FVSpGEMM: A Comprehensive Framework for Exploiting Versal ACAP in High-Performance SpGEMM Acceleration

Kai Shi, Zhe Lin, Xinya Luan, Jianwang Zhai, Kang Zhao,
Member IEEE

***Abstract*—Sparse general matrix-matrix multiplication (SpGEMM) serves as a fundamental operation in real-world applications such as deep learning. Different from general matrix multiplication, matrices in SpGEMM are highly sparse and therefore require a compact representation. This places an additional burden on data preprocessing and exchanging, and also causes irregular memory access patterns, which can in turn lead to communication and computation overheads. To tackle these bottlenecks, we present FVSpGEMM, a system-level hardware accelerator framework that advances the VSpGEMM design, tailored and optimized on Versal ACAP for SpGEMM. Firstly, a new storage format called BCSX is proposed to offer a unified and block-wise compression strategy to deal with both row-major and column-major representations of non-zero data, enabling fixed-pattern memory accesses and effective data preloading. Secondly, a multi-level tiling mechanism is introduced to decompose the holistic SpGEMM into multiple computation granularities that fit into the AI Engines (AIEs) on Versal in a hierarchical manner, enhancing data reuse. Thirdly, a hybrid partitioning scheme is presented to orchestrate both the AIEs and programmable logic (PL) for intermediate product merging, which together resolve the issues of high memory utilization and communication demand. Fourthly, a systematic design abstraction model is composed for navigating the complexity of tiling and partitioning configurations. Fifthly, a fast and reliable auto-tuner is designed for obtaining the optimal configuration under peak performance. Experimental results demonstrate that FVSpGEMM achieves a 3.03× speedup over the state-of-the-art (SOTA) GEMM designs, while delivering performance gains of 19.14% and 22.53% over VSpGEMM on Versal for INT16 and FP32 precisions, respectively. Furthermore, regarding energy efficiency, FVSpGEMM exhibits an average 31.10× improvement compared to cuSPARSE on an RTX 4090 GPU, alongside gains of 12.61% and 16.35% over VSpGEMM, conclusively validating the efficacy of the proposed accelerator.**

I. Introduction

GENERAL matrix-matrix multiplication (GEMM) is a widely used operation defined as $A \times B = C$, characterized by accessing matrices A and B in a fixed pattern. However, in many real-world applications, matrices usually have high sparsity, meaning most elements are zeros. In these cases, the conventional GEMM solution becomes inefficient due to the large number of redundant computations performed on zero elements. This inefficiency highlights the need for SpGEMM, where the matrices involved in the computation are sparse. SpGEMM must handle non-zero elements with random spatial locations, which poses challenges including the increased communication overhead of transferring intermediate products and the limited computational intensity with unpredictable memory access to index the non-zero elements from compressed matrix elements, which can severely slow down the applications.

In academia, some methods have been proposed to accelerate SpGEMM on different hardware platforms, including CPUs [1], [2], GPUs [3]–[9], and FPGAs [10], [11]. These solutions mainly focus on the optimization of three aspects, namely, storage formats for storing and indexing sparse matrices [12], algorithms to perform matrix multiplication [7], and workload distribution methods [13]. CPUs and GPUs offer flexible control units and parallelized computation cores to optimize SpGEMM in all computation phases, but they incur high power consumption, which undermines energy efficiency. FPGAs enable customized operations for indexing elements from storage formats but they are limited by on-chip resources when the problems become complicated. Recently, AMD/Xilinx introduced the Versal Adaptive Compute Acceleration Platform (ACAP) [14], a heterogeneous system composed of AI Engines (AIEs), programmable logic (PL), and a processing system (PS). This platform excels in delivering flexible customization for hardware implementation, with computationally intensive multiplications handled by high-performance AIEs and complex control logic managed by PL, which is a promising substrate for efficient SpGEMM implementation.

Despite the high computational power provided by Versal, accelerating SpGEMM on Versal is non-trivial. The recent work CHARM [15] has proposed an efficient method to accelerate GEMM on Versal, which leverages the regular data access patterns of GEMM to achieve efficient pipelining. Nevertheless, new challenges arise when the matrices become sparse. The first challenge lies in excessive memory access to the local memory of AIEs, where the predominant row-wise computation pattern under compressed sparse row (CSR) or compressed sparse column (CSC) format contributes to irregular and unpredictable data accesses, leading to poor data reuse and avoiding data preloading when indexing elements in the local data memory of AIEs. These formats are depicted in

This work is supported by the Beijing Natural Science Foundation (No. L2603021, QY26369), the National Natural Science Foundation of China (No. 62404021, 62404257), the Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515013155, 2023A1515110769), the Fundamental Research Funds for the BUPT (No. 2025AI4S20), and the Research Initiation Project for Introduced Talents of BUPT (No. 510124058, 510224062). (*Corresponding author*: Kang Zhao.)

Kai Shi, Xinya Luan, Jianwang Zhai, and Kang Zhao are with the School of Integrated Circuits, Beijing University of Posts and Telecommunications, China (e-mail: shikai@bupt.edu.cn, luanxinya@bupt.edu.cn, zhaijw@bupt.edu.cn, zhaokang@bupt.edu.cn).

Zhe Lin is with the School of Integrated Circuits, Sun Yat-sen University, China (linzh235@mail.sysu.edu.cn).

Fig. 1. Second, the limited bandwidth on Versal and the high communication demand of transferring intermediate products produced from performing SpGEMM algorithm can trigger a communication bottleneck, severely hampering overall performance.

To fully unleash the computational power of Versal for SpGEMM acceleration, we propose five innovative optimization strategies: a new compressed storage format for efficient data processing, a multi-level tiling scheme for scalable matrix decomposition, a hybrid workload partitioning scheme to coordinate different computation resources, a comprehensive design abstraction model, and an automatic tuner for obtaining the optimal design configuration. These strategies collectively facilitate parallelism of computation, enhance memory management, and reduce communication overhead, finally delivering salient gains in performance and energy efficiency over existing SpGEMM implementations on mainstream computation platforms. The contributions of this paper are summarized as follows:

- We introduce a new compressed storage format, BCSX, unifying the representation of CSR and CSC in blocks. This storage format enables fixed-pattern data accesses, preserves high data locality, and supports data preloading.
- We propose a multi-level tiling scheme to hierarchically distribute the computation of SpGEMM to multiple AIEs while enhancing data reuse during computation.
- We develop a hybrid workload partitioning method that efficiently allocates the intermediate product merging operations to AIEs and PL, ensuring minimal communication overhead.
- We compose a design abstraction model for the proposed tiling scheme and partitioning strategy to systematically navigate the complexity between high-level design and physical constraints of hardware.
- We implement a fast and reliable auto-tuner within the heterogeneous components on Versal, determining the optimal parameters amid tiling scheme and partitioning method in this design.
- To the best of our knowledge, we present the first system-level framework for SpGEMM acceleration on Versal ACAP, integrating the proposed storage format, multi-level tiling, hybrid partitioning, design abstraction, and automatic tuning into a unified accelerator flow.

Compared with the preliminary version [16], this journal extension makes four major advances. Firstly, we introduce a systematic design abstraction model to formally capture the interactions among storage format, tiling hierarchy, partitioning strategy, and hardware constraints on Versal ACAP. Secondly, we develop an automatic tuner on top of this abstraction to efficiently search the high-dimensional design space and identify the best architecture configuration for each sparse workload. Thirdly, we substantially extend the experimental evaluation with broader system- and format-level analyses, including FP32 data type and additional representative datasets. Lastly, we further provide new studies on the practical overheads and applicability of BCSX, including host-side conversion overhead and architectural tradeoff analysis.

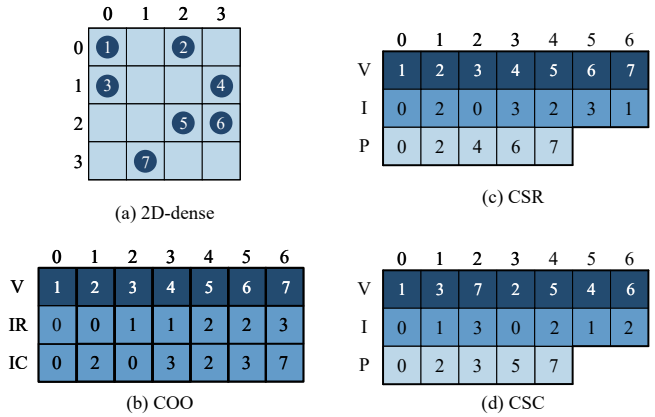


Fig. 1 Sparse matrix storage formats: (a) 2D dense matrix storage. (b) Coordinate storage format (COO). (c) Compressed sparse row (CSR) storage format. (d) Compressed sparse column (CSC) storage format. The values are marked in dark blue, pointers in steel blue, and indices and values in light blue.

II. Related Work

Previous works on accelerating SpGEMM using GPUs, CPUs, and FPGAs have proposed various optimizations, including hash tables and the ESC method. However, these approaches often suffer from poor data locality or high communication overhead. For instance, the work [5] on NVIDIA Pascal GPUs reduces scratchpad memory by grouping matrix multiplication (MM) computation tasks and predicting memory size for output matrices. However, it incurs significant overhead from probing hash tables and repeatedly accessing discontinuous memory under the CSR format. Recent GPU libraries such as spECK [7], OpSparse [8], and MOSparse [9] further optimize CSR-based SpGEMM inside a single CUDA device via hashing, binning, and adaptive size prediction, where kernel throughput is set as the primary metric. For example, OpSparse reports an average $7.35\times$ throughput speedup over cuSPARSE. The ESC method [17] enhances the parallelism of GPU cores by merging the sorted, index-adjacent MM results, but it faces substantial communication overhead due to the transfer of intermediate products generated during the expansion stage of ESC. FSpGEMM [11] improves data reuse in Gustavson's method [18] with row-wise computation pattern by introducing the compressed sparse vector (CSV) format and a row reordering strategy, maximizing on-chip resource utilization on FPGAs. Nonetheless, the CSV format does not fundamentally eliminate the overheads brought by the row-wise computation pattern, and the row reordering strategy introduces additional computational overhead. HASpGEMM [19] improves workload balance concerning computation and memory access through a micro-benchmarking scheme on asymmetric CPUs but remains constrained by the overheads inherent in the row-wise pattern. In summary, existing methods on SpGEMM

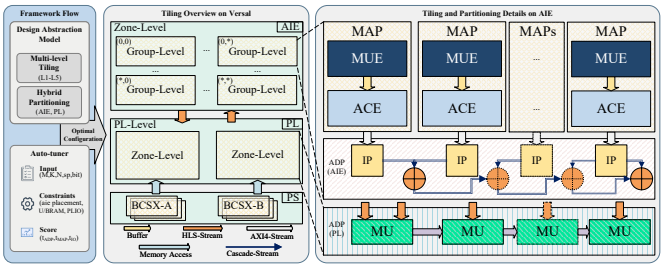


Fig. 2 Overall architecture of FVSpGEMM.

acceleration still fail to effectively address the problems of high communication costs and random memory access during computation.

On the Versal platform, CHARM [15] is the SOTA work on accelerating GEMM across all heterogeneous components of Versal, whereas it still struggles with sparse matrix processing. This is because the inherent strategies to process the dense matrices are incompatible with sparse matrix operations that primarily involve compact data representations with non-zero elements. In contrast, our approach seeks to establish fixed memory access patterns and reduce communication overhead for exchanging intermediate products in SpGEMM, which are essential for efficiently processing sparse matrices. Compared to the conference paper [16], this work proposes a systematic design abstraction model for navigating the complexity of tiling and partitioning and an automatic tuner for obtaining the optimal configuration for a system-level design.

III. Versal ACAP Architecture

The Versal ACAP is a heterogeneous platform that integrates AIEs, PL, and PS with a Network on Chip (NoC). The AIEs consist of very-long instruction word (VLIW) processors equipped with single instruction multiple data (SIMD) vector units, operating at a maximum clock frequency of 1.25 GHz, enabling high parallelism for computation-intensive applications. Each processor in the AIEs is denoted as an AIE tile, which comprises an AIE core for computation, a data memory for program storage and caching, and an AXI4-Stream switch [20] for stream accessing to the data memory.

Communications within the AIEs are categorized into two types: buffer-based access and stream-based access. The local data memory inside the AIE tile is composed of eight physical memory banks, allowing for horizontal or vertical data access to neighboring tiles. This configuration provides a four-way memory access, which exposes four local data memories to one AIE tile. The AXI4-Stream switches divide stream-based access into two modes: cascade-stream for horizontally adjacent tiles and normal streams for all tiles. Notably, the cascade-stream offers a 384-bit data width to transfer large data in one single clock cycle, while normal streams merely support a 32-bit data width per cycle. The PLIO interfaces are responsible for the communication between AIE and PL, each of which offers eight output channels and six input channels operating at the PL clock frequency with 64-bit data width for

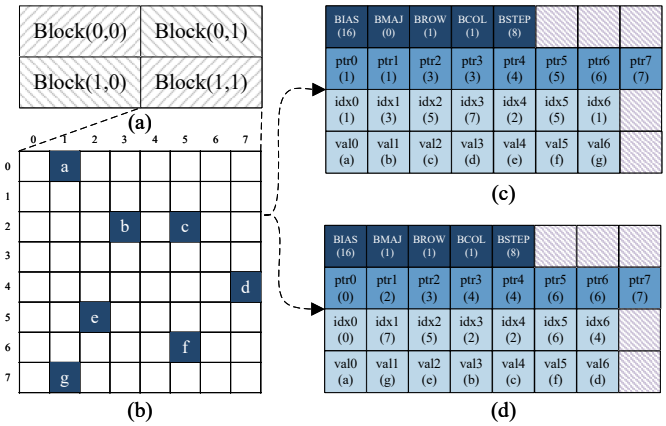


Fig. 3 BCSX Storage Format: (a) The structure of block-wise matrices in BCSX format in memory. (b) The matrix Block(1,1) in 2D dense form. (c) The BCSX format of Block(1,1) in row-major. (d) The BCSX format of Block(1,1) in row-major. Attribute descriptors in BCSX are marked in dark blue, pointers in steel blue, and indices and values in light blue.

both, totaling 1.2 TB/s for the VCK190 device with 39 PLIO interface tiles providing 78 connections on 128-bit stream. Data transmission between the PS and PL should use DDR memory as an intermediate buffer, with a bandwidth of 100 GB/s. Consequently, a communication gap arises between the high performance of AIE tiles and the limited PLIO and DDR bandwidth, which could become a bottleneck to deploying applications on Versal.

IV. Design Methodology

In this section, we introduce the acceleration paradigm, FVSpGEMM, whose overall architecture is depicted in Fig. 2. It decomposes the holistic computation process of SpGEMM into two types of partitions: the MAC-Partition (MAP) and the hybrid ADD-Partition (ADP).

The MAP conducts MMs at a fine-grained level, while the ADP merges the intermediate products to form the final results at a coarse-grained level. Based on this, the MAP can be further divided into a multiplication engine (MUE) and an accumulation engine (AUE) to perform the outer product for SpGEMM, which constitutes the main part of MM. The hybrid ADP is designed to be conducted partially in AIEs and PL to minimize the transmission of intermediate products generated by MAPs. The complete architecture of FVSpGEMM is supported by the proposed BCSX storage format, multi-level tiling scheme, and hybrid workload partitioning strategy.

A. BCSX Storage Format

The commonly used storage formats like CSR and CSC form the foundation of various works on SpGEMM since they preserve row-wise and column-wise data locality, respectively. The prior arts attempt to use a single storage format to construct the two input matrices (denoted as A and B), and adopt Gustavson's algorithm for SpGEMM. In such cases, although matrix A exhibits high data locality, matrix

TABLE I Features of BCSX compared to CSR and CSC.

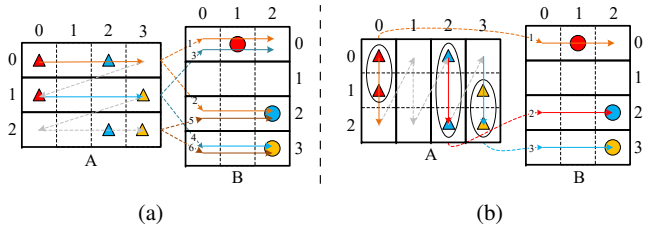
Storage Format \ Memory Access	Row Major	Column Major	Block-wise Structrue	Vectorized Loading
CSR	✓	×	×	×
CSC	×	✓	×	×
BCSX	✓	✓	✓	✓

B usually experiences irregular and unpredictable memory accesses, preventing effective memory optimizations such as data preloading and reuse. Moreover, CSR and CSC are monolithic representation methods that compress the entire matrix at once, making it difficult to partition and distribute computation workloads to hardware resources at a fine-grained level. To address these issues, in this paper, we propose a new storage format called Blocked Compressed Sparse eXtension (BCSX). This storage format aims to merge the strengths of these conventional storage formats, offer a consistent storage method for both row- and column-major representations, and facilitate data access in a regular and block-wise pattern. In the rest of this section, we illustrate the details of the BCSX format, the implementation of BCSX on AIE, and the memory access patterns supported by BCSX.

Construction of BCSX Format. BCSX utilizes five descriptors named BIAS, BMAJ, BROW, BCOL, BSTEP, and three arrays named idx*, ptr*, and val* to capture the structural information and non-zero elements in a per-block manner. Fig. 3 demonstrates BCSX with an 8×8 matrix as an instance.

The five descriptors identify the arrays in matrix blocks from memory and facilitate vectorized loading. The BIAS serves as a relative offset of the idx* array to the memory address of the current block. The BMAJ is a binary value that specifies whether the matrix is constructed in row-major (0) or column-major (1) order. When both the input matrices A and B are constructed in row-major order with BMAJs set to 0, a row-wise computation pattern is enabled. If the two BMAJs are set to 1 and 0 respectively, an inner product method can be applied. Conversely, when the BMAJs are set to 0 and 1, the outer product can be conducted, which is the adopted algorithm in this paper. With BMAJ to indicate the order, all computation patterns can be flexibly supported. The BROW and BCOL reveal the spatial coordinates of the current block within the entire sparse matrix, aiding both AIE and PL in organizing matrix blocks for multiplication and merging. The BSTEP determines the vector length of AIEs to fetch data from data memory, which could be adjusted in light of matrix sparsity and workload for a single AIE tile, thus enabling a dynamic load step for AIE tiles working on various matrix blocks.

The ptr* array records the number of non-zero elements in each line of the current block. Specifically, each element of ptr* records the cumulative number of non-zeros for the current and preceding rows or columns. The idx* and val* arrays store all the non-zeros of the current block in BMAJ order, with each element in these two arrays corresponding one-to-one.



(a) (b)

Fig. 4 Memory access patterns of two representative SpGEMM algorithms: (a) Gustavson's method; (b) Outer product. Gustavson's method involves six irregular memory accesses to matrix B, while the outer product has only three accesses in a fixed pattern with data locality.

Vectorized Loading on AIE. Different from CSR and CSC, BCSX skips the prefix 0 and captures the exact number of rows or columns in blocks to facilitate vectorized loading in AIE kernels. Within each AIE tile, memory access to BCSX for blocks in B is vectorized, which loads data in vectors of length BSTEP, which is a power of two. However, the prefix 0 in conventional formats impedes data alignment because the length of ptr* becomes non-divisible by a vector step, requiring an extra load to fetch the last few ptr* entries. In an AIE tile, the iterator for vector can only iterate at an entire vector step, causing misalignment when loading ptr* and leading to incorrect vector iterator for the idx* and val* arrays. To enable efficient vectorized loading in AIE tiles, BCSX eliminates the prefix 0, records the vector step in descriptor BSTEP, and utilizes the BIAS descriptor to mark the relative starting address of the idx* array, ensuring an aliquot length of ptr* for efficient vectorized loading.

In general, BCSX facilitates MMs on AIEs with the features shown in TABLE I.

Memory Access Patterns with BCSX. Suppose the input matrices are in square with non-zeros uniformly distributed across rows and columns, and let NNZ denote the average number of non-zeros per row. In row-wise production, as illustrated in Fig. 4(a), to calculate row $C_{(0,*)}$ of matrix C, the non-zeros $A_{(0,0)}$ and $A_{(0,2)}$ in row $A_{(0,*)}$ are loaded. For each non-zero in row $A_{(0,*)}$, the corressponding rows $B_{(0,*)}$ and $B_{(2,*)}$ are loaded as multipliers. Similarly, while calculating row $C_{(1,*)}$, row $B_{(0,*)}$ and $B_{(3,*)}$ are loaded, resulting in worse performance due to the random and discontinuous accesses to the same row $B_{(0,*)}$, preventing data preloading in AIE kernels, since the indexing of those non-zeros in matrix A is non-prior knowledge to AIEs. In contrast, the memory access becomes regular by using BCSX, as illustrated in Fig. 4(b), with each column $A_{(*,k)}$ and corresponding row $B_{(k,*)}$ loaded simultaneously for calculating the output matrix C. This allows non-zero $A_{(0,k)}$ and $A_{(1,k)}$ from $A_{(*,k)}$ to get multiplied by vector $B_{(k,*)}$, creating fixed memory access patterns to column $A_{(*,k)}$ and row $B_{(k,*)}$ due to the prior knowledge of k that is considered in design, which enables data preloading for columns in A and rows in B, thus opening up new opportunities for performance improvement.

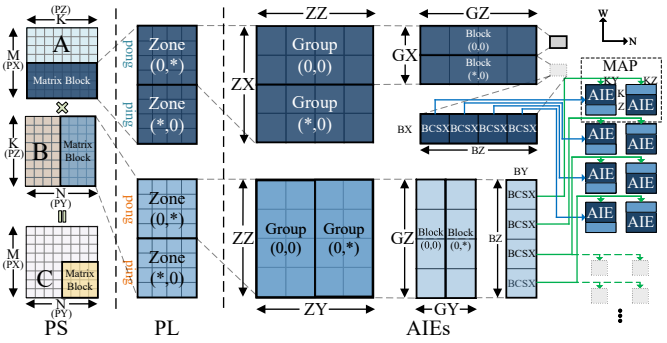


Fig. 5 Multi-level tiling scheme across AIEs and PL.

B. Multi-Level Tiling Scheme

We propose a multi-level tiling scheme that distributes the computation workload of SpGEMM to PL and AIEs. The key idea is to fully utilize the computational power of AIEs to perform multiplication and addition operations at a fine-grained level while taking advantage of the abundant memory resources of PL to accomplish the merging of intermediate results at a coarse-grained level. This tiling scheme can significantly reduce the *reuse distance* [21] of matrix elements from a matrix scale to a block scale, thereby enhancing performance by minimizing redundant data transfers. It is worth noting that the proposed tiling process is performed hierarchically, progressively breaking down the SpGEMM computation from the PL level to the zone level, then to the group level, block level, and finally, the kernel level, as shown in Fig. 5. Despite the three-level memory hierarchy of Versal ACAP, it's not naive to distribute the work directly into global DDR memory, PL on-chip RAMs, and AIE local memory. As will be discussed later, the additional Group-Level and Block-Level tilings are tailored for efficiently locating the ADP and MAP units to AIE array.

PL- and Zone-Level Tiling. Originally, matrix A with the size of $M \times K$ and matrix B with the size of $K \times N$ are stored in DDR in the BCSX format at a PL level. Matrix A and B are then divided into $PX \times PZ$ and $PZ \times PY$ zones, respectively, where each zone is transferred from DDR to on-chip BRAMs of PL and then mapped to the AIEs. As shown in Section III, the bandwidth of DDR is much smaller than that of the PLIOs. Hence, we allocate ping-pong BRAMs to store the zone data for both of the input matrices, as shown in Fig. 6. Specifically, one BRAM acts as a ping buffer to receive a zone from DDR, while the other functions as a pong buffer to produce a zone for AIEs, allowing parallel communication between DDR and PL, as well as between PL and AIEs.

Group-Level Tiling. We further divide the zones into smaller groups to enhance the parallel execution of AIEs. We note that the size of a zone may exceed the capacity of the AIE array. Therefore, the zones of matrix A and B are subdivided into $\frac{ZX}{GX} \times \frac{ZZ}{GZ}$ and $\frac{ZZ}{GZ} \times \frac{ZY}{GY}$ groups, respectively, where each group corresponds to a group-level region on AIEs. To obtain the result $Group_{(*,0)}$ for zone C, all groups from A and $ZZ \times Group_{(*,0)}$ from B must be sent to the AIEs. Consequently, these groups from zone B need to be transferred ZZ times to get multiplied with all the groups from zone A. Hence, using the on-chip BRAMs to store all groups of zone B, the entire ZZ times of accesses could be reused, avoiding repetitively loading data from the DDR in the monolithic methods. Furthermore, these groups of zone B need to get transferred to AIEs ZZ times as well, which heavily increases the communication burdens. Therefore, on the zone level, we utilize circuit switches to broadcast the $Group_{(*,0)}$ to group-level tiles on AIEs simultaneously, achieving an ZZ-fold data reuse ratio compared to non-titling scenes.

Block- and Kernel-Level Tiling. Matrices A and B at the group level are further divided into $GX \times GZ$ and $GZ \times GY$ blocks, respectively, where data accesses to A are column-wise and to B are row-wise at the block level. Finally, the blocks are then divided into kernel-level tiles, and the outer product method is performed at this fine-grained level. To be more specific, the outer product at the kernel level can be reduced to the scalar-vector multiplications using elements in columns from A and corresponding rows from B in a predetermined execution order. This approach boosts the computation efficiency of AIE cores by facilitating data reuse in a fixed pattern.

Again using Fig. 4(b) as an example, to complete the outer product, matrix A follows a per-column data fetching pattern. Both the two elements in the first column of A are fetched and multiplied with the element in the first row of B (marked as red triangles for A and circles for B, respectively). Since the access pattern is determined beforehand, the first row of B only needs to be fetched once, whereas Gustavson's computation pattern shown in Fig. 4(a) necessitates fetching the same row of B multiple times. For instance, the first row of B should be multiplied with both the first element in the two rows of A. However, this is a random way of fetching the rows in B due to the inherent irregularity of the spatial distribution of non-zeros from rows of A in Fig. 4(a), thus impeding data reuse. In summary, our method subtly leverages both the fixed memory access pattern enabled by BCSX and the outer product method of MM to maximize the data reuse during the data fetching of matrices A and B at the kernel level.

C. MAC-Partition and Hybrid ADD-Partition

The MM of $A_{M\times K} \times B_{K\times N}$ generates K batches of intermediate products (IP), which are sent to adders to be merged into the final results. We design MAPs on AIEs to perform detailed MMs using the outer product method with the support of BCSX. Nonetheless, merging all IPs to standalone adders within AIEs is impractical due to limited data memory and communication bandwidth for non-neighboring AIEs. The situation gets exacerbated when performing all the merges with frequent data transfers via the PLIOs between AIEs and PL. To address these issues, we propose the hybrid ADP to perform partial merges at fine-grained levels within AIEs to reduce the size of IP locally. The remaining merge operations are completed on PL to leverage the large on-chip memory, thereby further reducing the communication burden.

MAC-Partition on AIEs. The MUE at the kernel level produces sparse vectors, composing at most KZ batches of

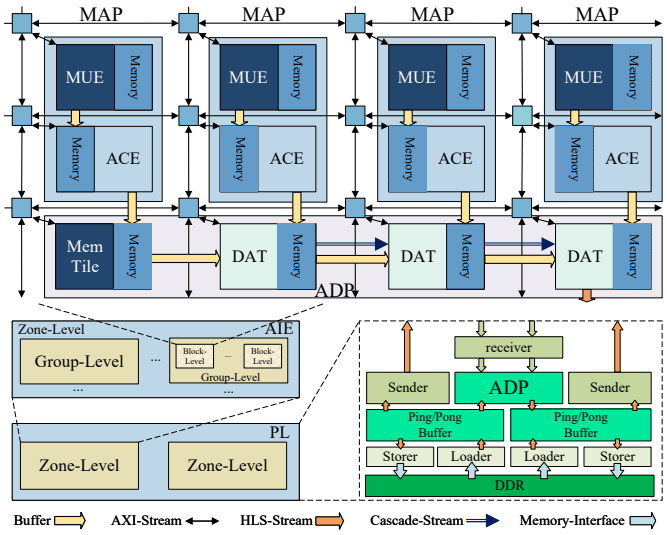


Fig. 6 Hybrid ADD-Partition on AIEs and PL.

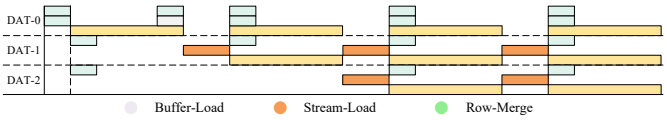


Fig. 7 Pipeline model of the cascading DATs.

IPs with the size of $KX \times KY$ when both the column from A and the corresponding row of B contain non-zeros. Once a batch of IP is produced, it is transferred to the ACE through AXI-Stream. The ACE accumulates the IPs to the data memory on the local AIE tile to form the first-stage IPs. Since the MUE and ACE are physically adjacent to AIEs, compared to transferring IPs entirely to PL through PLIOs or accumulating entirely on AIEs, the adjacent AXI-Switches enable fewer cycles for moving IPs to accumulators, significantly reducing communications latency. Besides merging operations in the accumulating stage, combining corresponding blocks for a result matrix requires additional transfers and data memory as well. Therefore, we propose the hybrid ADP to address this.

ADD-Partition on AIEs. As illustrated in Fig. 6, each ADP is positioned vertically above or below a MAP, where the MAP shares the same memory bank with the ADP to achieve maximum bandwidth in AIEs. Inside each ADP, dense accumulation tiles (DAT) are connected in cascade-stream to combine rows of each matrix simultaneously. Specifically, the first-stage IPs in the output from the first two MAPs are input into the first DAT through a single-cycle load via buffer-based communication manner. Subsequent DATs receive one IP stored in a shared memory bank from MAP and accept rows for accumulation from the cascaded preceding DATs, providing high communication bandwidth. The first DAT queries two IPs for merging and utilizes an additional neighboring AIE tile as shared memory storage, allowing parallel operation of these chained DATs. Consequently, the pipeline of this IP merging scheme at the kernel level is presented in Fig. 7. For each block level in AIEs, the ADP merges BZ blocks with $(BZ-1)$ DATs and outputs one block

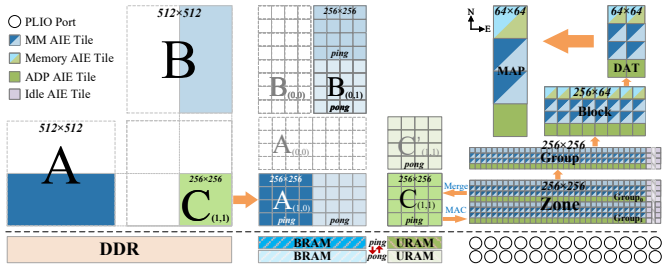


Fig. 8 Data view of tiling details across the memory hierarchy of Versal ACAP, with M, K, and N at 512.

as the second-stage IP to PL, reducing the communication amount of IPs at a fine-grained level.

ADD-Partition on PL. On the PL side, ADPs are instantiated in multiple Merging Units (MU). For each zone-level second-stage IPs produced from AIEs, MUs in ADPs of PL receive $(ZX \times ZZ \times ZY) \times (GZ \times GY)$ block-level IPs for merging, which operate sequentially with ping-pong buffers as illustrated in Section IV-B. The MUs are chained in a dataflow manner, with the header MU merging two IPs in row order, simultaneously forwarding the sum of each row to the next MU as an augend. This enables ADP on PL with the initiation interval (II) = 1, which allows for a minimum number of clock cycles between successive loop iterations, facilitating continuous data processing without stalls. Under the cooperation of ADPs across AIEs and PL, we achieve low latency with reduced communication burdens in merging IPs.

V. Design Abstraction

The synergy between the multi-level tiling scheme and our proposed hybrid partitioning strategy engenders a massive design space, where variations in tiling dimensions and partitioning configurations precipitate profound disparities in both throughput and resource efficiency. To systematically navigate this complexity and identify optimal configurations, we establish a quantizable design abstraction model. This framework abstracts the hierarchical tiling and partitioning processes, effectively bridging the gap between the high-level algorithmic decomposition and the physical constraints of the Versal ACAP architecture.

A. Tiling Abstraction

In Section IV, we introduce a hierarchical five-level tiling strategy designed to effectively orchestrate the SpGEMM workload across the heterogeneous components of the Versal ACAP. While the memory hierarchy of the Versal platform naturally presents three distinct levels: the global DDR memory, the BRAM and URAM on-chip resources from PL, and the local data memory within AIE tiles, where a naive mapping of the problem to a traditional three-level tiling scheme proves insufficient. Such a direct mapping often neglects the intricate physical constraints of the AIE array, such as the requisite spatial locality for inter-tile communication and the limited number of PLIO interfaces. The three-level tiling scheme selects a partial submatrix from the original input matrices in DDR and further splits the

TABLE II Tiling and Partitoning Parameters in Design Abstraction.

Tiling Level	Tiling Param.	Description
L1	P_X, P_Z, P_Y	Tiling input matrices of M, K, and N into $P_X + P_Y$ block matrices
L2	Z_X, Z_Z, Z_Y	Tiling block matrices in URAMs on PL into Zones, bufferized in ping-pong banks
L3	G_X, G_Z, G_Y	Tiling zones at full AIE array into logical groups on a portion of AIE array
L4	B_X, B_Z, B_Y	Tiling groups into blocks on a 2D physical layout in AIE array
L5	K_X, K_Z, K_Y	Tiling blocks into kernel AIE tiles placed on horizontally neighboured AIE tiles

submatrices on the second tiling step. For instance, the MAP units operate on tuples that necessitate collaboration between horizontally adjacent AIE tiles; a standard tiling approach fails to account for this topological dependency, leading to unbalanced workloads and severe underutilization of the AIE array. To bridge the gap between high-level algorithmic decomposition and the physical constraints of the hardware, we augment the traditional scheme with two additional tiling layers. These intermediate levels are specifically tailored to rearrange submatrices into logical groups and physical blocks, ensuring that the data distribution aligns with the 2D spatial layout of the AIE array and the bandwidth characteristics of the NoC. The fourth tiling level partitions the submatrices, denoted as groups, into multiple blocks that can be mapped to different AIE tiles for parallel processing. The fifth tiling level further splits these blocks into smaller chunks that can fit into the local memory of MAP.

To illustrate this hierarchical decomposition, consider an input matrix $\mathbf{A}$ with dimensions of 512×512, then the five-level tiling scheme can be represented as follows:

- Level-1 on DDR: At the coarsest granularity, the input matrix residing in DDR is tiled into submatrices of size 256×512, which are streamed to the PL in sequence through AXI-Streams.
- Level-2 on PL URAM: these submatrices are further partitioned into zones of size 256×256. These zones are buffered within PL BRAMs using ping-pong buffers, enabling the PL DMA engine to orchestrate parallel data transfers between the PL and the AIE array.
- Level-3 on AIE Array (logical): The decomposition then proceeds to the AIE-specific logical layer, where zones are divided into groups scaling 256×256. This logical partitioning is critical for dedicating independent PLIO interfaces to specific regions, thereby guaranteeing parallelization for blocks distributed within tightly coupled neighborhoods. (depicted in Fig. 8).
- Level-4 on AIE Array (2D-neighboured): Descending to the physical layout, each group is tiled into blocks of size 256×64, which are assigned to AIE tiles in a 2D-neighbored manner to maximize spatial locality.
- Level-5 on AIE Tiles (tuple): At the finest granularity, these blocks are split into chunks of size 64×64 that fit precisely within the local data memory of an AIE tile, allowing the MAP unit to perform efficient SpGEMM computation with minimized latency.

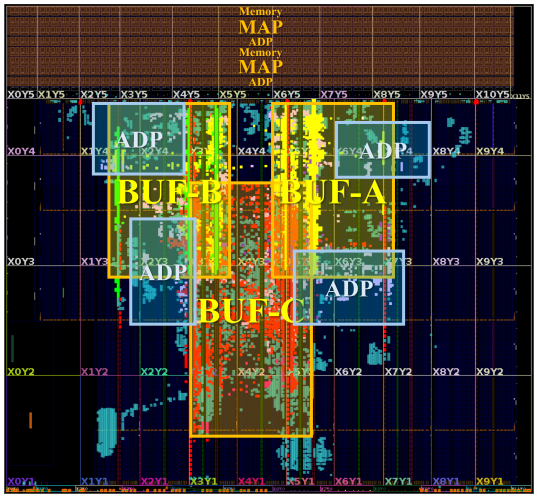


Fig. 9 Device view of system implementation for the 700×700 SpGEMM design from FVSpGEMM.

As delineated in Fig. 8, the proposed five-level tiling strategy systematically maps the input matrices onto the multi-tiered memory hierarchy of the Versal ACAP. The selection of tiling dimensions at each level is a critical design decision that necessitates rigorous deliberation to balance the competing constraints of limited on-chip memory capacity and the high bandwidth requirements of the AIE array. By carefully tuning these parameters, we ensure efficient data movement across the NoC and maximize the computational intensity within the MAP units. A comprehensive summary of the tiling factors utilized in our design is presented in TABLE II.

B. Partitioning Abstraction

In sparse matrix multiplication, the intermediate product generated during computation must be systematically merged to construct the final result matrix. As outlined in Section IV, our partitioning method divides the output matrices of MAPs into distinct partitions, each encapsulating a subset of intermediate dense matrices required for accumulation. The overarching objective of this method is to orchestrate a balanced workload distribution among AIE tiles while minimizing the latency penalties associated with data movement between the AIE array and PL memory. To achieve this equilibrium, the design must navigate the specific architectural constraints of the Versal ACAP memory hierarchy and communication fabric. Specifically, the AIE memory subsystem presents a stringent capacity limitation. Although each tile is equipped with eight physical memory banks of 4KB each, the requirement for simultaneous read and write operations necessitates bank pairing, effectively leaving only four logical banks available for buffering intermediate products produced by MAPs. Conversely, the PL domain provides extensive storage capacity via BRAMs and URAMs, it suffers from a significant frequency deficit, operating at approximately one-quarter the

TABLE III Best Configurations of AIE, PL Resource Utilization, and Tiling Parameters Explored from Auto-Tuner Framework

Sparse Matrix Datasets				AIE Attr.			PL Attr.				Tiling Params.				
Benchmark	$M \times N$	NNZ	Datatype	AIE Vec.	AIE Tiles	AIE Mem.	PLIOs	LUT	BRAM_18K	URAM	K	$B_{x,z,y}$	$G_{x,z,y}$	$Z_{x,z,y}$	$P_{x,z,y}$
football	115×115	663	INT16	8	176	632848	32	101171	377	128	32	1,4,1	2,1,2	2,1,2	1,1,1
			FP32	4	176	1054672	32	104242	249	128	32	1,4,1	2,1,2	2,1,2	1,1,1
TF11	216×236	1607	INT16	8	176	1812496	32	55202	377	128	64	1,4,1	2,1,2	2,1,2	1,1,1
			FP32	4	176	3168208	32	55817	377	128	32	1,4,1	2,1,2	1,1,1	4,2,4
GL6_D_10	163×341	2053	INT16	8	264	2663448	44	68092	377	192	64	1,4,1	2,1,2	3,1,2	1,1,1
			FP32	4	264	4752312	44	67973	377	192	64	1,4,1	2,1,2	1,1,1	3,1,2
Trefethen_500	500×500	4489	INT16	8	176	1800208	32	55882	377	128	64	1,4,1	2,1,2	2,1,2	2,2,2
			FP32	4	176	3195856	32	56497	377	128	32	1,4,1	2,1,2	1,1,1	8,4,8
Trefethen_700	700×700	6677	INT16	8	264	2718744	44	67822	377	192	64	1,4,1	2,1,2	3,1,2	2,3,3
			FP32	4	264	4793784	44	69004	377	192	32	1,3,1	2,1,2	1,1,1	12,8,12

speed of the AIE array, which creates a natural bottleneck for merging tasks. This disparity is further exacerbated by the limited communication bandwidth, where a single PLIO channel delivers only 1/12th the throughput of the intra-AIE cascade stream between horizontally neighboring AIE tiles. Furthermore, the physical restriction of 39 PLIO interface tiles severely constrains the column-wise placement of MAPs, risking data transfer stalls if interface allocation is not effectively managed. Consequently, the partitioning parameters must be meticulously calibrated to harmonize the computational throughput of the AIEs with the bandwidth limitations of the PL interface.

Accordingly, we define the number of partitions along each dimension of the input matrices, which are strictly aligned with the tiling dimensions established at Level-4. These parameters serve as the primary determinants for both the degree of concurrency executed by the MAP units and the data compression ratio realized within each partition for the subsequent ADP stage. While increasing the partitioning magnitude at Level-4 facilitates higher parallelism across the AIE tiles and improves the compression efficacy for the ADP, and it simultaneously introduces significant complexity regarding the physical placement of MAPs. This challenge is predominantly driven by the unbalanced spatial distribution of the PLIO interfaces, where an excessive demand for concurrent data transfers can saturate the limited routing resources and provoke interface conflicts.

Given the intricate dependencies between tiling dimensions and partitioning parameters to the specific micro-architectural characteristics of the AIE array and the Versal memory hierarchy, a manual selection of these variables is intractable. Therefore, a comprehensive design abstraction is essential to systematically navigate this expansive solution space and identify the optimal configuration that maximizes performance, while strictly adhering to the platform's rigorous resource constraints.

VI. Automatic Tuner

Leveraging the comprehensive abstraction model that encapsulates the storage format, multi-level tiling scheme, and hybrid partitioning strategy, we design an end-to-end auto-tuning framework. This framework is designed to systematically explore the design space and identify the optimal configuration that harmonizes algorithmic parameters with the underlying hardware constraints.

Algorithm 1 Auto-tuning for Tiling and Partitioning

Input: Matrix spec. (M, K, N, s, b), Hardware constraints $\mathcal{H}$ (AIE_{XY}, PLIO_{BW}, URAM/BRAM)
Output: Optimal configuration $\mathcal{C}^*$

```
Initialize score_min ← +∞
for all K_{X,Y,Z} ∈ {32, 64} do
    Generate tiling factors F = {P, Z, G, B} such that:
    // Construct Tiling Levels
        (M, K, N) = P · Z · G · B · (K_X, K_Z, K_Y)
    for all feasible tuples T ∈ F do
        // Verify placement and resource constraints
        C ← (T, K_{X,Y,Z})
        if not IsFeasible(C, H) then
            continue
        end if
        // Estimate latency using analytical models
        score ← t_AIE + t_PLIO + t_DDR
        if score < score_min then
            (score_min, C*) ← (score, C)
        end if
    end for
end for
return C*
```

By abstracting the tiling and partitioning parameters, we develop an auto-tuning framework, formalized in Algorithm 1, that systematically explores the design space to identify optimal configurations on the Versal ACAP. This framework leverages an analytical model to quantify the projected performance and resource utilization of each candidate configuration, effectively circumventing the prohibitive temporal overhead associated with the cycle-accurate AIE simulator.

The primary objective function targets the minimization of the global end-to-end execution time, which encompasses both the computational latency within the AIEs and PL, and the communication overhead incurred during data transfer across PLIO interfaces and the DDR memory subsystems. This score Equation (1) is exactly the optimization target used

in Line 13 of Algorithm 1.

$$\text{score} = t_{AIE} + t_{PLIO} + t_{DDR}. \tag{1}$$

As elucidated in Section IV, the total execution latency is predominantly governed by the throughput of the MAP and DAT units. These performance characteristics are determined by the kernel dimensions and are further modulated by the empirical coefficients α_{MAP} and α_{ADT}. These coefficients exhibit a linear dependency on the data bitwidth b and the operating frequencies of the AIE and PL domains, which are calibrated offline from a set of microbenchmarks under fixed data type and clock-frequency pairs. Specifically, we sweep representative kernel sizes and sparsity levels, measure the corresponding MAP and ADT latencies, and fit the linear coefficients used by the design abstraction. Consequently, the computational latency on the AIE array is estimated via:

$$\begin{aligned} t_{MAP} &= K_X \cdot K_Y \cdot s \cdot b \cdot \alpha_{MAP}, \\ t_{ADT} &= K_X \cdot K_Y \cdot s \cdot b \cdot \alpha_{ADT}, \\ t_{AIE} &= (t_{MAP} + t_{ADT}) \cdot \textstyle\prod_{i\in\{X,Y,Z\}} P_i. \end{aligned} \tag{2}$$

As shown in Equation (3), the tiling progress of both matrix A and B is unfolded at dimension K, with the data transferring time binding tightly with the tiling parameters, which can be estimated by tiling dimensions and the size p in padding BCSX format. To account for the architectural asymmetry in channel provisioning between the PL and AIE array, we differentiate the PLIO bandwidth capacities as $BW_{plio,in}$ and $BW_{plio,out}$ in Equation (4). This formulation allows for a precise estimation of PLIO transfer latency based on the aggregate data volume. Similarly, the communication overhead associated with global DDR memory is derived in Equation (5), comprising the top-level zone traffic between DDR and PL under the assumption of equivalent channel allocation for input and output streams.

$$\begin{aligned} N_{DIN} &= \textstyle\prod_{i\in\{X,Y,Z\}} P_i \cdot (dim_X + dim_Y) \cdot dim_Z \cdot p, \\ N_{DOUT} &= \textstyle\prod_{i\in\{X,Y,Z\}} P_i \cdot (dim_X + dim_Y) \cdot (K_X \cdot K_Y), \end{aligned} \tag{3}$$

where

$$\begin{aligned} dim_X &= Z_X \cdot G_X \cdot B_X, \\ dim_Y &= Z_Y \cdot G_Y \cdot B_Y, \\ dim_Z &= Z_Z \cdot G_Z \cdot B_Z. \end{aligned}$$

$$t_{PLIO} = \left(\frac{N_{DIN}}{BW_{plio,in}} + \frac{N_{DOUT}}{BW_{plio,out}}\right) \cdot b. \tag{4}$$

$$t_{DDR} = \frac{b \cdot P_Z \cdot dim_Z \cdot \sum_{i\in\{X,Y\}} P_i \cdot dim_i}{BW_{DDR,in}} + \frac{M \cdot N}{BW_{DDR,out}}. \tag{5}$$

Specifically, t_{DDR} models the zone-level data traffic between DDR and PL. The input part captures loading the tiled submatrices of A and B for one top-level traversal along the K dimension, whereas the output part captures writing back the final $M \times N$ result matrix. Sparsity is reflected in two places: the factor s in t_{MAP} and t_{ADT} models the effective nonzero workload during computation, while the packed size p in Equation (3) captures the transferred BCSX payload including padding. Tiling reuse is reflected by the separation of t_{PLIO} and t_{DDR}, where lower-level reuse inside PL on-chip memory and AIE groups is amortized before DDR transfer.

Upon establishing the analytical constraints, the autotuner conducts an exhaustive enumeration of all feasible configurations, strictly adhering to the physical limitations of AIE placement and the finite capacities of the BRAM/URAM resources within the PL. A system-level device view after synthesis is depicted in Fig. 9.

As systematically delineated in TABLE III, our framework performs a rigorous design space exploration to identify optimal configurations for each benchmark, elucidating the complex interplay between sparse matrix topologies, the microarchitectural attributes of the AIE and PL, and the hierarchical tiling parameters. To maximize the occupancy of AIE local memory and the computational throughput of the vector units, the framework dynamically tailors the kernel configuration specifically for INT16 and FP32 data types. Crucially, to accommodate the intrinsic irregularity of sparse data structures, we enforce a design preference for the minimum viable vector length. This strategy effectively minimizes the overhead associated with zero-padding, thereby mitigating the risk of invalid computations arising from interleaved rows and preventing out-of-bound memory accesses. Consequently, we standardize on a configuration of 128-bit vector width that effectively saturates the vector unit's computational capabilities while yielding vector sizes of eight for INT16 and four for FP32 to ensure precise data alignment.

VII. EXPERIMENTS

We evaluate the performance, power consumption, energy efficiency, and latency of FVSpGEMM. To demonstrate the effectiveness of FVSpGEMM, we compare our results with VSpGEMM [16], the SOTA GEMM work CHARM [15] on Versal ACAP, and the cuSPARSE [22] library on NVIDIA GPUs.

A. Experimental Setup

For experiments on the Versal ACAP, we use the VCK190 toolkit to evaluate FVSpGEMM and CHARM. These experiments are built with Vitis 2024.1, and results are collected from an onboard testbench. We set the clock frequency of AIEs at 1.25 GHz along with the clock frequency of PL obtained from the auto-tuner for FVSpGEMM, and use BEAM Tool [23] and xbutil [24] to evaluate the onboard power of VCK190. For comparisons with cuSPARSE, the experiments are conducted on an NVIDIA RTX 4090 GPU with CUDA 12.6 and an Intel Xeon Platinum 8375C CPU with 128 cores at 3.5 GHz for the host-side program. The reported GPU latency measures the computation with input matrices already resident in GPU global memory, and thus excludes additional host-device PCIe transfer overhead. We use nvidia-smi [25] to evaluate the power of the RTX 4090. Additionally, iterations for all of these experiments are configured for more than one minute to obtain a stable performance at runtime. The

TABLE IV Latency and Memory Utilization of Performing Different Sparse Storage Formats on a Single AIE Tile

Kernel Shape	Sparsity (%)	Datatype	2D		CSR		COO		CSR-CSC	
			Latency (ns $\times 10^6$)	Memory Util. (%)	Latency (ns $\times 10^6$)	Memory Util. (%)	Latency (ns $\times 10^6$)	Memory Util. (%)	Latency (ns $\times 10^6$)	Memory Util. (%)
32	5	INT16	22.73	41.14	9.17	19.25	10.83	19.45	46.33	19.25
	10		22.73		15.84	21.40	18.26	22.77	67.42	21.40
	5	INT32	21.94	78.64	9.17	19.25	14.90	36.73	42.05	35.66
	10		21.93		51.08	40.54	26.44	42.88	60.34	39.76
	5	FP32	32.46	79.42	17.39	35.56	13.50	36.15	63.58	35.56
	10		32.46		42.99	41.32	27.89	44.93	95.39	41.32
64	1	INT16	90.37	153.64	4.69	57.34	12.40	56.46	122.50	57.14
	5		142.60		30.86	64.17	35.16	67.00	261.10	64.17
	10		142.60		74.18	73.35	75.02	80.87	392.50	73.35
	1	INT32	87.00	303.64	19.40	147.57	14.84	108.80	103.10	110.27
	5		87.00		101.80	111.93	51.75	132.82	252.40	126.28
	10		87.00		341.40	127.06	137.60	163.68	380.70	146.79
	1	FP32	142.60	304.42	19.40	57.14	16.21	110.85	158.10	111.93
	5		142.60		84.61	127.55	53.32	134.29	355.70	127.55
	10		142.60		284.40	150.40	153.70	168.47	572.80	150.40

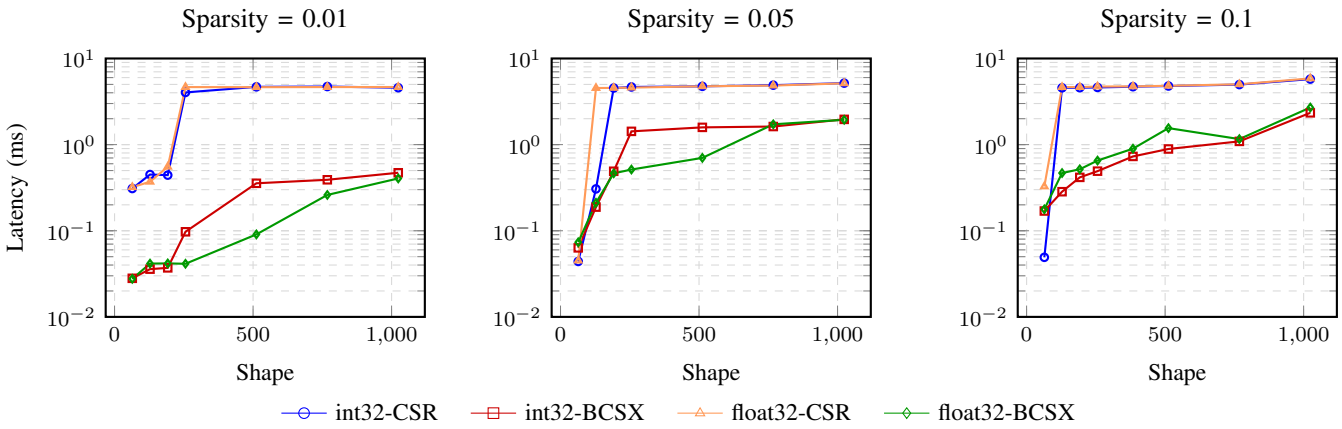


Fig. 10 Latency comparisons of different sparsity levels and shapes for CSR and BCSX storage formats on NVIDIA GPU.

backend implementation options of Vitis, including the target PL frequency and register-threshold settings, are kept fixed across all designs. In particular, the maximum of 39 PLIO interface tiles on VCK190 is treated as a fixed platform constraint rather than a tunable optimization objective, and the actual utilization of PLIOs is determined in tiling parameters, whose impact on the latency-resource tradeoff is analyzed in Section VII-D.

We utilize datasets with sparse matrices from the SuiteSparse Matrix Collection [26] whose sparsity and matrix shape range from 115 to 768, and 0.5% to 5.0%, respectively, covering most of the scenarios in real-world high-sparsity applications. Following the convention adopted by mainstream accelerator works such as MOSpGEMM [9], OpSparse [8], and spECK [7], we assume the input matrices are already stored in their target sparse format and the host-side format conversion time is excluded in the reported SpGEMM latency. The CHARM implementation was initially based on Vitis 2021.1, where some AIE APIs are deprecated, and the AIE clock operates at a maximum of 1.0 GHz, different from Vitis 2024.1, which operates at 1.25 GHz. Therefore, we port CHARM to Vitis 2024.1 to ensure a fair comparison.

B. Baseline Sparse Format Study on AIE

In this study, we compare four storage formats on a single AIE tile to identify the strongest baseline for the subsequent BCSX study. All formats are implemented with AIE kernels to avoid confounding effects from format-specific data movement.

As shown in TABLE IV, CSR provides the best overall latency-memory tradeoff among the conventional sparse formats due to its row-wise data locality and is therefore selected as the baseline for the following BCSX comparisons. More specifically, CSR-CSC consistently incurs the highest latency because its inner-product style execution leads to irregular accesses for both inputs and poor vector utilization on AIE. COO can be competitive for INT32 and FP32 due to index reuse in vector registers, but it still introduces extra loading overhead both in index fetch and memory occupation. Although CSR degrades as sparsity decreases due to increasingly irregular row-wise accesses, it remains the most stable conventional sparse format on AIE.

C. Applicability and Conversion Overhead

We next compare BCSX against CSR from two complementary aspects: architectural applicability and host-side

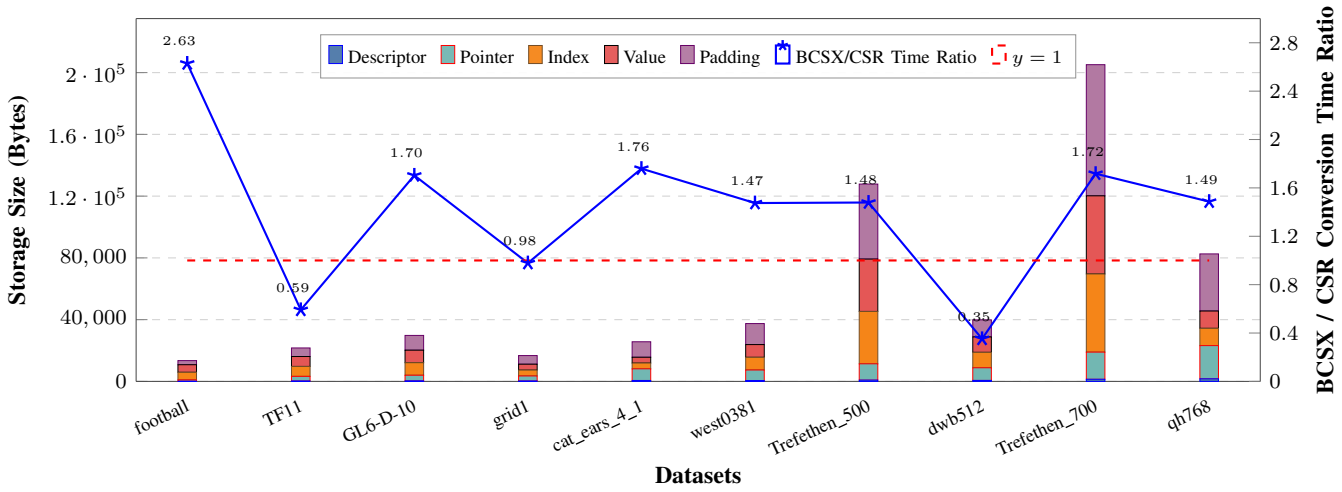


Fig. 11 BCSX storage size breakdown and conversion-time ratio over CSR on the SuiteSparse datasets under FP32 with per-line padding, BSTEP=4, and 64×64 blocking. The stacked bars show the BCSX storage components, and the right-axis line shows the conversion-time ratio of BCSX and CSR.

TABLE V Comparison of Conversion Overhead under Different Padding Modes and BSTEP Settings under FP32 Data Type.

Configuration		32×32					64×64				
Padding Mode	BSTEP	Avg. Time Ratio ↓	Median Time Ratio ↓	Wins ↑	Avg. Storage Ratio ↓	Avg. Padding Share ↓	Avg. Time Ratio ↓	Median Time Ratio ↓	Wins ↑	Avg. Storage Ratio ↓	Avg. Padding Share ↓
Block	1	1.348	1.042	4/10	1.504	0.00%	2.047	1.585	2/10	1.300	0.00%
	4	1.079	0.960	6/10	1.534	1.28%	0.826	0.894	7/10	1.289	0.53%
	8	1.284	0.956	5/10	1.553	3.00%	1.265	1.096	4/10	1.320	1.23%
Line	4	**1.449**	**1.361**	**2/10**	**2.561**	**40.22%**	**1.417**	**1.483**	**3/10**	**2.016**	**33.83%**

conversion overhead. The GPU-side applicability results are shown in Fig. 10, while the conversion-overhead results are summarized in Fig. 11 and TABLE V.

Applicability of BCSX. To validate the architectural applicability of our proposed BCSX format, we extend our evaluation to general-purpose GPUs, implementing the acceleration logic on NVIDIA CUDA cores. We conduct a rigorous comparative analysis against the widely adopted CSR format, sweeping across a diverse range of sparsity levels and matrix geometries. As illustrated in Fig. 10, BCSX consistently achieves superior end-to-end latency compared to CSR, with the performance differential becoming particularly pronounced at lower sparsity levels. With the increasing shape of matrices, the intensity of computation rises for the CSR designs from 256×256, thus reaching the computation boundary with a latency of about 4.6 ms. However, the designs of BCSX reach the compute boundary later and obtain lower latency with a speed up from 0.40 ms to 2.69 ms for 1% and 10% sparsities, respectively. This advantage stems from BCSX's inherent ability to regularize memory access patterns. By enforcing a unified block-wise structure, BCSX facilitates efficient memory coalescing and maximizes thread-level parallelism within matrix tiles, thereby unleashing the computational potential of the GPU. Conversely, the CSR format encounters the memory bandwidth wall significantly earlier across the spectrum of sparsity ratios. This bottleneck is precipitated by the massive, irregular indexing overhead required to traverse long and untiled matrix rows. In contrast, BCSX effectively mitigates this limitation through its built-in blocked coordinate design, which maps naturally to the GPU's thread block hierarchy. Collectively, these results underscore the versatility of BCSX, establishing it as a robust and efficient sparse storage format capable of delivering high performance across heterogeneous computing platforms.

Conversion Overhead of BCSX. To quantify the overhead in an offline conversion of BCSX, we benchmark the in-memory conversion from a normalized dense representation to BCSX and CSR on an additional representative set of ten SuiteSparse matrices spanning different shapes and densities. File parsing and I/O are excluded, hence the measurement isolates the pure conversion logic, and both converters share the same pre-allocation strategy and OpenMP backend with four threads. As shown in Fig. 11, BCSX incurs an average conversion-time ratio of 1.42 over CSR, and the per-dataset ratios range from 0.35 to 2.63. This variation is primarily explained by the interaction between block-level dynamic scheduling for BCSX and row-level static scheduling for CSR under non-uniform sparsity. On the largest and most skewed matrices such as dwb512 and TF11, the block-level dynamic schedule absorbs the load imbalance, whereas on small or uniformly distributed matrices such as football, CSR remains preferable. Overall, the BCSX packing cost stays within the same order of magnitude as that of CSR.

TABLE V additionally lists three block-padding configurations as a sensitivity study where the per-line BSTEP alignment is relaxed to a single per-block alignment. These configurations indicate the achievable lower bound of the conversion cost where unaligned access is featured. To summarize, the

TABLE VI Multi-Level Tiling Tradeoff under Fixed Merge Families for 64-bit Kernel and INT16 Data Type.

Sparse Matrix Datasets			Flattened Tile Level	Fixed Ratio		Tiling Params.				AIE Attr.		PL Attr.		
Benchmark	$M \times K \times N$	NNZ		B_z	P_z	$B_{x,y}$	$G_{x,z,y}$	$Z_{x,z,y}$	Latency (ms, ratio)	AIE Tiles	PLIOs	LUT	BRAM_18K	URAM
grid1	$252 \times 252 \times 252$	476	Zone	4	1	1,1	2,1,2	1,1,1	**0.40(100.00%)**	44	12	48743	377	32
			Group	4	1	1,1	1,1,1	2,1,2	**0.40(98.45%)**	44	20	43565	249	32
			Zone	4	1	1,1	4,1,4	1,1,1	**0.29(100.00%)**	176	24	72150	633	128
			Group	4	1	1,1	1,1,1	4,1,4	–	–	–	–	–	–
			normal	4	1	1,1	2,1,2	2,1,2	0.78(266.32%)	176	32	64768	377	128
west0381	$381 \times 381 \times 381$	2157	Zone	3	2	1,1	3,1,3	1,1,1	**1.01(100.00%)**	72	15	54383	409	72
			Group	3	2	1,1	1,1,1	3,1,3	1.00(98.57%)	72	27	51989	217	72
			Zone	6	1	1,1	3,1,3	1,1,1	0.73(100.00%)	153	21	69531	697	72
			Group	6	1	1,1	1,1,1	3,1,3	**0.70(96.53%)**	153	45	64905	313	72
			normal	6	1	1,1	2,1,2	3,1,1	**0.72(98.99%)**	204	36	75371	505	96
dwb512	$512 \times 512 \times 512$	2500	Zone	4	2	1,1	4,1,4	1,1,1	**0.62(100.00%)**	176	24	72102	633	128
			Group	4	2	1,1	1,1,1	4,1,4	–	–	–	–	–	–
			Zone	8	1	1,1	2,1,4	1,1,1	0.81(100.00%)	184	24	75455	889	64
			Group	8	1	1,1	1,1,1	2,1,4	**0.78(96.76%)**	184	56	72167	377	64
			normal	8	1	1,1	2,1,2	2,1,1	**0.81(100.47%)**	184	32	68527	633	64

* "–" indicates aiecompiler fails to satisfy the routing constraints.

TABLE VII Hybrid Add-Partition Tradeoff under Fixed Tiling Families for 64-bit Kernel and INT16 Data Type.

Sparse Matrix Datasets			Fixed Tiling				Hybrid Ratio			AIE Attr.		PL Attr.		
Benchmark	$M \times K \times N$	NNZ	$B_{x,y}$	$G_{x,z,y}$	$Z_{x,z,y}$	$P_{x,y}$	B_z	P_z	Latency (ms, ratio)	AIE Tiles	PLIOs	LUT	BRAM_18K	URAM
grid1	$252 \times 252 \times 252$	476	1,1	2,1,2	2,1,2	1,1	4	1	**0.32(100.00%)**	176	32	64768	377	128
							2	2	0.34(106.96%)	80	24	55978	249	128
cat_ears_4_1	$377 \times 313 \times 377$	938	1,1	3,1,3	1,1,1	2,2	6	1	**0.32(100.00%)**	153	21	201257	697	72
							3	2	0.35(108.45%)	72	15	119763	409	72
west0381	$381 \times 381 \times 381$	2157	1,1	3,1,3	1,1,1	2,2	6	1	**0.73(100.00%)**	153	21	69531	697	72
							3	2	1.00(137.93%)	72	15	54383	409	72
dwb512	$512 \times 512 \times 512$	2500	1,1	1,1,1	1,1,1	8,8	8	1	**1.75(100.00%)**	23	17	40249	377	8
							4	2	2.18(124.58%)	11	9	33727	249	8
			1,1	2,1,2	2,1,1	2,4	8	1	**0.80(100.00%)**	184	32	68527	633	64
							4	2	0.88(109.81%)	88	20	52901	377	64
qh768	$768 \times 768 \times 768$	2934	1,1	2,1,2	2,1,2	3,3	6	2	**1.21(100.00%)**	272	40	64658	505	128
							4	3	1.32(109.26%)	176	32	78953	447	128
							3	4	**1.18(97.04%)**	128	28	77922	313	128

host-side packing of BCSX is a one-time preprocessing step that lies outside the kernel-latency accounting of all related SpGEMM works and is fully amortized by the repeated kernel iterations of our main experiments.

D. Tradeoff Study of Architecture

We conduct two tradeoff studies to isolate the effects of the multi-level tiling scheme and hybrid add-partition. In each study, only one architectural dimension is varied while the remaining parameters are fixed.

To demonstrate architectural necessity of the multi-level tiling hierarchy, TABLE VI studies the tiling configurations under fixed merge families. FVSpGEMM achieves lower latency in most Group-Level flattened cases through parallel PLIO utilization. Conversely, Zone-Level flattening introduces higher latency overheads stemming from the queue-based packet-split mechanism. However, relying solely on Zone-Level tiling is infeasible, as an increased number of tiles leads to routing constraint violations, which is evident in the cases of grid1 and west0381. Hence, Group- and Zone-Level tiling occupy distinct positions in the latency-resource tradeoff space, and no single hierarchy dominates across all datasets, which highlights the essential role of multi-level tiling.

To explore the efficacy of varying add-partition ratios between the AIE and PL, TABLE VII adjusts B_z and P_z for controling the scale of ADP while keeping spatial tiling configurations constant for modulating the distribution of merge workloads. For datasets including grid1, cat_ears_4_1, west0381, and both dwb512 families, deploying a larger B_z and smaller P_z yields lower latency. This trend indicates that consolidating merge operations within the AIE is generally more efficient than offloading them to the PL. Nevertheless, the qh768 dataset presents a notable exception where $(B_z, P_z) = (3, 4)$ outperforms $(6, 2)$ by 2.96%, while reducing the required AIE tiles and PLIOs from 272 and 40 to 128 and 28, respectively. This phenomenon reveals that excessively concentrating merge workloads on the AIE can exacerbate resource pressure and degrade overall performance. Consequently, the optimal ratio for hybrid add-partition cannot be statically determined, which necessitates the auto-tuner to dynamically balance merge locality, PLIO constraints, and horizontal AIE placement costs.

TABLE VIII Onboard Throughput, Latency, Power, and Energy Efficiency (EE.) Comparisons under INT16 Data Type.

Benchmark	CHARM [15]						VSpGEMM [16]						This work (FVSpGEMM)			
	AIE Tiles	PLIOs	Throughput (GOPS)	Latency (ms)	Power (W)	EE. (GOPS/W)	AIE Tiles	PLIOs	Throughput (GOPS)	Latency (ms)	Power (W)	EE. (GOPS/W)	Throughput (GOPS)	Latency (ms)	Power (W)	EE. (GOPS/W)
football	144	64	26.53	0.36	22.28	1.19	58	12	46.64	0.21	15.45	3.02	82.69 (+77.29%)	0.24	21.71	3.81 (+26.19%)
TF11	144	64	179.59	0.36	23.26	7.72	176	32	435.65	0.26	24.20	18.00	448.31 (+ 2.91%)	0.24	22.39	20.03 (+11.25%)
GL6_D_10	144	64	147.28	0.36	22.41	6.57	264	44	568.01	0.26	28.19	20.15	582.39 (+ 2.53%)	0.25	26.84	21.70 (+ 7.69%)
Trefethen_500	198	146	339.17	0.55	28.23	12.01	176	32	1204.39	0.46	25.38	47.45	1303.43 (+ 8.22%)	0.41	23.75	54.89 (+15.67%)
Trefethen_700	192	96	1219.91	0.67	31.08	39.25	264	44	2055.49	0.64	28.64	71.77	2152.43 (+ 4.72%)	0.59	29.34	73.37 (+ 2.23%)

TABLE IX Onboard Latency, Power and Energy Efficiency (EE.) Comparisons under FP32 Data Type.

Benchmark	cuSPARSE [22]			VSpGEMM [16]			This Work (FVSpGEMM)			EE. Gain	
	Latency (ms)	Power (W)	EE. (GOPS/W)	Latency (ms)	Power (W)	EE. (GOPS/W)	Latency (ms)	Power (W)	EE. (GOPS/W)	V.S. cuSPARSE	V.S. VSpGEMM
football	39.21	89	0.08	0.21	16.95	5.71	0.25	22.79	5.47	68.38×	- 4.20%
TF11	28.60	90	0.21	0.32	24.39	11.74	0.30	25.21	12.41	59.10×	+ 5.76%
GL6_D_10	24.81	89	1.50	0.32	29.29	12.08	0.31	28.26	13.47	8.98×	+11.51%
Trefethen_500	25.67	92	2.58	0.65	25.71	23.23	0.55	25.69	29.31	11.36×	+26.17%
Trefethen_700	25.91	95	5.61	1.10	32.10	30.28	0.82	31.70	43.18	7.69×	+42.53%

E. Evaluation of Performance

In this experiment, we evaluate the performance of FVSpGEMM and compare our results with the SOTA GEMM implementation on Versal, the CHARM framework. The AIE kernels in FVSpGEMM framework are currently implemented using the INT16 and FP32 data types, so we limit the values in datasets to the range of 16-bit integer and 32bit floating-point for both input and output while maintaining the original spatial distribution of non-zeros. Auto-tuner schedules the number of MAPs based on the size of input matrices to efficiently utilize AIE tiles while maintaining high communication efficiency within MAPs. In current settings, the zone on AIEs performs SpGEMM with two maximum matrix dimensions of 256 or 384 per iteration, which are mapped to 176 or 264 AIE tiles, respectively. FVSpGEMM determines the usage of AIE tiles according to the input matrix size that is closest to the multiples of these two maximum dimensions, ultimately achieving optimal tile utilization. The experimental results are shown in TABLE VIII. As can be seen, for smaller matrices, FVSpGEMM utilizes fewer AIE tiles because the dimensions of each level in tiling are small, which can fit into a single MAP. As the problem size increases, a large number of MAPs are mapped into AIEs, and a higher level of parallelism is achieved, resulting in a performance boost. We can observe that FVSpGEMM consistently outperforms CHARM on different problem sizes, achieving an average speedup of 3.03× and outperforming VSpGEMM. And FVSpGEMM outperforms VSpGEMM with an average speedup of 19.14% and 22.53% on INT16 and FP32 data types, repectively.

F. Evaluation of Energy Efficiency

We investigate the latency, power, and energy efficiency of VSpGEMM in TABLE IX, with a comparison to cuSPARSE, CHARM, and VSpGEMM. Given that cuSPARSE does not support INT16 datatype, we set the datatype of cuSPARSE to FP32 to perform the calculation. The latencies of FVSpGEMM and CHARM both increase as the matrix scale increases, while the latency of cuSPARSE decreases due to the low utilization of SMs on GPU at small matrix scales. For all problem sizes, FVSpGEMM achieves the lowest latency and outperforms CHARM on Versal and cuSPARSE on GPU, which fully justifies the capability of the Versal platform on SpGEMM acceleration.

The power consumption of FVSpGEMM is primarily composed of AIE cores, the PL domain, and the PS domain [27]. Among all the platforms, FVSpGEMM achieves the lowest power consumption on average. Moreover, FVSpGEMM demonstrates the highest energy efficiency among all the problem sizes, with an average energy efficiency gain of 31.10×, 3.11×, and 16.35% compared to cuSPARSE, CHARM, and VSpGEMM on FP32 and INT16 data types. Therefore, the experimental results show that FVSpGEMM can achieve high performance and high energy efficiency at the same time, which is an ideal solution for real-world applications, especially on edge devices that have a stringent requirement for energy efficiency.

VIII. Conclusion

In this work, we design FVSpGEMM, an accelerator framework for SpGEMM on Versal ACAP. We introduce the BCSX format, a unified compressed storage format for storing sparse matrices in a block-wise row or column-major order, facilitating efficient memory access patterns with efficient data preloading. We also propose a multi-level scheme under BCSX to enhance data reuse across AIEs and PL hierarchically. Additionally, the hybrid ADD-partition method is developed to reduce communication overhead for intermediate products. As a result, on INT16 data type, our approach achieves an average speedup of 3.03× and 19.14% compared to CHARM and VSpGEMM. On FP32 data type, FVSpGEMM achieves an average speedup of 22.53% compared to VSpGEMM, and a 31.10× and 16.35% energy efficiency gain compared to cuSPARSE and VSpGEMM.

Acknowledgments

We acknowledge the use of GPT-5.4 exclusively for language editing and readability enhancements. All technical content, ideas, and experiments are entirely drafted by the authors.

References

[1] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, "Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020, pp. 293–303.

[2] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms," in *IEEE International Conference on High Performance Computing (HiPC)*, 2015, pp. 48–57.

[3] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Adaptive Sparse Matrix-Matrix Multiplication on the GPU," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 68–81.

[4] N. Bell, S. Dalton, and L. N. Olson, "Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods," *SIAM Journal on Scientific Computing (SISC)*, vol. 34, no. 4, p. C123–C152, Jan. 2012.

[5] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU," in *International Conference on Parallel Processing (ICPP)*, 2017, pp. 101–110.

[6] S. Luo, B. Wang, Y. Shi, X. Zhang, Q. Xue, and S. Ma, "Sparm: A Sparse Matrix Multiplication Accelerator Supporting Multiple Dataflows," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2024, pp. 122–130.

[7] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "Speck: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020, pp. 362–375.

[8] Z. Du, Y. Guan, T. Guan, D. Niu, L. Huang, H. Zheng, and Y. Xie, "OpSparse: a Highly Optimized Framework for Sparse General Matrix Multiplication on GPUs," *IEEE Access*, vol. 10, pp. 85 960–85 974, 2022.

[9] Y. Wang, H. Lin, B. Wei, J. Gao, and W. Ji, "Optimizing General Sparse Matrix-Matrix Multiplication on the GPU," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 22, no. 4, pp. 1–25, 2025.

[10] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022, pp. 65–77.

[11] E. B. Tavakoli, M. Riera, M. H. Quraishi, and F. Ren, "FSpGEMM: A Framework for Accelerating Sparse General Matrix–Matrix Multiplication Using Gustavson's Algorithm on FPGAs," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2024.

[12] Z. Xie, G. Tan, W. Liu, and N. Sun, "A Pattern-based SpGEMM Library for Multi-core and Many-core Architectures," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 1, pp. 159–175, 2021.

[13] C. Zhang, M. Bremer, C. Chan, J. Shalf, and X. Guo, "ASA: Accelerating Sparse Accumulation in Column-wise SpGEMM," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–24, 2022.

[14] AMD, "AI Engine Kernel Coding User Guide," 2024. [Online]. Available: https://docs.amd.com/r/zh-CN/ug1079-ai-engine-kernel-coding

[15] J. Zhuang, J. Lau, H. Ye, Z. Yang, S. Ji, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu *et al.*, "CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 17, no. 3, pp. 1–31, 2024.

[16] K. Shi, Z. Lin, X. Luan, J. Zhai, and K. Zhao, "VSpGEMM: Exploiting Versal ACAP for High-Performance SpGEMM Acceleration," in *ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[17] S. Dalton, L. Olson, and N. Bell, "Optimizing Sparse Matrix—Matrix Multiplication for the GPU," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, pp. 1–20, 2015.

[18] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[19] H. Cheng, W. Li, Y. Lu, and W. Liu, "HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors," in *International Conference on Parallel Processing (ICPP)*, 2023, pp. 807–817.

[20] AMD, "Versal AI Engine Documentation," 2024. [Online]. Available: https://docs.amd.com/r/en-US/am009-versal-ai-engine

[21] R. K. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 145–156.

[22] NVIDIA, "cuSPARSE Documentation," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/

[23] Xilinx, "BEAM Tool for VCK190 Evaluation Kit," 2024. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/973078551/BEAM+Tool+for+VCK190+Evaluation+Kit

[24] AMD, "VCK190 Evaluation Board User Guide," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd/Environmental

[25] NVIDIA, "NVIDIA System Management Interface Documentation," 2024. [Online]. Available: https://docs.nvidia.com/deploy/nvidia-smi/index.html

[26] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 2011.

[27] AMD, "Power Design Manager User Guide," 2024. [Online]. Available: https://docs.amd.com/r/zh-CN/ug1556-power-design-manager

Kai Shi is currently pursuing a Master's degree in the School of Integrated Circuits at the Beijing University of Posts and Telecommunications, Beijing, China. He received the B.S. degree from the School of Computer Sciences, Hangzhou Dianzi University, Zhejiang, China. His research interests include heterogeneous computation acceleration, compiler infrastructure of AI, and LLM inference. He has published papers in the DAC 2025 and FPT 2025.

Zhe Lin (Member, IEEE) received the B.S. degree from the School of Electronic Science and Engineering, Southeast University, China, in 2014, and the Ph.D. degree from the Department of Electronic and Computer Engineering at Hong Kong University of Science and Technology, Hong Kong, China, in 2020.

Dr. Lin is an Associate Professor with the School of Integrated Circuits, Sun Yat-sen University, China. Previously, he was an Associate Research Fellow with Peng Cheng Laboratory, China. Dr. Lin's research interests cover front-end design automation for FPGA, hardware-software co-design, and heterogeneous computing systems. Dr. Lin was a recipient of the Best Paper Award Nomination at ICCD 2024, DATE 2022 and FCCM 2019.

Xinya Luan is currently pursuing the Ph.D. degree with the School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China. He received the B.E. degree from the School of Computer Science, Xidian University, Shaanxi, China. His research interests include heterogeneous and reconfigurable computing architecture, programming models, and automation design. He has published papers at DAC 2025, and FPT 2025.

Jianwang Zhai is currently an Associate Professor in the School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing. He received the B.E. degree in communication engineering from Beijing Jiaotong University, Beijing, China, in 2018, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, in 2023. His research interests include architecture modeling, design space exploration, and physical design. He received the William J. McCalla Best Paper Award from ICCAD 2021, and the Best Paper Award Nominations from ASP-DAC 2023 & GLSVLSI 2025.

Kang Zhao received the Ph.D. degree in the department of computer science and technology from Tsinghua University, China in 2009. Then, he worked in Tsinghua University, Intel, Xilinx and AMD respectively. When working in Xilinx and AMD, he played as the senior staff manager and leader of the Vitis HLS product team. Now he is the professor of Beijing University of Posts and Telecommunications, and leads the EDA team. His research interests include FPGA and EDA, especially on high-level synthesis, logic synthesis, compiler techniques, placement and floorplan, and other VLSI CAD algorithms.