for each non-zero element, indicating the start of row (*sor*) for the first nonzero element and the end of row (*eor*) for the last, while the third bit, the valid bit (*vld*), indicates whether the element is an injected null element. The remaining bits contain column information (*col*) and the values of each nonzero element (*val*), with log_2N bits for the column position and the remaining H bits for the value. If a row has no nonzero elements, header bits are set to $sor = eor = 1$ and $vld = 0$. Thus, representing each nonzero element in a sparse matrix requires $3 + log_2N + H$ bits.

Optimized PCOO (OPCOO): OPCOO [4] is an improved format of PCOO. It leverages the implicit row index information during matrix multiplication access and incorporates an OPCOO format to further compress the storage of sparse matrices. As illustrated in Figure 6(b), OPCOO retains only the *eor* group, which indicates the end of a row, the *col* group for column indices, and the *val* group for storing nonzero values.

CPCOO: CPCOO [64] is also an improvement on the PCOO format, primarily focusing on eliminating redundant information to further compress sparse matrices. As shown in Figure 6(c), CPCOO retains the same representation as PCOO for rows with nonzero elements, but it divides into two parts: the header and the main body. For rows that contain at least one nonzero element, the encoder stores *vld*, the *sor*, and the *eor* markers in the Header section. These markers define the storage range of nonzero elements in the Body section. For rows with all zero elements, the

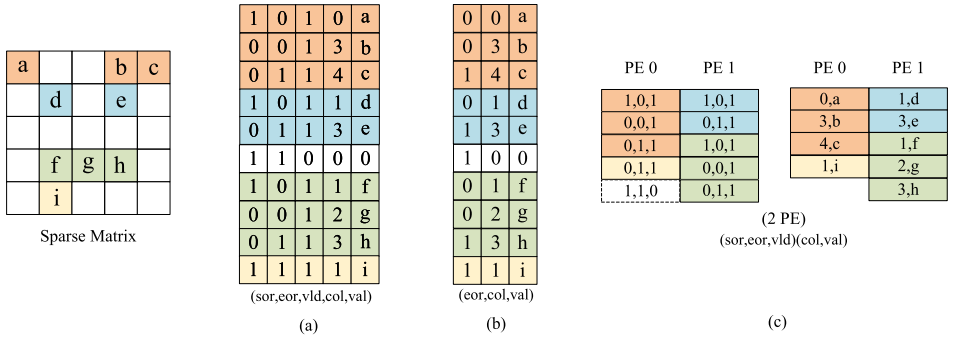

Fig. 6. (a) PCOO format of the sparse matrix; (b) OPCOO format of the sparse matrix; (c) CPCOO format of the sparse matrix.

CPCOO format sets the vld bit to 0, indicating that the Body section does not store any information for these rows, as empty rows do not contribute to matrix computations.

Compressed Variable-Length Bit-Vector (CVBV): Kestur et al. [32] proposed three compression methods for sparse matrices: **Bit-Vector (BV)**, **Compressed Bit-Vector (CBV)**, and CVBV. Among these, the CVBV format is an optimization of BV and CBV, with its core principle being the use of variable-length encoding schemes to represent sequences of consecutive zero or nonzero values in the matrix. This method leverages the fine-grained bit manipulation capabilities of FPGAs. In the CVBV format, variable-length data fields are used to store run-length encoding. Each sequence of consecutive zero or nonzero values is preceded by a 4-bit header. The first bit of this header indicates whether the sequence is zero (0) or nonzero (1). The next three bits specify how many nibbles (4-bit units) are used to store the count, allowing for a maximum of 4 bytes.

4.2.2 Symmetric Sparse Matrix. With the growing prevalence of graph structure data processing, attention on symmetric sparse matrices originating from undirected graphs has significantly increased among the research community [11]. Symmetric sparse matrices, as depicted in Figure 7(a), are characterized by diagonal symmetry, where matrix elements exhibit equal values on either side of the diagonal. Owing to their unique configuration, it's possible to store only nonzero elements of either the upper or lower triangular sub-matrix, effectively downsizing the overall matrix. Therefore, beyond the compression formats already discussed, there are specifically tailored formats for symmetric sparse matrices, aimed at optimizing memory usage.

Symmetric Sparse Skyline (SSS): SSS [17] is a compression format specifically designed for symmetric sparse matrices. It stores only the lower triangular sub-matrix and its main diagonal, effectively reducing the matrix size by approximately half. As illustrated in Figure 7(b), the SSS format stores the main diagonal elements of the matrix in the *dvalues* array, with a size of N, corresponding to the matrix's dimension. Notably, zero elements on the diagonal are retained in the array alongside nonzero elements. The standard CSR format is used to store the remaining nonzero elements of the lower triangular matrix, including arrays *rowptr*, *colind*, and *value*.

SCSR: SCSR [5] is a data format tailored for **Symmetric Sparse Matrix-Vector Multiplication (SSpMV)** computations. SCSR combines the advantage of SSS and C^2SR, optimizing to reduce memory access load. Different from SSS, SCSR is similar to C^2SR, using only three arrays to represent matrix data: values, column indices, and row lengths. As depicted in Figure 7(c), the *value* array in SCSR includes only the nonzero elements from the upper triangle and the main diagonal. The

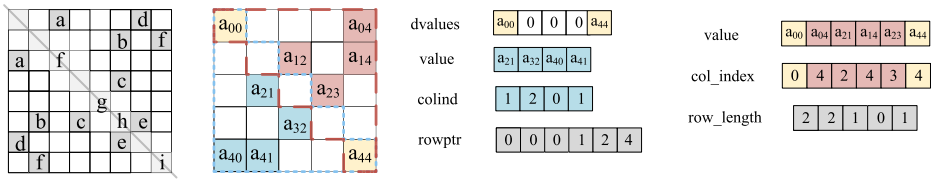

Fig. 7. (a) Symmetric sparse matrix; (b) SSS format of the symmetric sparse matrix; (c) SCSR format of the symmetric sparse matrix.

Table 4. Compression Ratio Formula for Different Compression Formats

Compression format	Compression ratio	Data types	Index types
COO [53]	(3×32nnz)/(32n×n)	32-bit	32-bit
CSR [53]	[2×32nnz + 32(n+1)]/(32n×n)	32-bit	32-bit
CSC [53]	[2×32nnz + 32(n+1)]/(32n×n)	32-bit	32-bit
C^2SR [23]	[2×32nnz + 32n]/(32n×n)	32-bit	32-bit
MCSR [51]	(nnz + 18nnz)/(18n×n)	18-bit (fixed-point)	1-bit
ReMCOO [43]	(2×32nnz)/(32n×n)	32-bit	32-bit
PCOO [60]	$[3 \times (nnz + n_{void}) + \log_2 n \times (nnz + n_{void}) + 16(nnz + n_{void})]/(16n \times n)$	16-bit	$\log_2 n$ bit/1-bit
OPCOO [4]	$[(nnz + n_{void}) + 31(nnz + n_{void}) + 32(nnz + n_{void})]/(32n \times n)$	32-bit	31-bit/1-bit
CPCOO [64]	$[3\times(nnz + n_{void}) + \log_2 n \times nnz + 16nnz]/(16n \times n)$	16-bit	$\log_2 n$ bit/1-bit
SSS [17]	$[32\times n + 2\times 32nnz_{down} + 32(n+1)]/(32n \times n)$	32-bit	32-bit
SCSR [5]	$[2\times 32(nnz_{up} + nnz_{diag}) + 32\times n]/(32n \times n)$	32-bit	32-bit

col_index and *row_length,* respectively, maintain the column indices and the quantity of nonzero elements per row.

4.2.3 Comparison. We compare the compression ratios of the various compression formats introduced in this section, as described by

$$Compression\_Ratio = \frac{Compressed\_Size}{Uncompressed\_Size}, \tag{5}$$

whereas *Size* is denoted by

$$Size = Number\_of\_Data \times Data\_Width. \tag{6}$$

Table 4 summarizes the formulas for compression ratios corresponding to different compression formats. It should be noted that we consider all matrices as square matrices with dimensions of n × n. The n_{void} represents the data required for padding in the compression format, such as the zeros in PCOO.

To intuitively compare the compression rates of various formats, we conduct tests on the University of Florida's sparse matrix dataset [11] using the formula in Table 4. The test matrices include both general and symmetric sparse matrices, with dimensions ranging from 10 to 50,000 and data densities varying from 1%–30%. For fairness in comparison, we standardize the bit width of the original sparse matrices to match the compression formats. Due to the wide range of sparsity in the original matrices, we apply a logarithmic function to compress the scale of the compression rates, making analysis and comparison more manageable.

Figure 8 shows the compression rates of different formats applied to general sparse matrices, consistent with their theoretical formula representations. For both MCSR and ReMCOO, the compression formats are designed for the sparse matrices in SpMV. The commonality is that they both compress the elements of the vector involved in the multiplication along with the nonzero

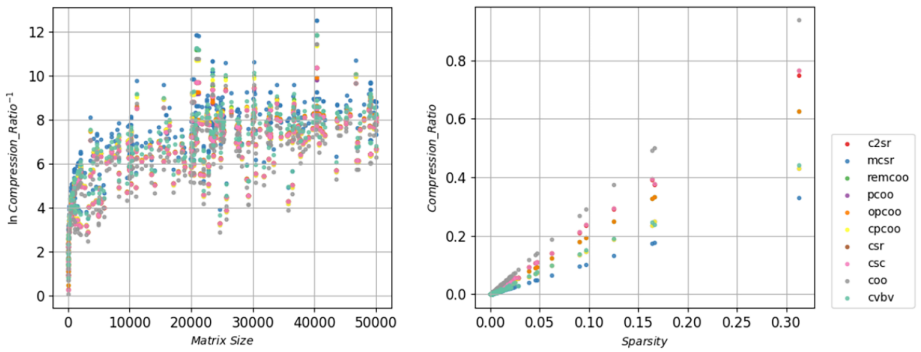

Fig. 8. Comparison of compression rates of different compression formats: treatment of ordinary sparse matrices.

elements of the matrix. However, to fairly compare compression ratios with other formats, we only account for the compression of the sparse matrix portion and do not include the memory usage of the vector part in our calculations. Among these, MCSR is the most efficient due to its transformation of the original matrix into just two arrays: Matrix Value and Finish Flag. The Finish Flag, indicating the row-end marker, has a bit width of only 1 bit. However, this comes at the cost of requiring an additional scheduling algorithm, such as Hu's algorithm [25], to optimize the arrangement of nonzero elements in the sparse matrix. This method alters the original matrix's storage in memory, thereby adding an unavoidable time-consuming scheduling phase. In contrast, the COO format is the least efficient as it requires storing both row and column indices. The overlap between CSR and CSC is due to the fact that the difference between these two formats lies only in the order of storage, resulting in the same memory consumption. PCOO and OPCOO perform similarly because the difference between these formats lies mainly in the storage of inserted null elements, which has a negligible impact. We also observe that as the sparsity of the matrices increases, the compression rates of different formats gradually decrease. This indirectly indicates that for matrices with higher sparsity, saving memory through matrix compression becomes more efficient.

For symmetric sparse matrices, as shown in Figure 9, their dedicated compression formats, SSS and SCSR, outperform the general compression formats. Among these, SCSR is more efficient due to its more effective handling of the diagonal elements in symmetric sparse matrices.

4.3 Memory Access Efficiency

FPGAs, with their constrained resources, necessitate efficient access to nonzero data in SpMM, aiming to optimally utilize available memory bandwidth. The inherent sparsity of matrices, particularly in large-scale instances, results in an irregular distribution of nonzero data [6], posing challenges to memory access efficiency. Appropriately leveraging data reuse becomes crucial in enhancing computational efficacy. Therefore, to maximize the use of memory bandwidth on FPGAs, memory access strategies that harness data reuse are commonly devised, significantly improving the efficiency of memory access.

The processing of SpMM typically involves random access to three matrices: two matrices to be multiplied (denoted as matrix A and matrix B) and the resultant matrix (denoted as matrix C). Given the immense dimensions of sparse matrices, it's impractical to store all matrix data in on-chip memory, even when compressed formats are used. Consequently, off-chip memory is used for storage. However, issuing numerous irregular accesses to off-chip memory is highly inefficient.

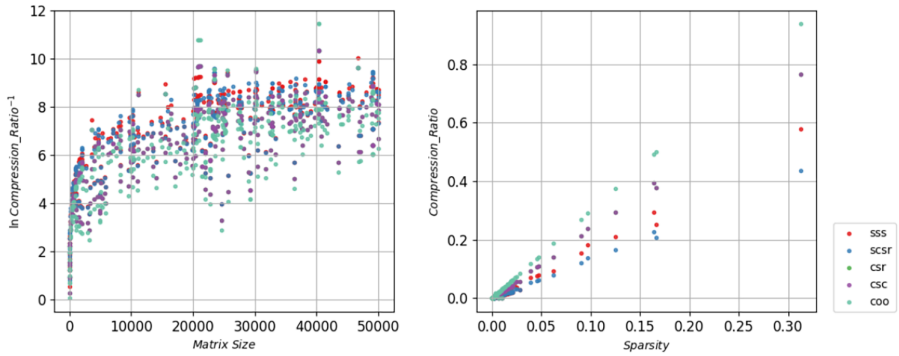

Fig. 9. Comparison of compression rates of different compression formats: treatment of symmetric sparse matrices.

The following section introduces various design modules or strategies aimed at enhancing memory access efficiency. These include architectures specifically developed for accelerating SpMM, as well as those related to GNN accelerators, where SpMM is a crucial kernel.

4.3.1 Optimized Cache Design. Li et al. [33, 36] observe that traditional caching mechanisms adversely affect SpMM performance. To address this, they introduce SpCache, a specialized cache design. SpCache expedites responses to irregular requests by prefetching off-chip memory accesses, mitigating conflicts caused by simultaneous accesses to the same memory bank. It consists of a banked cache and four cache managers. When a request arrives, a cache manager queries the targeted bank. SpCache checks for a cache hit in one cycle; if it's a miss, it initiates off-chip memory access, fetching data into the cache bank and the target component. If it's a hit, the response is simply to the cache manager. Additionally, it implements strategies to ensure sequential access and caching of elements in matrix B, effectively handling sequences with unpredictable access lengths. By reducing memory access conflicts from irregular accesses, SpCache significantly enhances overall performance.

Tavakoli et al. [61] propose an OpenCL-based HPC framework, FSpGEMM, geared toward accelerating FPGA-based SpGEMM. FSpGEMM utilizes the Gustavson algorithm, effectively bypassing zero-output calculations and minimizing synchronization costs for computing partial products. To tackle the irregular memory access inherent in sparse matrices, FSpGEMM introduces a tailored buffering strategy optimized for the Gustavson algorithm, thereby enhancing the reuse of input matrices. Unlike traditional sparse matrix compression formats like SCR and CSR, FSpGEMM adopts a vector-major compression sequence. Its **Compressed Sparse Vector (CSV)** format, illustrated in Figure 10(a), aligns vector lengths with the quantity of computation units, ensuring consistent off-chip memory access. This approach facilitates parallel processing of multiple rows from the first input matrix and concurrent sharing of rows from the second input matrix. As shown in Figure 10(b), substantially reducing the amount of off-chip memory access is required.

Liu and Liu [40] propose an FPGA-based high-bandwidth-utilization SpMV accelerator, featuring a novel memory architecture to address irregular memory accesses caused by compressed data. To navigate the constraints of on-chip resources, they partition large matrices and vectors into blocks and segments, breaking down the entire SpMV process into multiple batches. A read-conflict-free buffer with two sub-buffers stores identical vector elements, each buffer containing four **Block RAMs (BRAMs)** and two 4-to-1 multiplexers to prevent read port conflicts from various PEs. They also design a write-conflict-free adder tree, where multiplication results pass through a 2-input

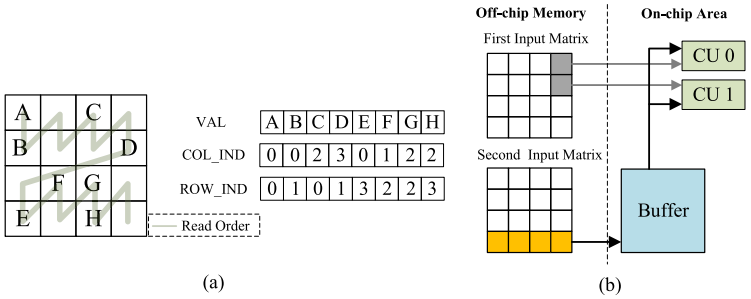

Fig. 10. The design within [61]: (a) sparse matrix representation using CSV format; (b) example of proposed data buffering scheme.

MUX, then a pipelined adder sequentially accumulates the results. A crossbar switch selects outputs to avoid write conflicts from nonzero elements in the same row. Additionally, they use a ping-pong accumulator, composed of several registers and adders, to calculate partial sums per row. Three registers facilitate functional switching to store partial sums from the previous batch, accumulate the current batch, and load for the next batch, masking the overhead of loading and storing. This approach of a read-conflict-free buffer and write-conflict-free adder tree effectively eliminates and masks delays caused by irregular vector accesses.

4.3.2 Preprocessing Design. To enhance access efficiency, data preprocessing is a crucial approach. A typical preprocessing method involves reorganizing and rearranging input data. The objective of data reordering is to cluster adjacent sparse elements together, thereby reducing the number of memory accesses. This maximizes memory access efficiency, minimizes data transfer latency, and fully leverages the parallelism inherent in FPGAs. The specific implementation methods and processing strategies vary depending on the design architecture and application context.

Li et al. [34] propose an algorithm for rearranging nonzero element data to facilitate data reuse. They configure the system to transfer four nonzero elements from off-chip memory to the FPGA chip each cycle, with vector elements stored on-chip, allowing access to up to two vector elements per cycle. The algorithm follows three rules: Rule 1 (Intra-Group Reuse): Arrange reusable columns within the same cycle as much as possible, minimizing repeated accesses to the same vector elements across different cycles. This also considers prioritizing rows with a higher count of nonzero elements to maximize data reuse and fully utilize memory ports. Rule 2 (Inter-Group Reuse): Vector elements obtained can be reused across different cycles. Elements that are reused are accessed before their reuse, enabling omission of memory requests for these data. Rule 3: Rows with a higher count of nonzero elements are prioritized for earlier computation.

Li et al. [35], building on data rearrangement, introduce **Data Reuse-Aware Compression (DRC)** format to further utilize limited memory bandwidth. Since reusable vector values are known before accelerator initiation, compressing the column indices of these vector values is feasible. As shown in Figure 11, N and M represent the counts of reusable vector values and vector values to be fetched, respectively, with a maximum of two vector values retrievable per cycle. The Val holds the N+M nonzero matrix element values in the group, *Rid* stores the row indices for these N+M elements in the group, *Cid* contains the M memory addresses for the vector values to fetch, and *Map* maintains the N+M mappings between matrix elements and vector values, where 0 indicates pairing with the first unbuffered vector value and 1 with the second. The DRC's implementation enables further use of data reuse and a fast format conversion algorithm to reduce preprocessing time.

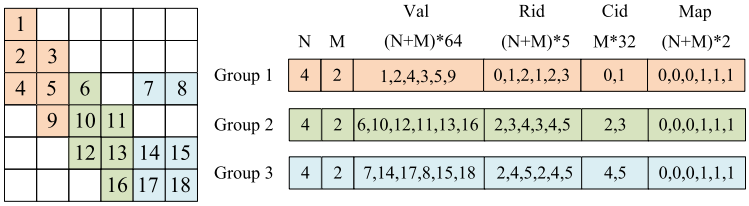

Fig. 11. DRC format within [35].

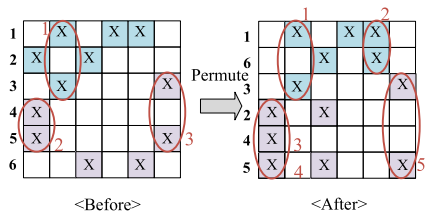

Fig. 12. Example of vertex clustering optimization within [16].

The adjacency matrices' high sparsity leads to increased frequency of random memory accesses during GNN with SpMM. Gao et al. [16] propose a vertex clustering optimization method to enhance data reusability through reordering sparse matrices. This is viable because the processing order of graph vertices is irrelevant to the final outcomes in GNN models. Their method reorganizes data by swapping rows of coefficients, as shown in the adjacency matrix figure. It is divided into tiles with equal nonzero elements, each represented by different colors. The areas circled in red indicate opportunities for data reuse. Figure 12 illustrates that after vertex clustering optimization, the reuse rate of nonzero elements is significantly improved.

Spaghetti [22] is a computation architecture based on outer product operations. To address the poor output reusability in outer product computations, the authors propose a scheduling algorithm that partitions the input matrix into blocks. This involves dividing part of the matrix into independent data streams while retaining some parts on-chip for complete merging, thereby enabling increased parallelism. Figure 13 illustrates an example of pattern-aware scheduling, with two key objectives: Matrix Tiling: Considering on-chip memory and the depth of sort-merge, the matrix A is divided into horizontal tiles to maximize reuse of partial matrices. Matrix Segmentation: Each tile is further divided into vertical segments scheduled across multipliers, minimizing idle cycles in mapping nonzero values to multipliers. Figure 13 shows the corresponding hardware microarchitecture, where a group of multipliers is connected to two buffers at the DRAM stream interface for data transfer. The scheduler calculates the starting addresses of input data and controls the computation units. The storage formats for input data vary: matrix A uses CSC and matrix B employs CSR, providing row and column indices, respectively, while the output matrix adopts the COO format. Virtual channels are used to reduce the number of stalls at the allocator's input end. The allocator distributes the unsorted data stream produced by multiplication to different sort-merge units in a cyclical fashion, parallelizing the sort-merge phase. The sort-merge units implement a streaming sort method, generating one sorted data element per cycle and merging and accumulating elements with the same COOs, achieving a fully pipelined design.

4.3.3 HBM-Based Design. When conducting SpMM on FPGAs, the extensive size of matrices often prevents their complete on-chip storage. The data bandwidth limitations of conventional off-chip DDR memory hinder the throughput of SpMM accelerators. To tackle this challenge,

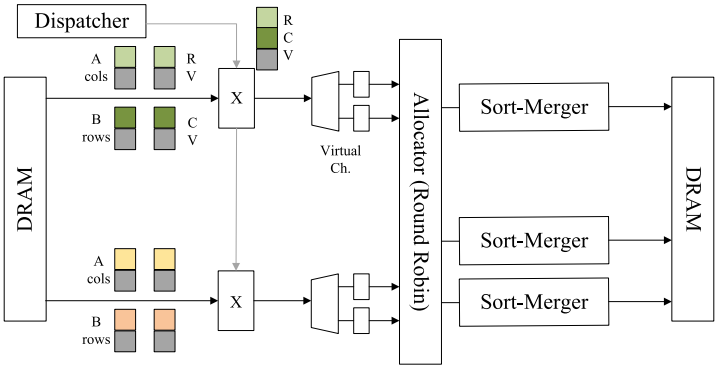

Fig. 13. The overall microarchitecture of Spaghetti [22].

some designs incorporate FPGAs with HBM, introducing specific optimizations to fully exploit the high bandwidth of HBM. These optimizations include the following: (1) Partitioning the matrix into several sub-matrices, followed by simultaneously loading each sub-matrix onto the FPGA for buffering and computation. (2) Designing customized data packaging for HBM to effectively mitigate memory access bottlenecks. (3) Implementing a modular design to enhance data processing and management efficiency and flexibility.

Sextans [58] represents the first application case integrating HBM within FPGA and proposes the partitioning of the three matrices that need to be accessed. Initially, matrix B undergoes row-wise division into smaller sub-matrices, denoted as B_i, with a predetermined window size of K_0. This is followed by the column-wise segmentation of each Bi into smaller sections, referred to as B_{ij}. Since the columns of Bi correspond with the rows of matrix A, a similar window size is utilized to divide matrix A's rows into sub-matrices A_j. This partitioning technique simplifies the processing of large matrices by breaking them down into more manageable window-sized computational tasks. Sextans also introduces a method to distribute nonzero points uniformly by dividing the zero points in A_j into P bins, where each bin p contains nonzero values that comply with the condition (row mod P) == p. Figure 14(b) illustrates how matrix A is partitioned, with an example where $p = 2$, and highlights the compression of both row and column indices. This approach facilitates sequential processing in manageable window-sized segments and optimizes random memory read and write operations within these specific windows, enhancing efficiency in the fast on-chip memory. All sub-matrices are stored in HBM, eliminating the need to access individual matrix elements. Instead, only sub-matrices are read or written, allowing HBM to facilitate streaming transfers for efficient access. Figure 14(c) shows the overall architecture of Sextans, including eight sets of **Processing Engine Groups (PEGs)**, along with modules for data access and processing computational results. Each PEG contains eight Processing Engines, operating concurrently. Initially, eight Read A modules stream the partitioned matrix A from HBM to the PEGs. Simultaneously, the Read B module streams the B_{ij} window of matrix B located in HBM. The Read Ptr module can schedule unordered nonzero values for delivery to the PEGs. After completing multiplication operations, the Collect C module gathers disjoint intermediate results, which are then sent to the Comp C module. Finally, the Read C and Write C modules facilitate data access for matrix C, concluding the final accumulation calculations. Sextans' efficient exploitation of HBM, coupled with specialized optimizations for memory access strategies, enhances memory access efficiency and addresses the challenges associated with off-chip access to large matrices.

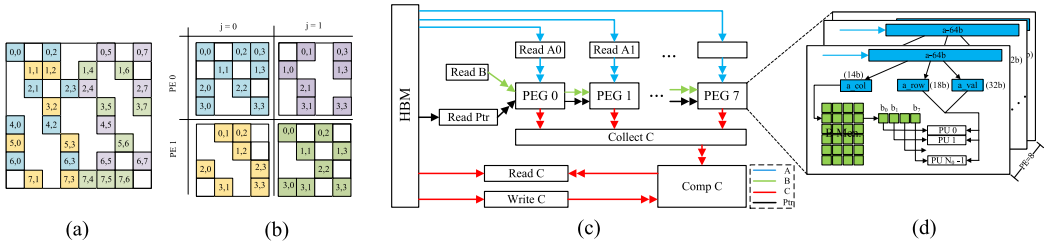

Fig. 14. The design within Sextans [58]: (a) examples of sparse matrices; (b) sparse matrix partitioning; (c) overall architecture; (d) architecture of PEs.

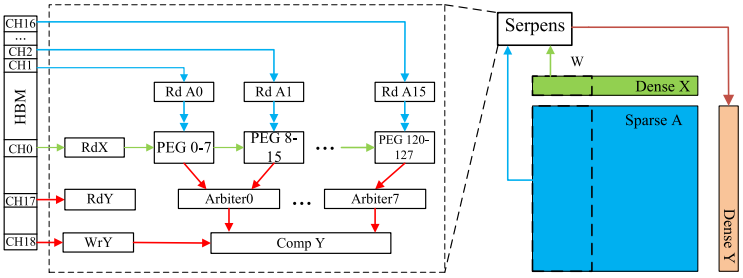

Fig. 15. Overall architecture of Serpens within [57].

Serpens [57] facilitates streaming transfer of sparse matrices utilizing HBM. To achieve sequential access to off-chip memory, Serpens customizes the read pattern for the dense vector x and the accumulation method for vector y. As illustrated in Figure 15, in Serpens, the dense vector x is divided into segments, with only one segment of x transmitted and stored in BRAM at a time. Subsequently, corresponding segments of the sparse matrix A are streamed in, using **UltraRAM (URAM)** as an accumulation buffer for accumulating results. Iteratively processing segments of x minimizes communication with off-chip memory. Compared to sparse matrices, dense vectors are smaller in size, so each dense vector is allocated an HBM channel, while 16 HBM channels are designated for sparse matrices. Each HBM channel is connected to a read (Rd) or write (Wr) module, facilitating streaming access to off-chip memory. PEs perform vector multiplication, and an arbiter selects computation results from 16 PEs to send to the Comp Y module, which accumulates these results. Serpens leverages the HBM memory channels for high-throughput processing of sparse matrices.

Du et al. [13] propose an innovative storage approach for sparse matrices, focusing on partitioning, streaming, and packaging into a new format to optimize the use of HBM bandwidth for sparse matrix access. As matrix sizes exceed on-chip memory capacity, they partition matrices row-wise and column-wise based on vector buffer and output buffer sizes. Streaming involves cyclically distributing rows to PEs while forming a data stream for rows assigned to the same PE. They insert next-row markers at each row's end to signal row transitions and skip empty rows. Element streams are then packed into data packet streams stored in HBM channels, accessed by clusters of PEs. They propose an on-chip buffer area design, shown in as Figure 16. In a cluster, all PEs share a dense vector buffer, with vector buffer access units handling requests for the vector buffer group. Input vectors are duplicated across different clusters. Two shuffle units handle vector buffer access unit requests and responses, dynamically solving bank conflicts by reordering payloads in input

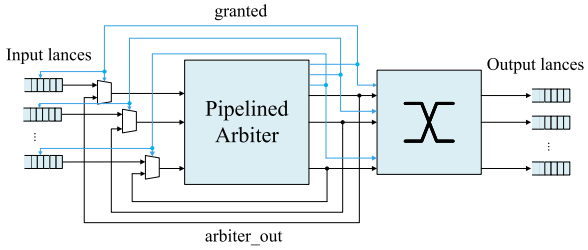

Fig. 16. Shuffle unit with explicit control logic within [13].

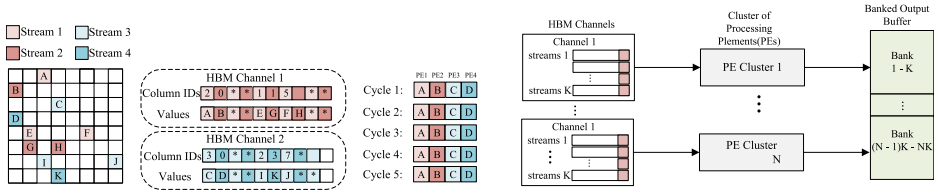

Fig. 17. The design within GraphLily [26]: (a) CPSR sparse matrix storage format with two HBM channels; (b) architecture of the SpMV accelerator. CPSR, cyclic packed streams of rows.

channels, incorporating a pipelined arbiter and resend logic for enhanced efficiency and conflict resolution.

GraphLily [26] specifically proposes a sparse matrix storage format for HBM, named **Cyclic Packed Streams of Rows (CPSR)**. A figure exemplifies four packed row streams across two HBM channels, where eight parallel rows are grouped into four sets. CPSR comprises only two arrays: one for nonzero values and the other for the corresponding column indices. End-of-row markers are inserted to signal the conclusion of each row. For instance, in HBM channel 1, as shown in Figure 17(a), elements from two streams are alternately placed until both streams are depleted. Virtual elements are added to the shorter stream for alignment, enabling parallel operation of four PEs. Figure 17(b) displays the SpMV accelerator's architecture, including a set of HBM channels, multiple PE clusters, and an output buffer. Each cluster is connected to an HBM channel. To minimize random off-chip accesses to the dense vector, each PE cluster contains a **Cluster Shared Vector Buffer (CSVecBuf)**. This buffer is cyclically partitioned into K banks, each cycle supplying a vector value to K PEs. CSVecBuf, integrating vector replication and storage, can provide data to a large number of PEs.

Jiang et al. [29] propose an optimized design for a systolic array architecture on FPGA platforms, tailored for sparse deep learning models. They suggest replacing traditional compressed format indices of nonzero values with a sparse bitmap, as shown in Figure 18(a). In this bitmap, the binary code 0 and 1 indicate the positions of zero and nonzero numbers, respectively. During index matching, only input data corresponding to the positions marked with 1 are extracted. To address the unpredictability of a given sparse matrix's sparsity, a dual-buffer structure is employed. One buffer sends data to the PE array only when it is full, while the other buffer stores overflow data, as shown in Figure 18(b). For enabling OpenCL kernels to access multiple HBM2 pseudo channels, they duplicated input reader and filter reader kernels. This allows each duplicated instance to read partitioned data from different HBM2 pseudo channels, thereby accelerating memory-bound operations in the computing, as shown in Figure 18(c).

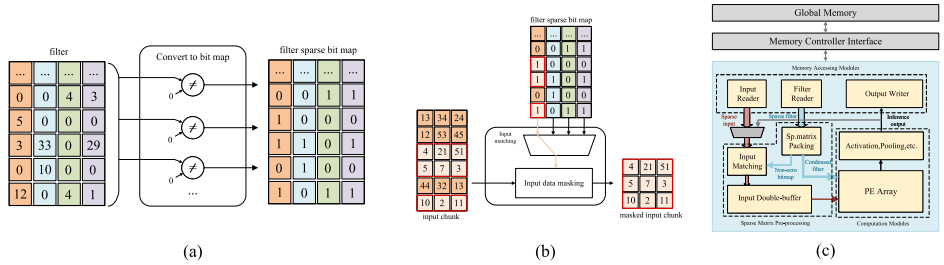

Fig. 18. The design within [29]: (a) representation of locations of zero and nonzero numbers with sparse bitmaps; (b) compression of input feature maps based on filter sparse bitmaps; (c) overall architecture of proposed optimization of sparse CNN accelerator.

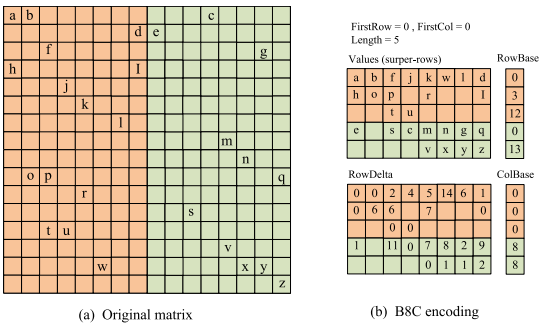

Fig. 19. The b8c encoding of sparse matrices within [47]. b8c, block-8 compress.

Oliver et al. [47] propose a novel encoding format named **Block-8 Compress (b8c)** and implement its corresponding hardware architecture using OmpsS@FPGA and HLS. For matrices exceeding memory size, a preprocessing step is involved. The matrix is partitioned into vertical blocks of size C, with C determined by the off-chip memory's data path width and the size of nonzero elements. The process then attempts to merge sparse rows, ensuring the row distance does not exceed an adjustable parameter, and that they are non-overlapping. The merged structure is illustrated in Figure 19(b). The b8c metadata also includes the matrix values in the Values structure, along with a set of metadata associated with each super-row (*RowBase*, *RowDelta*, *ColBase*) and metadata for the entire sub-block (*FirstRow*, *FirstCol*, *Length*). This representation facilitates conflict-free access to the matrix. Storing b8c super-rows continuously in memory allows for streaming data processing, reducing associated access delays.

Li et al. [37] design a row merging algorithm to enhance data locality between rows. This algorithm iterates through all rows, grouping together those with similar counts of nonzero elements and significantly overlapping column indices. For groups with rows of varying lengths, shorter rows are padded with zero elements to equalize their length. This approach not only increases data density by reorganizing nonzero rows but also improves the buffer hit rate for vector indices. The algorithm then assigns different row groups to various PEs for parallel processing. Each PE efficiently accesses matrix data through its local HBM Pseudo Channel. In every PE, a private L1 cache is integrated. When column indices become memory access addresses, requests are initially directed to the L1 cache. If there's a cache miss, the request is then sent to the shared L2 cache. This dual-cache structure effectively utilizes the locality of vectors, enhancing both data efficiency and access speed.

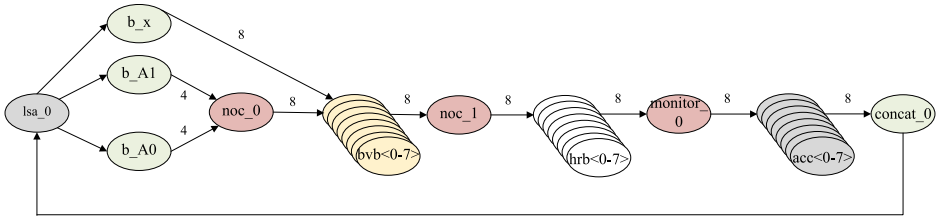

Fig. 20. Decomposing SpMV pipeline into modular blocks communicating over elastic channels within [28].

Jain et al. [28] propose an SpMV accelerator design that utilizes limited on-chip resources while fully leveraging available bandwidth. This approach incorporates the concept of modular Intellectual Property blocks. As shown in Figure 20, the SpMV accelerator comprises various building blocks, each designed as a modular component. This modularization allows for a flexible and scalable architecture, enabling the accelerator to adapt to different hardware constraints and performance requirements. The 2×2 switches of noc_0 and noc are used to construct a **Network on-Chip (NoC)**, consisting of four elastic buffers, two 2-way split units (S2), and two 2-way merge units (M2). The multistage switching network formed by these 2×2 switches in the NoC enables inputs of nonzero values with any column index to be routed to any port and indexed to one of the appropriate output ports. The **Banked Vector Buffers (bvb)**<0-7> modules are used for storing input vector groups and as input buffers for multiplication operations in the SpMV accelerator, with the size of the bvb being selectable based on matrix dimensions. The combination of NoC and bvb in the switching network allows for irregular indexing and increased throughput by keeping vector elements on-chip. The lsa module is a Load Store Adapter used for controlling data movement, enhancing the design's flexibility and managing the connection between the HBM channels and the SpMV accelerator. The hrb<0-7> units are **Hazard Resolving Backpressure (HRB)** units that can handle dependencies between data. One challenge in achieving high performance is accumulating values provided in consecutive clock cycles to a deeply pipelined adder. However, due to data dependencies, it's necessary to wait for earlier data to complete addition before accumulating subsequent data. The HRB uses shift registers to track all active indices and compares incoming row indices with values in the register. If they do not match, the HRB moves and accumulates; if they match, the HRB waits for the conflicting index to clear from the register before accepting a new pair of index values. The acc<0-7> modules are accumulators (ACC) used for storing output vector groups and performing accumulation operations. The b_A0, b_A1, and b_x modules are stream splitters that can divide a wide stream into multiple narrower streams. The monitor module is used to count nonzero data packets flowing through the pipeline and can immediately output results after processing the last nonzero packet. This modular and lean architecture allows the accelerator design to fully utilize available bandwidth and provides potential for scalability.

4.4 Parallel Computing Efficiency

4.4.1 Load Balancing. Parallel computing serves as a promising solution for achieving HPC. However, without achieving load balancing among parallel computing units, it will lead to high idleness and computing resource wastage in these units [42]. In SpMM, each computing unit is tasked with processing a varying number of nonzero elements. Several optimization strategies are dedicated to achieving load balancing among computing units [24]. For instance, task division algorithms are employed to evenly distribute matrix multiplication tasks among different computing units, ensuring a relatively balanced workload [10]. Additionally, dynamic load balancing techniques

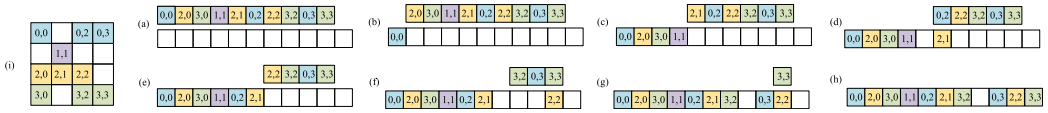

Fig. 21. The nonzero dispatch process within Sextans [58].

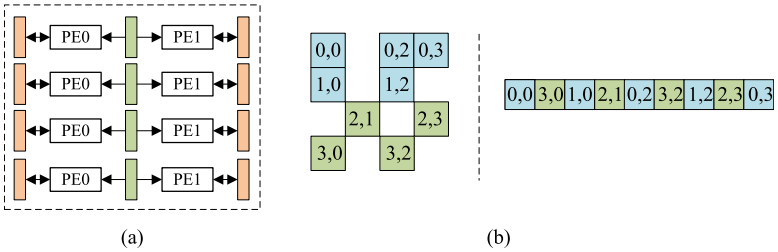

Fig. 22. The design within Serpens [57]: (a) eight PE processing units; (b) example of data rearrangement, assuming a DSP delay of 2.

can be considered, which adjust the distribution of computational tasks in real-time, based on current computational demands and resource utilization.

To enhance load balancing in SpMM, Sextans [58] applies a novel approach to matrix A by dividing and recompressing it, aiming for an approximately uniform distribution of nonzero points. This methodology is crucial in circumventing **Read-after-Write (RAW)** dependencies, especially when assigning values from various rows to the same processing unit. Sextans introduces a unique PE-aware nonzero scheduling algorithm centered on a specific cycle interval, D. This interval is used to schedule nonzero elements in the earliest possible cycle, ensuring their row indices do not have any RAW dependencies and have been in processing for the past D cycles. As depicted in Figure 21, with a RAW dependency distance of 4, nonzero elements with identical row indices are indicated in the same color. During the initial four cycles, nonzero elements with distinct row indices are sequentially scheduled. Conflicts, such as between the yellow (2,1) and yellow (2,0), are resolved by scheduling them in the earliest subsequent cycle, such as the fifth one, and the resultant gaps or bubbles are efficiently filled with nonconflicting entries like blue (0,2). This approach of PE-aware nonzero value scheduling significantly contributes to maintaining an II=1 pipeline, thereby facilitating the parallelism of SpMM computations.

Serpens [57] addresses the challenge of module access across multiple HBM channels by allocating sparse elements from one channel to eight PEs. Each PE performs a segment of the SpMV $A \times x$. As shown in Figure 22, since each BRAM features two ports and accommodates four duplicated segments of the dense vector x, it enables two PEs to share one BRAM, thereby resolving potential conflicts in storage access. Additionally, each PE operates with distinct URAM addresses, preventing bank conflicts in URAM that could arise from concurrent access by multiple PEs. Serpens further optimizes the process by merging two values with adjacent target row indices into a single URAM address. This strategy not only streamlines the reordering of nonzero elements but also effectively manages RAW conflicts, thus enhancing the computational efficiency of the architecture.

Li et al. [33, 36] propose an FPGA-based architecture to accelerate SpMM using the Gustavson method. This design effectively tackles the load imbalance in Gustavson's parallel approach by implementing parallel processing of matrix A's elements within each row, rather than parallelism across rows. Each PE sequentially handles elements from matrix A and employs a pipelined approach to merge intermediate results. Specifically, these results are temporarily stored in each

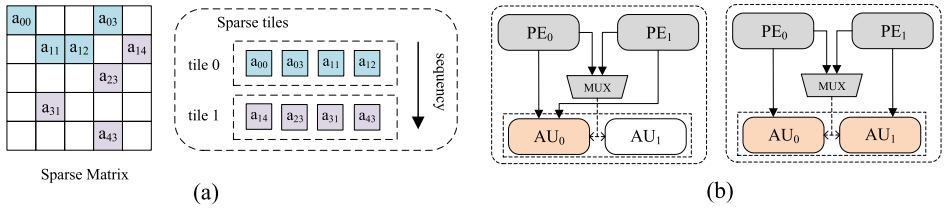

Fig. 23. The propose partitioning approach [16]: (a) divided into multiple sparse tiles according to equal values; (b) merging of partial result.

merger's local buffer. When a new row begins processing, these buffered results are forwarded to a final merger unit. This final merger combines four partial results using two auxiliary mergers and subsequently writes the combined output to DDR memory. During the merge, it compares the first elements' column indices in both input streams to check if they represent the same position. If they match, a merge occurs; if not, the element with the lesser column index is moved to the output stream. This approach of parallel processing at the element level minimizes the idle time of PEs, thereby enhancing the efficiency of the entire matrix multiplication operation.

Gao et al. [16] propose an accelerated SpMM architecture for GNNs, incorporating a partitioning approach that divides sparse matrices into equal-value partitions for balanced dataflow. As shown in Figure 23(a), this method splits CSR-formatted sparse matrices into multiple tiles based on their storage sequence, with each tile containing an identical number of nonzero elements. These tiles are sequentially transferred to on-chip buffers for processing. Unlike two-dimensional (2D) block partitioning, equal-value segmentation is more suited for highly sparse and imbalanced graph data. During the multiplication phase, nonzero elements are processed in parallel, as each sparse matrix element's multiplication with a dense matrix row is independent. The merging phase is responsible for accumulating the partial results from multiplication. As depicted in Figure 23(b), nonzero elements within a tile are multiplied with corresponding rows in PEs. The partial results generated are routed to merging units based on their row indices, ensuring workload balance throughout the computation.

Li et al. [37] propose a row assignment algorithm designed to distribute nonzero elements evenly across PEs for computation. The algorithm features a scoring system based on the current status of the PEs. It establishes an ideal workload for each PE as the scoring benchmark. When allocating a group of nonzero rows to a PE makes it more aligned with this benchmark, the allocation scores higher. The algorithm also awards additional points if there is a locality association between the row group and the PE. This method balances load distribution while considering data locality.

Chen et al. [5] propose an efficient parallel computing method for symmetric SpMV. The previous approach, illustrated in Figure 24(b), struggles to avoid row-wise imbalance, leading to resource and power wastage. The ideal solution for balancing rows, shown in Figure 24(c), requires thorough data analysis and packaging, consuming significant computational resources. To address the imbalance in row-wise computations, eSSpMV employs a pipelined approach. As depicted in Figure 24(d), data from n rows are allocated to n computing units. Once the computation is completed, the n+1 row data are immediately fed in for processing. During this operation, since the matrix is an upper triangular matrix, the number of nonzero elements per row gradually decreases, helping to balance the workload across different computing units to some extent.

4.4.2 Unified PE Architecture. For efficient matrix multiplication on FPGAs, designing an unified PE architecture offers significant benefits. This approach leverages FPGA flexibility to balance computational workloads across a computing module. Such a design streamlines resource

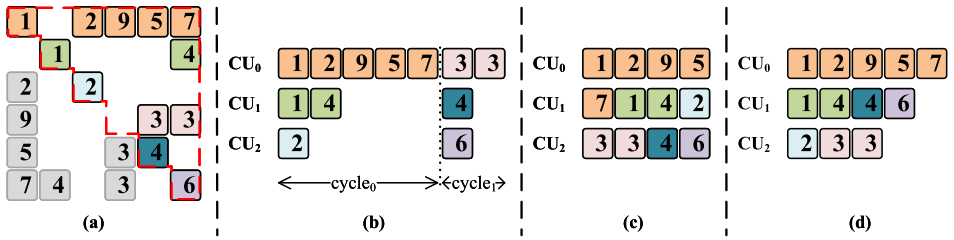

Fig. 24. Different parallelization strategy within [5]: (a) a symmetric sparse matrix; (b) row-based paralleliza-tion; (c) ideal parallelization; (d) pipeline-based parallelization.

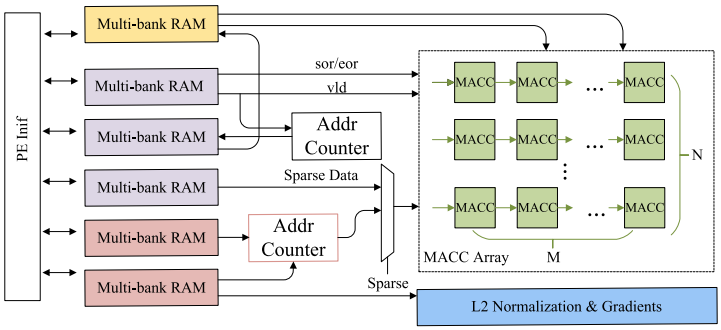

Fig. 25. Architecture of MACC within SkeletonGCN [64].

management, fostering adaptability and compatibility. Consequently, this strategy provides a flexible solution for diverse matrix multiplication applications in HPC environments.

Wu et al. [64] introduce a GCN accelerator named SkeletonGCN, featuring a unified PE architecture suitable for SpMM, GeMM, and **Matrix Multiplication with Transposed (TMM)**. This architecture primarily consists of a MACC array, where MACC units in each row share the same input. For SpMM, as shown in Figure 25, they employ three independent RAMs to store the CPCOO format, including Header, *col*, and *val*. The Header RAM's *vld* signal activates the address counter for generating addresses in the Body RAM, while *sor* and *eor* signals manage the start of new row computations and the saving of results. The corresponding dense data reside in Multibank RAM, indexed by the column values of nonzero elements in the Column Pos RAM. Owing to a fully pipelined architecture, the MACC units remain active for most SpMM computation cycles, ensuring high DSP efficiency. For GeMM and TMM operations, they add an extra data distribution module to manage the dataflow in these computations.

Zhang and Prasanna [69] propose Dynasparse, a software-hardware co-design accelerator capable of supporting various sparsity levels in data and GNN models. Dynasparse executes three computational modes on FPGA: GEMM, **Sparse-Dense Matrix Multiplication (SpDMM)**, and SpMM, along with sparsity analysis and format conversion tasks. Figure 26 illustrates the architecture under these three different computation modes. Each buffer in the system consists of P_{sys} banks for parallel on-chip storage access. In the GEMM mode, the **Arithmetic Logic Unit (ALU)** array is organized into a 2D systolic array, executing GEMM using an output stationary dataflow. This systolic array can perform P_{sys}^2 MAC operations per clock cycle. For SpDMM mode, the ALU array is divided into $P_{sys}/2$ Update Units and $P_{sys}/2$ Reduce Units. Nonzero elements of the sparse matrix X from Buffer U are read and sent to the Index Shuffle Network, then routed to Buffer O to read the dense matrix Y, forming input data pairs. The Update Units perform element-wise multiplication

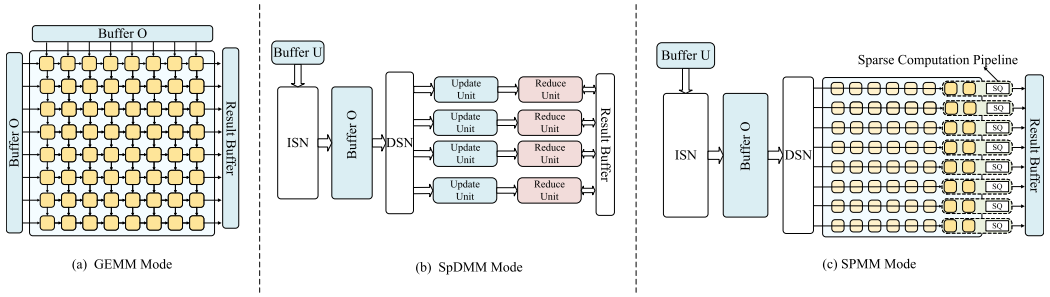

Fig. 26. Different execution modes of the computational core within [69]: (a) GEMM; (b) SpDMM; (c) SPMM.

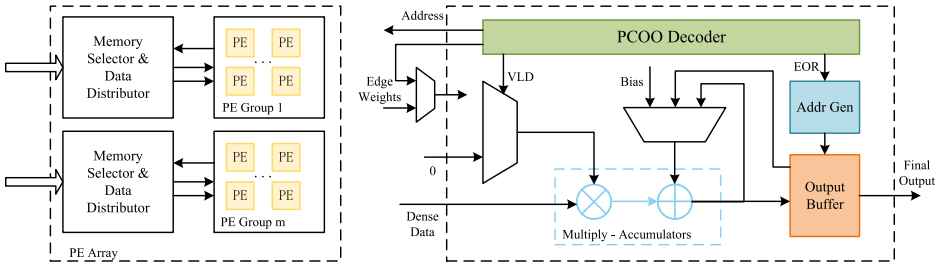

Fig. 27. Architecture of PE arrays within LW-GCN [60].

to generate intermediate results, which are then accumulated by the corresponding Reduce Units. The SpDMM mode can execute $P_{sys}^2/2$ MAC operations per clock cycle. In the SpMM mode, the ALU array is organized into parallel **Sparse Computing Pipelines (SCP)**. Each SCP contains two ALUs for nonzero element multiplication and intermediate result merging, with a Sparse Queue responsible for storing intermediate results. Following a row-product computation rule, one SCP is allocated to compute one row of the output matrix. The SpMM mode can execute P_{sys} MAC operations per clock cycle. The coupling and switching between these three modes effectively leverage the data sparsity in GNN inference, reducing inference latency.

Tao et al. [60] propose that GeMM is essentially SpMM with a density of 100%. Based on this concept, they designed a unified PE architecture to efficiently handle both GeMM and SpMM. As shown in Figure 27, a memory selector and data distributor appropriately schedule the data. Sparse data, compressed in the PCOO format, are streamed directly to the PE. The PCOO decoder and bit indicates valid route the data to the corresponding multipliers, with *SOR* and *EOR* signals indicating the start and end of a row, respectively, to differentiate multirow scenarios. The *EOR* signal controls address generation for storing current results into the output buffer. During MM processing, the $sparse\_flag$ is set to 0.

Graph-OPU [4] specifically design a PE unit and a selection adder tree compatible with the OPCOO format, to increase the efficiency of both SpMM and GEMM. Figure 28(a) showed the architecture of computational engine. Graph-OPU duplicates sparse elements and tiles dense vectors to achieve efficient execution of SpMM and can be extended to GEMM. As shown in Figure 28(b), PE units are connected to a selective adder tree that determines whether the products from different PE units need to be summed up by means of the *eor* signal, which ensures that the results of each row element are correctly outputted. The architecture of computational engine achieves full pipeline and allows switching to two different modes, SpMV and GEMM.

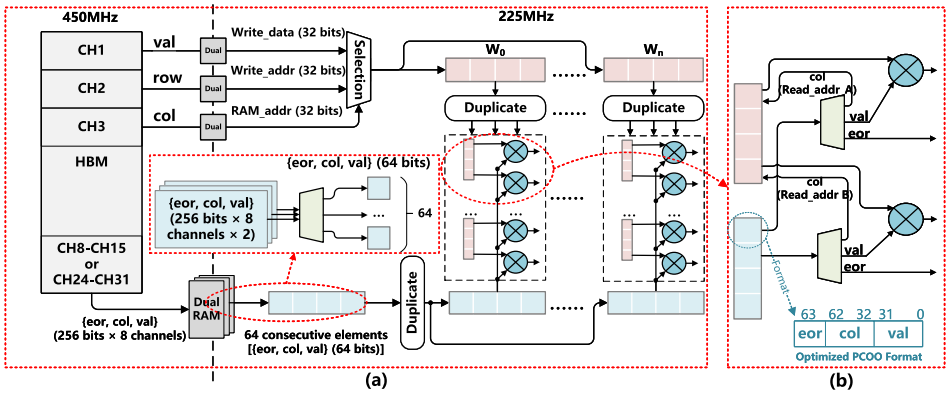

Fig. 28. Fully pipelined general purpose computing engine in Graph-OPU [4]: (a) the architecture of computational engine; (b) the architecture of PE units is connected to a selective adder tree.

5 Comparison of Recent FPGA-Based SpMM Accelerators

To provide readers with a more direct understanding through data about these FPGA-based SpMM accelerator designs, we compare them in this section. The comparison is mainly divided into two parts: the hardware resource consumption and the performance comparison.

5.1 Hardware Resource Consumption

Resource consumption is a crucial metric for evaluating the design of FPGA-based accelerators. We evaluate and statistically compare the resource consumption of the mentioned sparse matrix multipliers based on different design architectures, as depicted in Table 5. Our primary focus includes **Lookup Table (LUT)**, **Flip-Flop (FF)**, DSPs, BRAM, and URAM. LUTs, which can implement any logic function, indicate the usage of logical resources. The consumption of FFs reflects the design's timing requirements and storage demands. DSPs are crucial for executing key computational tasks like multiplication and addition. The use of BRAM and URAM indicates the on-chip storage capacity needed by the architecture. The data are derived from the corresponding descriptions in their respective articles. It should be noted that for accelerator design architectures that only provide the percentage of consumed resources, we calculate the actual resource consumption based on the logic resources configured in their implementation devices. Gao et al. [16] deploy two versions of design prototypes, corresponding to fixed-point and floating-point data types. In this context, we only provide the resource consumption data for the floating-point data accelerator. The *NA* in the table indicates that the relevant data are not mentioned in the original articles.

Overall, most SpMM accelerators operate around a frequency of 200 MHz. The highest operating frequency is achieved by [28] at 310 MHz, benefiting from its elastic communication between modules, facilitating the design's timing closure. The studies by [33, 36, 40] have a frequency of only 100 MHz, possibly due to their choice of edge FPGA, the ZCU106. Serpens, aiming to accommodate the 450 MHz frequency of HBM, sets its computational frequency at 223 MHz, maximizing the utilization of HBM's bandwidth.

In terms of resource consumption, Dynasparse [69] is the design with the highest resource usage among all accelerators. This is because Dynasparse implements a unified accelerator on FPGA capable of executing various computational primitives and develops efficient hardware mechanisms for analyzing data sparsity and performing real-time data format conversion. This endows Dynasparse with greater versatility and scalability.

Table 5. Frequency and Resource Consumption of Multipliers

Multiplier	Vintages	Device	Calculation	Frequency (MHz)	LUT	FF	DSP	BRAM	URAM	Max cols/rows
Li et al. [33, 36]	2023	Xilinx Zynq-UltraScale ZCU106	SpMM (Gustavson)	100	152K	203K	46	201	60	23,560
Liu and Liu [40]	2023	Xilinx Zynq UltraScale ZCU106	SpMV	100	171K	86K	43	93	70	281,903
Jain et al. [28]	2023	Xilinx Alveo U280	SpMV	310	500K	1,178K	644	460	384	14,734
Li et al. [35]	2023	Xilinx Zynq UltraScale ZCU106	SpMV	100	175K	178K	258	153	NA	217,918
Dynasparse [69]	2023	Xilinx Alveo U250	GEMM/SpDMM/SpMM	250	1,728K	NA	12,288	2,688	960	232,965
eSSpMV [5]	2023	Xilinx Zynq XCZU7EV	SSpMV	250	3,494	5,621	1,024	61.5	NA	41,731
Oliver et al. [47]	2023	Xilinx Alveo U280	SpMV	250	398K	490K	1,353	669	71	52,329
ReMCOO [43]	2023	Xilinx Zynq UltraScale+ ZCU104	SpMV	NA	17K	1,268	5	10	NA	331
Graph-OPU [4]	2023	Xilinx Alveo U50	SpMM/GEMM	225	475K	427K	2,742	927K	NA	45,954
Sextans [58]	2022	Xilinx Alveo U280	SpMM (Outer product)	189	379K	690K	3,316	3,086	768	5,133,551
Serpens [57]	2022	Xilinx Alveo U280	SpMV	223	173K	327K	720	655	384	108K
GraphLily [26]	2022	Xilinx Alveo U280	SpMV/SpMSpV	165	390K	493K	723	417	512	2,997K
Du et al. [13]	2022	Xilinx Alveo U280	SpMV	237	544K	NA	688	128	512	2,449K
Li et al. [37]	2022	Xilinx Alveo U50	SpMV	237	NA	NA	NA	NA	NA	345,241
LW-GCN [60]	2022	Xilinx Kintex-7 K325T	SpMM/GEMM	200	161K	94K	512	291.5	NA	19,717
SkeletonGCN [64]	2022	Xilinx Alveo U200	SpMM/GEMM/TMM	250	1,021K	NA	4,460	1,338	598	2,000
Gao et al. [16]	2021	Xilinx Alveo U280	SpMM (Gustavson)	200	623K	793K	2,251	1,163	896	65,755
FSpGEMM [61]	2021	Intel Arria 10 GX 1150	SpGEMM (Gustavson)	236	256K[a]	717K[b]	547	NA	NA	1,000K
Jiang et al. [29]	2021	BittWare 520N-MX	SpMM	257	NA	NA	1,142	NA	NA	NA
Hosseinabady and Nunez-Yanez [23]	2020	Xilinx ZCU102	SpMV/SSpMV	200	38K	70K	56	876	NA	39,900
Pligouroudis et al. [51]	2020	Intel Stratix IV GX	SpMV	289.77	NA	NA	NA	NA	NA	500

[a]Intel FPGAs in Adaptive Logic Modules (ALM).
[b]Intel FPGAs in ALM Registers.

5.2 Performance Comparison

In assessing FPGA-based SpMM accelerators, throughput and power consumption are key metrics for assessing system performance. Throughput represents the workload completed by the accelerator per unit of time, directly reflecting its efficiency and performance. The throughput is calculated as

$$Throughput = P/t(GOPS), \tag{7}$$

where P is the problem size and t is the execution time [58]. Power consumption, on the other hand, indicates the relationship between the accelerator's resource usage and energy expenditure, highlighting differences in architectural design. An effective FPGA-based accelerator design must strike a balance between performance and power consumption. We attempt to assess and compare the performance of various FPGA-based SpMM designs previously mentioned. Throughput data are obtained from the respective articles. For some designs that are open source but not directly discussed in the articles, we conducted tests using their reported development boards and recorded the relevant data. For power consumption, we standardized the measurement using the Xilinx Power Estimator [66] and Xilinx Board Utility [65] to obtain their power data. This approach is due to the variance in power consumption reporting methods across different articles. Some report the power consumption of the FPGA chip, while others report the entire board-level power consumption.

Following the statistical methodology of [19], Figure 29 presents a performance comparison of different accelerator designs. The horizontal axis represents power consumption (W), and the vertical axis indicates throughput (GOPS). For clarity, the values in the graph have been scaled using log10. Due to its design incorporating rapid random access and PE-aware nonzero scheduling, Sextans [58] achieves the best performance with a peak throughput of 5,796.2 GOPS. The preprocessing of nonzero elements in Sextans, which masks the inherent data dependencies and combines with the high data bandwidth of HBM, further enhances its performance. The throughput of LW-GCN is estimated based on the inference latency for the Cora dataset. The focus of LW-GCN on optimizations specific to graph data might contribute to its relatively lower throughput. Oliver et al. [47] shows the highest power consumption, attributed to its unique design optimized for 64-bit processing, which considerably increases its resource usage. eSSpMV [5] and Hosseinabady and Nunez-Yanez [23] demonstrate balanced performances, owing to their specific optimizations for

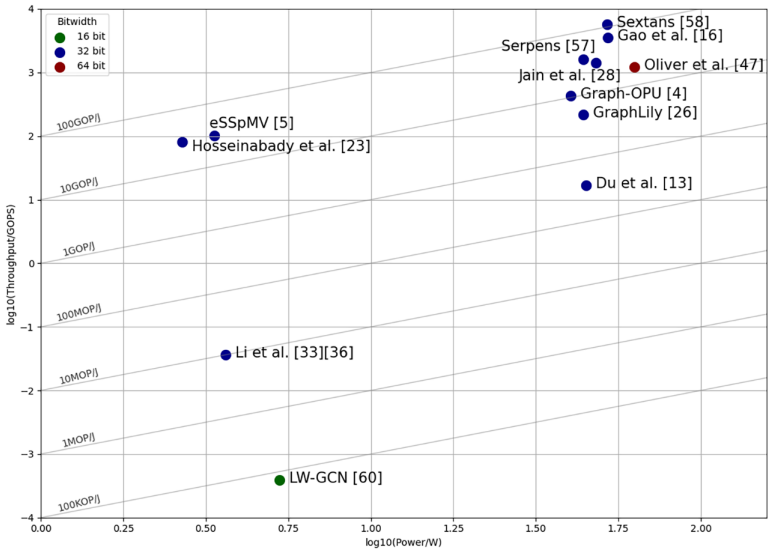

Fig. 29. A comparison between different designs on a logarithm COO of power and performance.

symmetric sparse matrices, which effectively reduce their computational load to a portion of the matrix. Their use of edge FPGAs also explains their lower power consumption.

6 Prospects

We believe that the demand for SpMM computation will emerge in more fields. Several of the above-mentioned articles also talk future research and specific applications. They hope to leverage the reconfigurable advantages of FPGAs to design more efficient computing architectures that can adapt to the varying characteristics of sparse matrices and maximize hardware performance. For instance, Gao et al. [16] anticipate exploring efficient architecture designs tailored to GNN models based on their proposed framework. In FlightLLM [67], to achieve FPGA-based LLM inference acceleration, a configurable sparse DSP chain is proposed for the SpMM process to reduce hardware overhead while supporting sparse reduction. They believe that FPGAs are promising candidates for efficient LLM inference. Jiang et al. [29] presented a preprocessing method for sparse matrices, aiming for its application not only in sparse filters but also in feature maps of sparse CNN models to further optimize sparse convolutions. Moreover, diversity and scalability are also crucial directions for future research. For example, ReMCOO [43] introduced a new method to represent sparse matrices and vectors and implemented it in FPGAs. They expect that future research could use the newly proposed sparse matrix vector representation to study the performance of very large SpMVs on GPU platforms. Furthermore, Zhang and Prasanna [69] proposed Dynasparse, a hardware-software co-design for dynamic sparsity utilization in GNN inference. In the future, it is hoped that this can be extended to heterogeneous platforms composed of CPUs, GPUs, and FPGAs. Here, we propose several specific or broad potential directions for future FPGA-based SpMM designs.

Optimization of Compressed Formats. Various storage compression formats, such as CSR, CSC, and COO, have been proposed to reduce memory overhead. However, these formats do not always provide sufficient support for HPC in SpMM in specific applications. Designing new matrix compression formats to overcome the limitations of existing ones is a research direction with practical application value. Additionally, developing assessment models to analyze the advantages

and disadvantages of each compression format in different applications can provide crucial guidance for kernel developers in these applications, representing a potential contribution to the field.

Reducing Data Preprocessing Overhead. To mitigate the complexity of sparse structures and more efficiently utilize FPGA computational resources, optimization algorithms such as matrix compression, matrix blocking, and load balancing strategies have been developed. However, preprocessing often incurs additional costs, as it requires extra computation and resources. Typically, the cost of converting to a matrix compression format rivals that of accelerating SpMM computations [8]. If data preprocessing cannot balance the benefits with the additional resource overheads, it cannot truly aid in accelerating sparse matrix operations, even if it makes computations more hardware-friendly. Therefore, exploring ways to reduce data preprocessing overhead, or even avoiding it through hardware-aware design, is a topic worth investigating in the design of SpMM accelerators.

Efficient Memory Access Strategies. As memory-intensive applications, SpMM is constrained by memory bandwidth, leading to widespread adoption of HBM in SpMM accelerators. However, achieving efficient access to HBM on FPGAs is challenging. FPGA on-chip logic can globally address HBM via standard AXI interfaces, but concurrent multichannel access to the sparse data stored in HBM can lead to substantial access conflicts. This increases the latency of AXI read/write transactions and reduces accelerator performance. Recent works have reduced redundant computations in networks through techniques like network pruning. However, this often results in both matrices involved in the multiplication being considered sparse, exacerbating the irregularity of memory access. In accelerating SpMM on FPGAs, a critical area of research is how to fully exploit the temporal and spatial locality of memory access to enhance memory access efficiency.

Balancing the Workload across Computing Units. This challenge spans multiple levels, encompassing coarse-grained task distribution among computational units, fine-grained data allocation within individual units, and data-level SIMD operations. In SpMM, the approach to scheduling at various levels significantly influences the equilibrium of computational workloads. In the context of coarse-grained parallelism, where multiple hardware logic units process data concurrently, the challenge lies in effectively breaking down tasks into manageable segments and judiciously assigning resources. In fine-grained parallelism, which operates at more elementary units, it becomes essential to devise strategies for organizing and managing a multitude of smaller tasks to execute parallel computations efficiently. For FPGA-accelerated SpMM, the creation of sophisticated scheduling algorithms is paramount. These algorithms should dynamically assign tasks to computational units of different granularities, based on an in-depth understanding of task characteristics and resource availability. Such adaptive algorithms aim to optimize task allocation dynamically, thus achieving a balanced workload distribution in an HPC environment.

Optimizing for Specific Dataflows. The choice of dataflow greatly impacts memory access and computational burdens [22, 58]. The Inner Product method facilitates tiled computation and exhibits high spatial locality in memory, yet suffers from low data reuse rates in large sparse matrices. The Outer Product paradigm, aligning with the conditions for DSP packing (an optimization technique in FPGA design), can achieve higher DSP efficiency. Gustavson's method significantly reduces operands in SpMM and directly produces an entire row of the output matrix, facilitating efficient coupling with downstream tasks. However, Gustavson's approach makes memory access more challenging, as it renders both temporal and spatial locality of memory access unpredictable. When computing SpMM in specific application scenarios, selecting the appropriate dataflow and customizing corresponding memory access patterns and architectures are vital for performance enhancement. We anticipate that future explorations will leverage application-specific characteristics more effectively, improving the efficiency of SpMM in targeted applications.

Optimizing for Specific Applications. There are an increasing number of applications whose computational processes involve SpMM, making specific optimizations crucial. For example, there

are special forms represented by symmetric sparse matrices in recommendation systems. How to perform specific optimizations for these while ensuring minimal resource overhead and compatibility with general SpMM computation will be a key problem to address. Additionally, in NLP applications such as Transformer, there are numerous matrix computations, including sparse-dense, sparse-sparse, and dense-dense multiplications. Designing efficient and low-resource-consumption PEs to handle these computations will also be a significant concern.

7 Conclusion

SpMM has always been a fundamental computational process in HPC. The rise of large-scale models like ChatGPT further underscores the importance of SpMM. General-purpose CPUs and GPUs struggle with fine-grained SpMM optimization, which presents a significant opportunity for FPGA platforms but also imposes greater demands and challenges. Designers face multifaceted challenges, ranging from matrix storage and memory access to parallel computations. In this article, we first categorize the challenges faced by FPGA-based SpMM accelerator designs into four types; then we summarize the existing work on how FPGAs effectively address these four types of challenges; finally, we propose further optimization directions and application prospects for FPGA-based SpMM designs. We hope that this article will attract more researchers to focus on this field. This interest is expected to enable FPGAs to continue providing efficient solutions for HPC, especially in the realm of SpMM.

References

[1] Leon Adams and Strategic Marketing. 2002. *Choosing the Right Architecture for Real-Time Signal Processing Designs*. Texas Instruments, Dallas, TX, USA.

[2] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, Hyderabad, India, 804–811. DOI: https://doi.org/10.1109/IPDPSW.2015.75

[3] Aydin Buluç and John R. Gilbert. 2011. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509. DOI: https://doi.org/10.1177/1094342011403516

[4] Ruiqi Chen, Haoyang Zhang, Shun Li, Enhao Tang, Jun Yu, and Kun Wang. 2023a. Graph-OPU: A highly integrated FPGA-based overlay processor for graph neural networks. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Gothenburg, Sweden, 228–234. DOI: https://doi.org/10.1109/FPL60245.2023.00039

[5] Ruiqi Chen, Haoyang Zhang, Yuhanxiao Ma, Jianli Chen, Jun Yu, and Kun Wang. 2023b. eSSpMV: An embedded-FPGA-based hardware accelerator for symmetric sparse matrix-vector multiplication. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Monterey, CA, USA, 1–5. DOI: https://doi.org/10.1109/ISCAS46773.2023.10181734

[6] Yuedan Chen, Guoqing Xiao, Fan Wu, Zhuo Tang, and Keqin Li. 2020b. tpSpMV: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures. *Information Sciences* 523 (2020), 279–295. DOI: https://doi.org/10.1016/J.INS.2020.03.020

[7] Yuedan Chen, Guoqing Xiao, and Wangdong Yang. 2020a. Optimizing partitioned CSR-based SpGEMM on the Sunway TaihuLight. *Neural Computing and Applications* 32, 10 (2020), 5571–5582. DOI: https://doi.org/10.1007/s00521-019-04121-z

[8] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic generation of efficient sparse tensor format conversion routines. In 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, Copenhagen, Denmark, 823–838. DOI: https://doi.org/10.1145/3385412.3385963

[9] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *55th Annual Design Automation Conference (DAC)*. ACM, New York, NY, 1–6. DOI: https://doi.org/10.1145/3195970.3195999

[10] Paolo D'Alberto, Abhishek Jain, Ismail Bustany, Henri Fraisse, and Mansimran Benipal. 2023. Entropy maximization in sparse matrix by vector multiplication (*ESpMV*). arXiv:2308.00106, 1–26. Retrieved from https://doi.org/10.48550/arXiv.2308.00106

[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1, Article 1 (Dec 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663

[12] Mehmet Deveci, Simon D. Hammond, Michael M. Wolf, and Sivasankaran Rajamanickam. 2018. Sparse matrix-matrix multiplication on multilevel memory architectures: Algorithms and experiments. arXiv:1804.00695, 1–24. DOI: https://doi.org/10.48550/arXiv.1804.00695

[13] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-performance sparse linear algebra on HBM-equipped FPGAs using HLS: A case study on SpMV. In *2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, New York, NY, 54–64. DOI: https://doi.org/10.1145/3490422.3502368

[14] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)* 28, 2 (2002), 239–267. DOI: https://doi.org/10.1145/567806.567810

[15] James J. Elliott and Christopher M. Siefert. 2018. Low thread-count Gustavson: A multithreaded algorithm for sparse matrix-matrix multiplication using perfect hashing. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. IEEE, Dallas, TX, 57–64. DOI: https://doi.org/10.1109/ScalA.2018.00011

[16] Yingxue Gao, Lei Gong, Chao Wang, Teng Wang, Xi Li, and Xuehai Zhou. 2023. Algorithm/hardware co-optimization for sparsity-aware SpMM acceleration of GNNs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4763–4776. DOI: https://doi.org/10.1109/TCAD.2023.3281714

[17] Theodoros Gkountouvas, Vasileios Karakasis, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2013. Improving the performance of the symmetric sparse matrix-vector multiplication in multicore. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, Cambridge, MA, 273–283. DOI: https://doi.org/10.1109/IPDPS.2013.43

[18] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking. In *32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, Virtual Event, USA, 293–303. DOI: https://doi.org/10.1145/3350755.3400216

[19] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2019. [DL] A survey of FPGA-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12, 1 (2019), 1–26. DOI: https://doi.org/10.1145/3289185

[20] Fred G. Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269. DOI: https://doi.org/10.1145/355791.355796

[21] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *34th ACM International Conference on Supercomputing (ICS)*. ACM, Barcelona, Spain, 1–12. DOI: https://doi.org/10.1145/3392717.3392751

[22] 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)*. IEEE, Seoul, Korea (South), 84–96. DOI: https://doi.org/10.1109/HPCA51647.2021.00017

[23] 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. DOI: https://doi.org/10.1109/TCAD.2019.2912923

[24] Mohammad Hosseinabady, Mohd Amiruddin Bin Zainol, and Jose Nunez-Yanez. 2019. Heterogeneous FPGA+ GPU embedded systems: Challenges and opportunities. arXiv:1901.06331, 1–10. Retrieved from https://doi.org/10.48550/arXiv.1901.06331

[25] Te C. Hu. 1961. Parallel sequencing and assembly line problems. *Operations Research* 9, 6 (1961), 841–848. DOI: https://doi.org/10.1287/opre.9.6.841

[26] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, Munich, Germany, 1–9. DOI: https://doi.org/10.1109/ICCAD51958.2021.9643582

[27] Satoshi Itoh, Pablo Ordejón, and Richard M. Martin. 1995. Order-N tight-binding molecular dynamics on parallel computers. *Computer Physics Communications* 88, 2–3 (1995), 173–185. DOI: https://doi.org/10.1016/0010-4655(95)00031-A

[28] Abhishek Kumar Jain, Chirag Ravishankar, Hossein Omidian, Sharan Kumar, Maithilee Kulkarni, Aashish Tripathi, and Dinesh Gaitonde. 2023. Modular and lean architecture with elasticity for sparse matrix vector multiplication on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Marina Del Rey, CA, USA, 133–143. DOI: https://doi.org/10.1109/FCCM57271.2023.00023

[29] Chao Jiang, David Ojika, Bhavesh Patel, and Herman Lam. 2021. Optimized FPGA-based deep learning accelerator for sparse CNN using high bandwidth memory. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Orlando, FL, USA, 157–164. DOI: https://doi.org/10.1109/FCCM51124.2021.00026

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley,

Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, 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 *44th Annual International Symposium on Computer Architecture (ISCA)*. ACM, Toronto, Canada, 1–12. DOI: https://doi.org/10.1145/3079856.3080246

[31] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. 2019. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. DOI: https://doi.org/10.1109/HPEC.2019.8916336

[32] Srinidhi Kestur, John D. Davis, and Eric S. Chung. 2012. Towards a universal FPGA matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Toronto, ON, Canada, 9–16. DOI: https://doi.org/10.1109/FCCM.2012.12

[33] Shiqing Li, Shuo Huai, and Weichen Liu. 2023a. 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. DOI: https://doi.org/10.1109/TCAD.2023.3281719

[34] Shiqing Li, Di Liu, and Weichen Liu. 2021. Optimized data reuse via reordering for sparse matrix-vector multiplication on FPGAs. In *2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, Munich, Germany, 1–9. DOI: https://doi.org/10.1109/ICCAD51958.2021.9643453

[35] Shiqing Li, Di Liu, and Weichen LiuDi Liu. 2023b. Efficient FPGA-based sparse matrix–vector multiplication with data reuse-aware compression. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4606–4617. DOI: https://doi.org/10.1109/TCAD.2023.3281715

[36] 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)*. IEEE, Antwerp, Belgium, 1–6. DOI: https://doi.org/10.23919/DATE56975.2023.10136958

[37] Tao Li, Li Shen, and Shangshang Yao. 2022. A high-performance SpMV accelerator on HBM-equipped FPGAs. In *2022 IEEE 24th International Conference on High Performance Computing & Communications; 8th International Conference on Data Science & Systems; 20th International Conference on Smart City; 8th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, Hainan, China, 1081–1087. DOI: https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00171

[38] Hui-Hsin Liao, Chao-Lin Lee, Jenq-Kuen Lee, Wei-Chih Lai, Ming-Yu Hung, and Chung-Wen Huang. 2021. Support convolution of CNN with compression sparse matrix multiplication flow in TVM. In *50th International Conference on Parallel Processing Workshop*. ACM, New York, NY, 1–7. DOI: https://doi.org/10.1145/3458744.3473352

[39] Valerii Likhosherstov, Krzysztof Choromanski, and Adrian Weller. 2023. On the expressive flexibility of self-attention matrices. In *AAAI Conference on Artificial Intelligence*. PKP, Washington DC, USA, 8773–8781. DOI: https://doi.org/10.1609/aaai.v37i7.26055

[40] Bowen Liu and Dajiang Liu. 2023. Towards high-bandwidth-utilization SpMV on FPGAs via partial vector duplication. In *28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. ACM, New York, NY, 33–38. DOI: https://doi.org/10.1145/3566097.3567839

[41] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2019. Register-aware optimizations for parallel sparse matrix–matrix multiplication. *International Journal of Parallel Programming* 47, 3 (2019), 403–417. DOI: https://doi.org/10.1007/s10766-018-0604-8

[42] Xunyun Liu and Rajkumar Buyya. 2020. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–41. DOI: https://doi.org/10.1145/3355399

[43] Uditnarayan Mandal and Arighna Deb. 2023. ReMCOO: An efficient representation of sparse matrix-vector multiplication. In *2023 IEEE Guwahati Subsection Conference (GCON)*. IEEE, Guwahati, India, 01–06. DOI: https://doi.org/10.1109/GCON58516.2023.10183488

[44] Wendong Mao, Meiqi Wang, Xiaoru Xie, Xiao Wu, and Zhongfeng Wang. 2024. Hardware accelerator design for sparse DNN inference and training: A tutorial. *IEEE Transactions on Circuits and Systems II: Express Briefs* 71, 3 (2024), 1708–1714. DOI: https://doi.org/10.1109/TCSII.2023.3344681

[45] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In *2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation (SCA)*. Eurographics Association, Madrid, Spain, 65–73. DOI: https://doi.org/10.2312/SCA/SCA10/065-073

[46] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Seoul, Republic of Korea, 90–106. DOI: https://doi.org/10.1145/3503221.3508431

[47] José Oliver, Carlos Álvarez, Teresa Cervero, Xavier Martorell, John D. Davis, and Eduard Ayguadé. 2023. Accelerating SpMV on FPGAs through block-row compress: A task-based approach. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Gothenburg, Sweden, 151–158. DOI: https://doi.org/10.1109/FPL60245.2023.00029

[48] 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)*. IEEE, Vienna, 724–736. DOI: https://doi.org/10.1109/HPCA.2018.00067

[49] 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. DOI: https://doi.org/10.1145/3140659.3080254

[50] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: Accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, San Diego, CA, USA, 362–375. DOI: https://doi.org/10.1145/3332466.3374521

[51] Michail Pligouroudis, Rafael Angel Gutierrez Nuno, and Tom Kazmierski. 2020. Modified compressed sparse row format for accelerated FPGA-based sparse matrix multiplication. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Seville, Spain, 1–5. DOI: https://doi.org/10.1109/ISCAS45731.2020.9181266

[52] Jeff Pool. 2020. Accelerating sparsity in the NVIDIA Ampere architecture. *GTC 2020* (2020). Retrieved from https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22085-accelerating-sparsity-in-the-nvidia-ampere-architecture%E2%80%8B.pdf

[53] Yousef Saad. 1992. *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, Manchester, UK.

[54] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Montreal, QC, Canada, 1099–1112. DOI: https://doi.org/10.1109/HPCA56546.2023.10071015

[55] Santiago Segarra, Antonio G. Marques, Gonzalo Mateos, and Alejandro Ribeiro. 2017. Network topology inference from spectral templates. *IEEE Transactions on Signal and Information Processing over Networks* 3, 3 (2017), 467–483. DOI: https://doi.org/10.1109/TSIPN.2017.2731051

[56] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2020. Synergistic CPU-FPGA acceleration of sparse linear algebra. arXiv:2004.13907, 1–12. Retrieved from https://doi.org/10.48550/arXiv.2004.13907

[57] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022a. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *59th ACM/IEEE Design Automation Conference (DAC)*. ACM, San Francisco, CA, USA, 211–216. DOI: https://doi.org/10.1145/3489517.3530420

[58] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022b. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Virtual Event, USA, 65–77. DOI: https://doi.org/10.1145/3490422.3502357

[59] 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)*. IEEE, Athens, Greece, 766–780. DOI: https://doi.org/10.1109/MICRO50266.2020.00068

[60] Zhuofu Tao, Chen Wu, Yuan Liang, Kun Wang, and Lei He. 2022. LW-GCN: A lightweight FPGA-based graph convolutional network accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 16, 1 (2022), 1–19. DOI: https://doi.org/10.1145/3550075

[61] Erfan Bank Tavakoli, Michael Riera, Masudul Hassan Quraishi, and Fengbo Ren. 2024. FSpGEMM: A framework for accelerating sparse general matrix–matrix multiplication using Gustavson's algorithm on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 4 (2024), 633–644. DOI: https://doi.org/10.1109/TVLSI.2024.3355499

[62] James Theiler, Guangzhi Cao, Leonardo R. Bachega, and Charles A. Bouman. 2011. Sparse matrix transform for hyperspectral image processing. *IEEE Journal of Selected Topics in Signal Processing* 5, 3 (2011), 424–437. DOI: https://doi.org/10.1109/JSTSP.2010.2103924

[63] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv:1909.01315, 1–18. Retrieved from https://doi.org/10.48550/arXiv.1909.01315

[64] Chen Wu, Zhuofu Tao, Kun Wang, and Lei He. 2022. SkeletonGCN: A simple yet effective accelerator for GCN training. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Belfast, United Kingdom, 445–451. DOI: https://doi.org/10.1109/FPL57034.2022.00073

[65] Xilinx. 2023a. xbutil Utility. Retrieved from https://www.xilinx.com/video/software/xilinx-board-utility-introduction.html

[66] Xilinx. 2023b. Xilinx Power Estimator. Retrieved from https://www.xilinx.com/products/technology/power/xpe.html

[67] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. 2024. FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs. In *2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, Monterey, CA, USA, 223–234. DOI: https://doi.org/10.1145/3626202.3637562

[68] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. 2021. PanGu-α: Large-scale autoregressive pretrained Chinese language models with auto-parallel computation. arXiv:2104.12369, 1–23. Retrieved from https://doi.org/10.48550/arXiv.2104.12369

[69] Bingyi Zhang and Viktor K. Prasanna. 2023. Dynasparse: Accelerating GNN inference through dynamic sparsity exploitation. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, St. Petersburg, FL, USA, 233–244. DOI: https://doi.org/10.1109/IPDPS54959.2023.00032

[70] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Virtual Event, USA, 687–701. DOI: https://doi.org/10.1145/3445814.3446702

[71] 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)*. IEEE, San Diego, CA, USA, 261–274. DOI: https://doi.org/10.1109/HPCA47549.2020.00030

Appendix

A Abbreviations

Table A1. Abbreviations and Meanings Appearing in This Document

Abridge	Hidden Meaning	Abridge	Hidden Meaning
ACC	Accumulators	ISN	Index Shuffle Network
ALU	Arithmetic Logic Unit	LSA	Load Store Adapter
ASICs	Application-Specific Integrated Circuits	LLM	Large Language Model
AIGC	Artificial Intelligence Generative Content	LUT	Lookup Table
b8c	Block-8 Compress	MAC	Multiplication and Accumulation
BRAM	Block RAM	MCSR	Modified Compressed Sparse Row
BV	Bit-Vector	NLP	Natural Language Processing
CBV	Compressed Bit-Vector	nzz	Nonzero Elements
C^2SR	Cyclic Channel Sparse Row	NoC	Network on-Chip
COO	Coordinate	OPCOO	Optimized PCOO
CNNs	Convolutional Neural Networks	PCOO	Packet-Level Column-Only Coordinate-List
CPCOO	Compact PCOO	PEGs	Processing Engine Groups
CPUs	Central Processing Units	PEs	Processing Elements
CSC	Compressed Sparse Columns	RAW	Read-after-Write
CSR	Compressed Sparse Rows	ReMCOO	Redundant Modified Coordinate
CSV	Compressed Sparse Vector	SCP	Sparse Computing Pipelines
CSVecBuf	Cluster Shared Vector Buffe	SCSR	Symmetric Compressed Sparse Row
CVBV	Compressed Variable-Length Bit-Vector	SOTA	State-of-the-Art
DRC	Data Reuse-Aware Compression	SpDMM	Sparse-Dense Matrix Multiplication
DSPs	Digital Signal Processors	SPs	Stream Processors
DGL	Deep Graph Library	SpGEMM	Sparse Matrix-Matrix Multiplication
EB	Elastic Buffers	SpMM	Sparse Matrix Multiplication
FF	Flip-Flop	SpMV	Sparse Matrix-Vector Multiplication
FPGA	Field-Programmable Gate Arrays	SQ	Sparse Queue
GANs	Generative Adversarial Networks	SSpMV	Symmetric Sparse Matrix-Vector Multiplication
GEMM	General Matrix Multiplication	SSS	Symmetric Sparse Skyline
GNNs	Graph Neural Networks	TMM	Matrix Multiplication with Transposed
GPUs	Graphics Processing Units	TPU	Tensor Processing Unit
HBM	High-Bandwidth Memory	URAM	Ultra RAM
HPC	High-Performance Computing	VAUs	Vector Access Units
HRB	Hazard Resolving Backpressure units	VCs	Virtual Channels
IP	Intellectual Property		

Received 21 January 2024; revised 11 June 2024; accepted 13 July 2024