VSpGEMM: Exploiting Versal ACAP for High-Performance SpGEMM Acceleration

Kai Shi[1], Zhe Lin[2†], Xinya Luan[1], Jianwang Zhai[1] and Kang Zhao[1†]
[1]Beijing University of Posts and Telecommunications [2]Sun Yat-sen University
{shikai, luanxinya, zhaijw, zhaokang}@bupt.edu.cn, linzh235@mail.sysu.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132619

***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 bottlenecks. To break these bottlenecks, we present VSpGEMM, a hardware accelerator for SpGEMM that is tailored and optimized on Versal ACAP. Firstly, a new storage format called BCSX is proposed in VSpGEMM, which offers a unified and block-wise compression strategy to deal with both row-major and column-major representation 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. Experimental results demonstrate a 2.65× speedup over state-of-the-art (SOTA) GEMM design on Versal and an average 33.62× improvement in energy efficiency compared to cuSPARSE on RTX 4090 GPU, showing the efficacy of VSpGEMM.**

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]–[6], and FPGAs [7], [8]. These solutions mainly focus on the optimization of three aspects, namely, storage formats for storing and indexing sparse matrices [9], algorithms to perform matrix multiplication [10], and workload distribution methods [11]. 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) [12], 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 [13] 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 in the AIE memories. Second, the limited bandwidth in Versal and the high communication demand of transferring intermediate products of SpGEMM can trigger a communication bottleneck, severely hampering overall performance.

To fully unleash the computational power of Versal for SpGEMM acceleration, we propose three innovative optimization strategies: a new compressed storage format for efficient data processing, a multi-level tiling scheme for scalable matrix decomposition, and a hybrid workload partitioning scheme to coordinate different computation resources. 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 implementation on mainstream computation platforms. The contributions of this paper are summarized as follows:

†Corresponding author

- 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.
- To the best of our knowledge, we present the first attempt to optimize SpGEMM on Versal ACAP, finally achieving a 2.65× speedup over CHARM on Versal and a 33.62× energy efficiency gain against cuSPARSE on the RTX 4090 GPU.

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 overheads from probing hash tables and repeatedly accessing discontinuous memory under the CSR format. The ESC method [14] 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 intermedia products generated during the expansion stage of ESC. FSpGEMM [8] improves data reuse in Gustavson's method [15] with row-wised 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-wised computation pattern, and the row reordering strategy introduces additional computational overhead. HASpGEMM [16] 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 acceleration still do not effectively address the problems of high communication costs and random memory access during computation.

On the Versal platform, CHARM [13] is the SOTA work on accelerating GEMM across all heterogenous conponents 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.

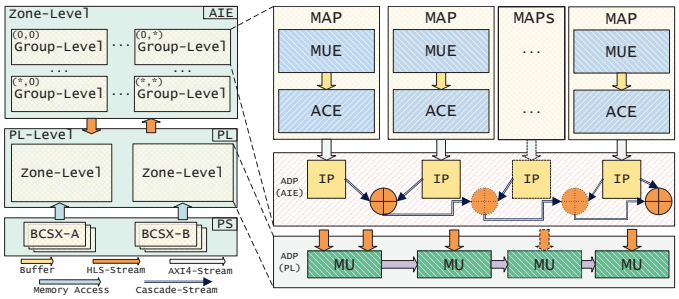

Fig. 1 Architecture of VSpGEMM

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 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 [17] for stream access 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 memory banks, allowing for horizontal or vertical data access to neighboring tiles. This configuration provides a four-way memory access which exposes four 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 64-bit output channels and six 64-bit input channels operating at the PL clock frequency, totaling 1.2 TB/s for VCK190 with 39 PLIOs. Data transmission between the PS and PL should use DDR memory as an intermediate buffer and the bandwidth is 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, VSpGEMM, whose overall architecture is depicted in Fig. 1. 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

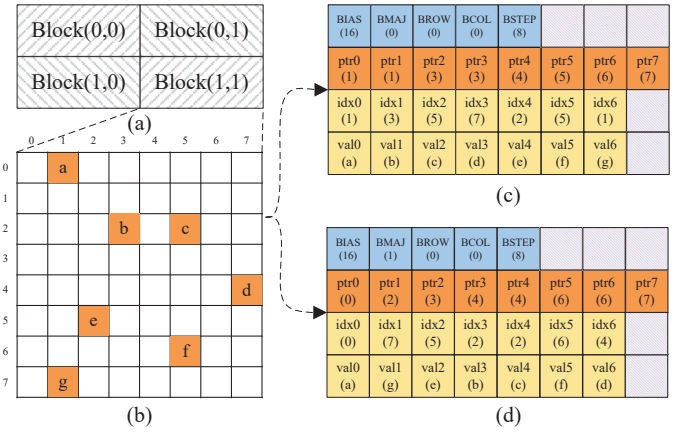

Fig. 2 BCSX Storage Format: (a) The structure of block-wise matrices in BCSX format in memory. (b) The matrix Block(1,1) in 2D dense form. (c) The BCSX format of Block(1,1) in row-major. (d) The BCSX format of Block(1,1) in column-major. Attribute descriptors in BCSX are marked in blue, pointers in red,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

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 VSpGEMM 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 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. 2 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.

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	✓	✓	✓	✓

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.

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. 3(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

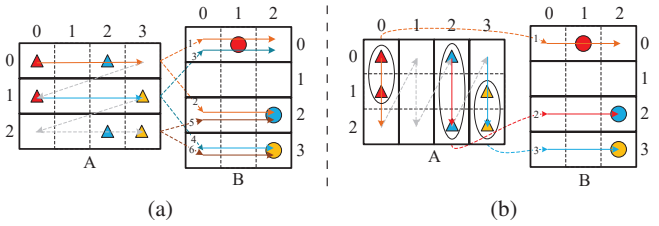

Fig. 3 Memory access patterns of two representative SpGEMM algorithms: (a) Gustavson's method; (b) Outer product. Gustavson's method involves six irregular memory accesses to matrix B, while the outer product has only three accesses in a fixed pattern with data locality.

$B_{(2,*)}$ 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. 3(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.

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 computation 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* [18] 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. 4.

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. 5. 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.

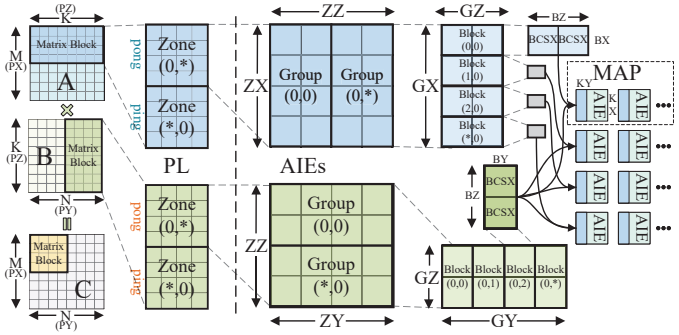

Fig. 4 Multi-Level Tiling Scheme across AIEs and PL

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 $\frac{ZX}{GX} \times \frac{ZK}{GK}$ and $\frac{ZK}{GK} \times \frac{ZZ}{GZ}$ groups, respectively, where each group corresponds to a group-level region on AIEs. To obtain the result $Group_{(0,0)}$ 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. 3(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 the Gustavson's computation pattern shown in Fig. 3(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. 3(a), thus impeding data reuse. In summary, our method subtly leverages

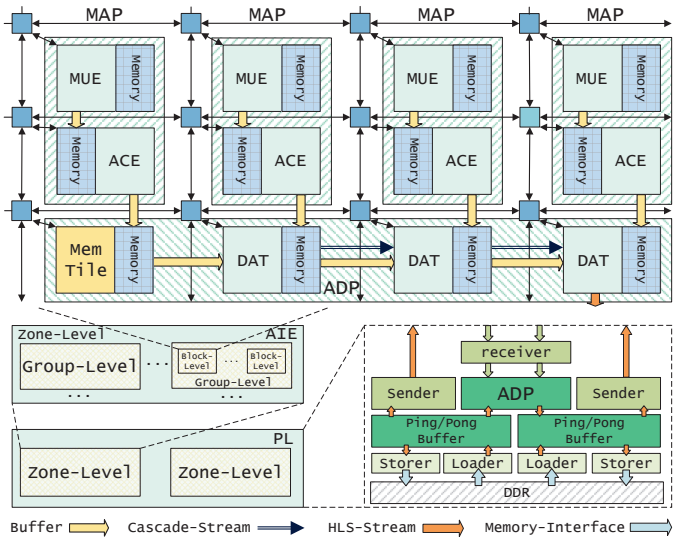

Fig. 5 Hybrid ADD-Partition on AIEs and PL

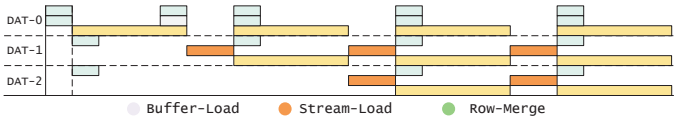

Fig. 6 Pipeline model of DATs

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 K batches of IPs with the size of $M \times N$ 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-Swithes 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. 5, 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 proceeding 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. 6. For each block level in AIEs, the ADP merges BZ blocks with $(BZ-1)$ DATs and outputs one block 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. EXPERIMENTS

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

A. Experimental Setup

For experiments on the Versal ACAP, we use the VCK190 toolkit to evaluate VSpGEMM and CHARM. These experiments are built with Vitis 2024.1, and results are collected from an onboard testbench. We set the PL running at 250 MHz and the AIEs at 1.25 GHz, and use BEAM Tool [20] and xbutil [21] 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. We use nvidia-smi [22] 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.

TABLE II Throughput of VSpGEMM

Datasets			CHARM [13]			VSpGEMM (Ours)			Speedup
Benchmark (ID-Name)	M × N	NNZ	AIE Tiles	PLIOs	Throughput (GOPS)	AIE Tiles	PLIOs	Throughput (GOPS)	
2397-football	115 × 115	663	144	64	26.53	58	12	46.64	1.76 ×
1945-TF11	216 × 236	1607	144	64	179.59	176	32	435.65	2.42 ×
1982-GL6_D_10	163 × 341	2053	144	64	147.28	264	44	568.01	3.85 ×
2209-Trefethen_500	500 × 500	4489	198	146	339.17	176	32	1204.39	3.55 ×
2210-Trefethen_700	700 × 700	6677	192	96	1219.91	264	44	2055.49	1.68 ×

TABLE III Latency, power and energy efficiency (EE) of VSpGEMM

Benchmark (ID-Name)	cuSPARSE [19]			CHARM			VSpGEMM (Ours)			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. CHARM
2397-football	39.21	89	0.08	0.36	22.28	1.19	0.21	15.45	3.02	37.75×	2.54×
1945-TF11	28.60	90	0.21	0.36	23.26	7.72	0.26	24.20	18.00	85.71×	2.33×
1982-GL6_D_10	24.81	89	1.50	0.36	22.41	6.57	0.26	28.19	20.15	13.45×	3.07×
2209-Trefethen_500	25.67	92	2.58	0.55	28.23	12.01	0.46	25.38	47.45	18.38×	3.95×
2210-Trefethen_700	25.91	95	5.61	0.67	31.08	39.25	0.64	28.64	71.77	12.79×	1.83×

We utilize datasets with sparse matrices from the SuiteSparse Matrix Collection [23], and we assume that matrices are pre-stored in the BCSX format, similar to the work on GPU [24]. 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. Evaluation of Performance

In this experiment, we evaluate the performance of VSpGEMM and compare our results with the SOTA GEMM implementation on Versal, the CHARM [13] framework. The AIE kernels in VSpGEMM are currently implemented using the INT16 datatype, so we limit the values in datasets to the range of 16-bit integer for both input and output while maintaining the original spatial distribution of non-zeros. VSpGEMM 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. VSpGEMM 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 II. As can be seen, for smaller matrices, VSpGEMM utilizes fewer AIE tiles because the dimensions of each level in tiling are small, which can fit into one 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 VSpGEMM consistently outperforms CHARM on different problem sizes, achieving an average speedup of 2.65×.

C. Evaluation of Energy Efficiency

We investigate the latency, power, and energy efficiency of VSpGEMM, with a comparison to cuSPARSE [19] and CHARM [13]. Given that cuSPARSE does not support INT16 datatype, we set the datatype of cuSPARSE to FP32 to perform the calculation. The latencies of VSpGEMM 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, VSpGEMM achieves the lowest latency and outperforms the CHARM on Versal and the cuSPARSE on GPU, which fully justifies the capability of the Versal platform on SpGEMM acceleration.

The power consumption of VSpGEMM is primarily composed of AIE cores, the PL domain, and the PS domain [25]. Among all the platforms, VSpGEMM achieves the lowest power consumption of 15.45W on the problem size of 115×115. Moreover, VSpGEMM demonstrates the highest energy efficiency among all the problem sizes, with an average energy efficiency gain of 33.62× and 2.74× compared to cuSPARSE and CHARM, respectively. Therefore, the experimental results show that VSpGEMM 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.

VI. Conclusion

In this work, we design VSpGEMM, 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 preload. 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, our approach achieves an average speedup of 2.65× compared to CHARM and a 33.62× energy efficiency gain compared to cuSPARSE.

Acknowledgment

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404257, 62404021), the Beijing Natural Science Foundation (No. 4244107, QY24216, QY24204), and the Guangdong Basic and Applied Basic Research Foundation (2024A1515013155).

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] 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.

[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] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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*, vol. 17, no. 3, pp. 1–31, 2024.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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 *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 145–156.

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

[20] 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

[21] AMD, "Vck190 evaluation board user guide," 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd/Environmental

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

[23] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 2011.

[24] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022, pp. 90–106.

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