AiSpGEMM: Accelerating Imbalanced SpGEMM on FPGAs with Flexible Interconnect and Intra-row Parallel Merging

Enhao Tang[1, 2], Shun Li[3], Hao Zhou[4], Guohao Dai[4], Jun Lin[2], Kun Wang[1, †]
[1]State Key Lab of Integrated Chips & Systems, and School of Microelectronics, Fudan University, Shanghai, China
[2]School of Electronic Science and Engineering, Nanjing University, Nanjing, China
[3]School of Integrated Circuits, Southeast University, Nanjing, China
[4]Shanghai Jiao Tong University, Shanghai, China
† kun.wang@ieee.org

Abstract*—The row-wise product algorithm shows significant potential for sparse matrix-matrix multiplication (SpGEMM) on hardware accelerators. Recent studies have made notable progress in accelerating SpGEMM using this algorithm. However, several challenges remain in accelerating imbalanced SpGEMM, where the distribution of non-zero elements across different rows is imbalanced. These challenges include: (1) the fixed dataflow of the merger tree, which leads to lower PE utilization, and (2) highly imbalanced data distributions, such as single rows with numerous non-zero elements, which result in intensive computations. This imbalance significantly challenges SpGEMM acceleration, leading to time-consuming processes that dominate overall computation time. In this paper, we propose AiSpGEMM to accelerate imbalanced SpGEMM on FPGAs. First, we improved the C^2SR format to adapt it for imbalanced SpGEMM acceleration based on the row-wise product algorithm. This reduces off-chip memory bank conflicts and increases data reuse of matrix *B*. Secondly, we design a reconfigurable merger (R-merger) with flexible interconnects to improve PE utilization. Additionally, we propose an intra-row parallel merging algorithm and its corresponding hardware architecture, the parallel merger (P-merger), to accelerate intensive operations. Experimental results demonstrate that AiSpGEMM achieves a geometric mean (geomean) speedup of 5.8× compared to the state-of-the-art FPGA-based SpGEMM accelerator. In Geomean, AiSpGEMM achieves a 3.0× speedup and a 9.8× improvement in energy efficiency compared to the NVIDIA cuSPARSE library running on an NVIDIA A6000 GPU. Moreover, AiSpGEMM-21 demonstrated a 4× increase in average throughput compared to the same GPU.

***Index Terms*—SpGEMM, FPGA, row-wise product, accelerator**

I. Introduction

Sparse-sparse matrix-matrix multiplication (SpGEMM), which involves multiplying two sparse matrices, has become a crucial computational kernel in machine learning and high-performance computing. Specifically, SpGEMM is essential for various graph algorithms and scientific computations. It plays a pivotal role in tasks such as compressed deep neural networks [14], graph contraction [4], triangle counting [1] and recommendation systems [8].

† Corresponding author.

This work was financially supported in part by National Key Research and Development Program of China under Grant 2021YFA1003602, and in part by Shanghai Pujiang Program under Grant 22PJD003 and Project of Shanghai EC under Grant 24KXZNA12.

Recently, the performance of current CPU and GPU solutions for SpGEMM has been limited due to the irregular computations and memory accesses, as well as their fixed architectures and deep memory hierarchies. Consequently, several specialized SpGEMM accelerators have been proposed to overcome these performance bottlenecks. Initially, most works used inner-product approaches [11]. However, these are not ideal for SpGEMM acceleration because sparse patterns influence the index intersection between the two input matrices, reducing data fetching efficiency. OuterSPACE [10] was the first outer-product-based SpGEMM accelerator, which avoids redundant data fetching. SpArch [16] introduced an on-chip real-time merge mechanism to reduce memory overhead and off-chip accesses in outer products. Recently, the row-wise product method [5] has emerged as a promising approach for SpGEMM due to its lower on-chip memory requirements and reduced off-chip traffic. MatRaptor [13], the first to use this method, improves efficiency by avoiding index matching and reducing dependence on on-chip memory. GAMMA [15] further enhances the row-wise algorithm's performance by reducing memory traffic through preprocessing.

Additionally, extending these approaches to embedded FPGAs, Li et al. [6] implement the row-wise product algorithm on a cache-based architecture to reduce bank conflicts on embedded FPGA devices. However, in memory-intensive applications such as SpGEMM, the inherent low data reuse and frequent random memory accesses impose significant demands on device memory bandwidth that embedded FPGAs cannot meet. In contrast, FPGAs equipped with High Bandwidth Memory (HBM) present opportunities for deploying efficient implementations of these memory-intensive tasks in data centers [17] [18] [19] [20].

However, thorough analysis of the access patterns in row-wise product algorithm for SpGEMM is crucial when designing on HBM-based FPGAs, especially when handling highly imbalanced matrices. Existing approaches are ineffective for accelerating imbalanced SpGEMM using the row-wise product algorithm, where non-zeros are highly imbalanced across different rows. For example, Fig. 1(c) shows the non-zero distribution in the *hangGlider_3* matrix used in the optimal control solver [2]. One row contains nearly 455 times more non-zeros than the average row. Such imbalanced matrices commonly appear in multiple application domains such as optimal control solver [2], circuit simulation [3], and natural language processing [3], which presents multiple new challenges for efficient SpGEMM acceleration.

(1) As shown in Fig. 1(b), in the row-wise product algorithm, each element a_{ij} in the i_{th} row of the first input matrix (matrix A) is multiplied by the j_{th} row (b_j) of the second input matrix (matrix B). This generates a partial result for the i_{th} row of the output matrix C (c_i), and all partial results are then merged to obtain the final c_i row. According to the row-wise product algorithm, the rows of matrix B can be reused for matrix A elements with the same column indices. However, irregular memory accesses can cause memory bank conflicts. Although the C^2SR format [13] effectively resolves bank conflicts, it does not leverage the opportunity to reuse the rows of matrix B.

(2) As shown in Fig. 1(a), due to the irregular data distribution of matrix A, the number of partial results of c_i becomes variable. This variability complicates parallel computation and hinders full pipeline utilization during the merging stage. For example, GAMMA proposed a high-radix merger tree that achieves high throughput and reduces overall computation latency. However, the uncertain number of partial results for c_i prevents the high-radix merger tree from fully utilizing PEs.

(3) Due to the irregular data distribution of matrix B, the lengths of the partial results for c_i also vary. Specifically, the presence of an evil row in matrix B, characterized by significantly more non-zero elements than other rows (*e.g.*, the *hangGlider_3* matrix in Fig. 1(c)), leads to similarly problematic evil rows in the partial results of c_i, as shown in Fig. 1(a). Existing approaches typically use inter-row parallelism to merge the partial results for c_i. However, when partial results of c_i include an evil row, the process becomes time-consuming and dominates the overall task (*e.g.*, partial results of c_1 in Fig. 1(a)).

To this end, we propose AiSpGEMM to accelerate imbalanced SpGEMM on HBM-based FPGAs. In summary, our work makes the following contributions:

- We propose an optimized C^2SR format that not only resolves memory bank conflicts but also leverages opportunities for reusing matrix B. Additionally, we observe that evil rows in imbalanced matrices lead to many elements in matrix A having the same column indices, even without preprocessing. Consequently, these evil rows in the imbalanced matrix increase the opportunities for reusing rows in matrix B.
- We propose a reconfigurable merger (R-merger). The R-merger balances PE utilization and parallelism through flexible interconnects, efficiently merging multi-set partial results of c_i in parallel and quickly outputting merged results. Unlike the high-radix merger tree, the R-merger can dynamically adjust data flow, ensuring optimal PE utilization across different data distributions, thereby improving overall computational performance.

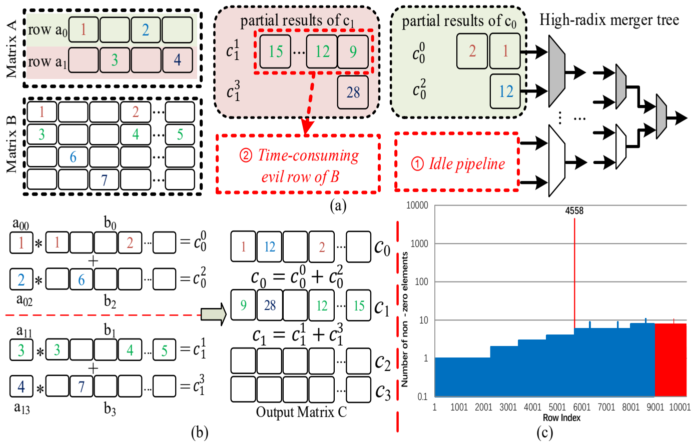


Fig. 1. The challenge of accelerating imbalanced SpGEMM. (a) idle pipeline and time-consuming intensive operations caused by imbalanced SpGEMM; (b) Example of SpGEMM data flow using row-wise product algorithm; (c) Highly imbalanced non-zero distribution of the *hangGlider_3* matrix.

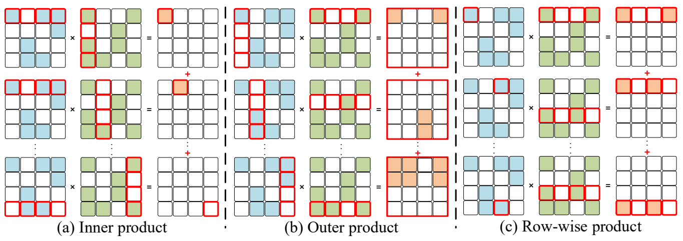


Fig. 2. Examples of three SpGEMM data flows. (a) inner product; (b) outer product; (c) row-wise product.

- We observed that the merging process for evil rows is highly time-consuming and dominates the overall task. To address this, we propose an intra-row parallel merging algorithm and its corresponding hardware architecture, the parallel merger (P-merger), to accelerate the merging process for evil rows.
- Experimental results demonstrate that AiSpGEMM achieves a geomean speedup of 5.8× (up to 9.6×) compared to the state-of-the-art (SOTA) FPGA-based SpGEMM accelerator. In Geomean, AiSpGEMM achieves a 3.0× speedup and 9.8× energy efficiency improvement compared to the NVIDIA cuSPARSE library running on an NVIDIA A6000 GPU. AiSpGEMM also achieves an average 4× higher throughput than the NVIDIA A6000 GPU.

II. Background

Currently, three different traversals can perform SpGEMM: inner product, outer product, and row-wise product. The characteristics of these dataflows are illustrated in Fig. 2.

The inner product is the widely used way to compute matrix multiplications. The inner product tries to compute every element of the output matrix C by calculating the inner product of a corresponding row of matrix A and a column of matrix B, which is effective for dense matrices. However, the inner product often encounters inefficiencies in index intersection due to the sparsity of matrices A and B. This inefficiency is exemplified in Fig. 2, where the index intersection between

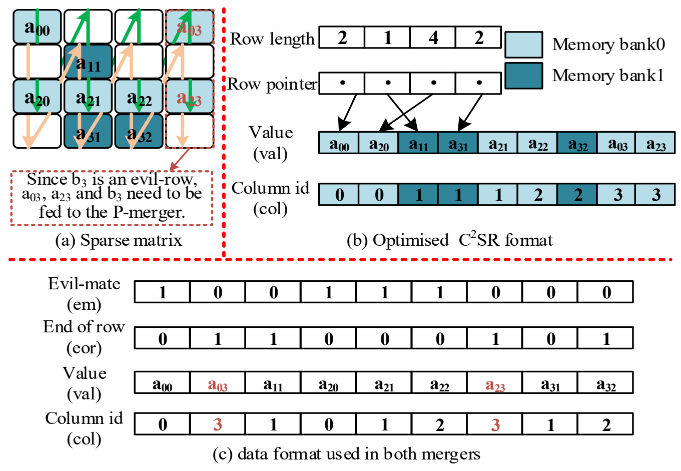


Fig. 3. Sparse data formats used in AiSpGEMM. (a)Sparse matirx; (b) Optimized C^{2}SR format; (c) merger-specific data format.

the last row of matrix A and the last column of matrix B is illustrated.

The outer product obtains many partial output matrices by iterating through the columns of matrix A and the rows of matrix B and then combining the partial output matrices to obtain the final output matrix C. Compared to the inner product, the outer product mitigates the inefficiencies in index intersection, since only entirely zero columns of A and rows of B do not contribute to the output, a relatively rare scenario. However, the outer product faces challenges in managing the size and number of partial output matrices, particularly when dealing with matrices that have highly irregular sparsity patterns.

The row-wise product multiplies each non-zero element a_{ij} in row a_i by the corresponding row b_j to generate a partial result for c_i, followed by merging all partial results to obtain the final output. This method efficiently handles index intersections and enables multiple processing elements to compute different rows of the sparse output matrix. Additionally, it minimizes on-chip memory usage by buffering only one row of C. Consequently, the row-wise product is a promising approach for SpGEMM.

III. Sparse Matrix Format

In AiSpGEMM, we use two sparse matrix formats: the optimized C^{2}SR format and a merger-specific data format (Fig. 3). The row-wise product method offers low on-chip memory usage and reduced off-chip traffic for matrix A, but increases off-chip traffic for matrix B due to irregular memory access. The C^{2}SR format reduces memory bank conflicts by assigning each row to a unique memory bank, but does not improve off-chip traffic for matrix B. Additionally, while the row-wise product algorithm arranges matrices in row-major order to reduce synchronization overhead, using row-major access for matrix A limits row reuse for matrix B. Accessing matrix A in column-major order maximizes row reuse but results in non-contiguous memory access, leading to performance loss.

Optimized C^{2}SR format. Therefore, we propose a dual-row column-major order access pattern, similar to a zigzag order, to achieve both row-major order calculations and maximize

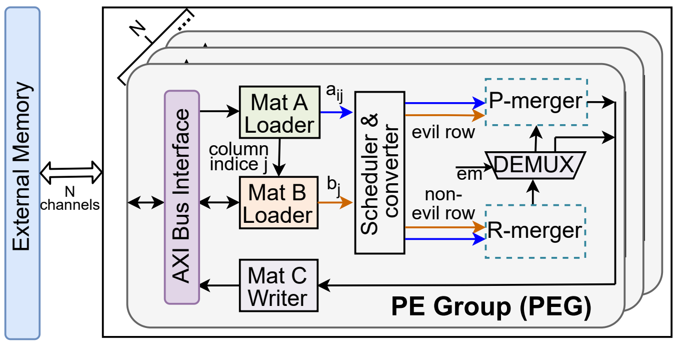


Fig. 4. AiSpGEMM architecture overview.

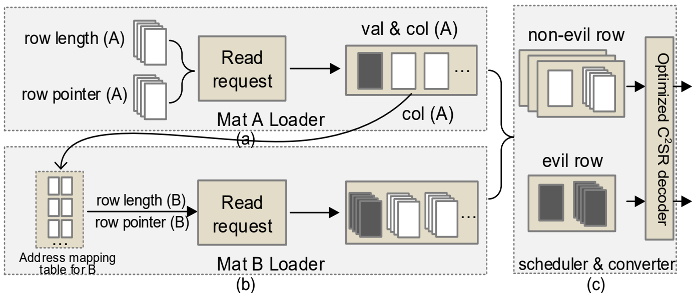


Fig. 5. Data access and scheduling schematic. (a) Mat A loader; (b) Mat B loader; (c) Scheduler and format converter.

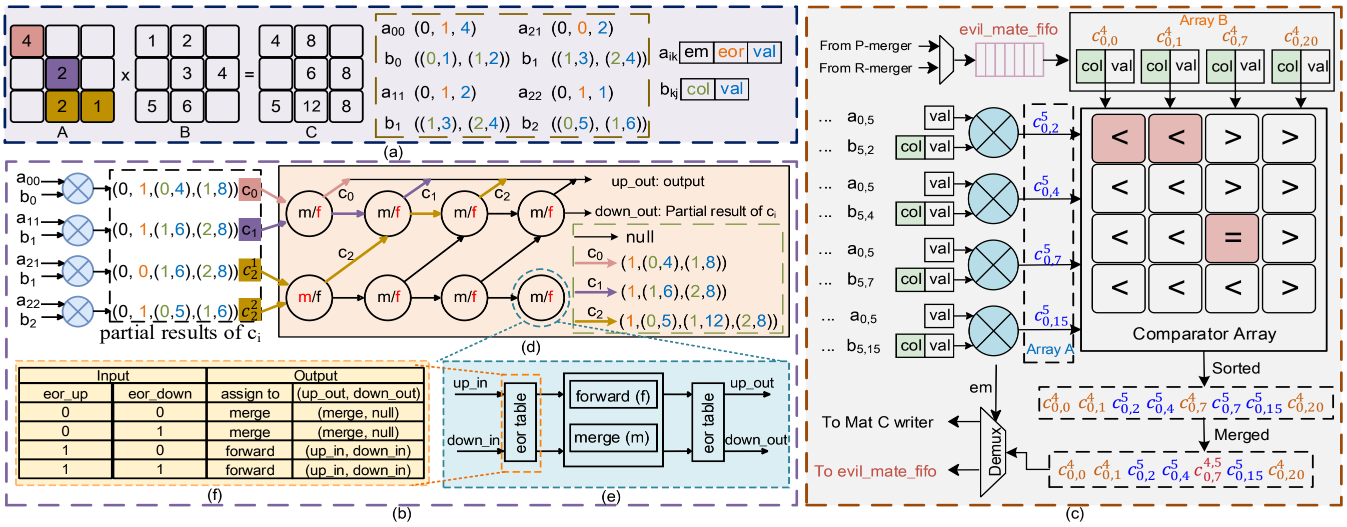
row reuse of matrix B. As depicted in Fig. 3(a), we retain the original memory allocation strategy of C^{2}SR, assigning each row to a unique memory bank to resolve memory conflicts. However, we modify the access pattern from row-major to dual-row column-major. Specifically, as shown in Fig. 3(a), two rows within the same bank are accessed in column-major order, while rows in different banks are accessed in row-major order. For storing non-zero elements, the optimized C^{2}SR format continues to use the original format, consisting of [$row\ length$, $row\ pointer$, val, col], where the i_{th} entry in the [$row\ length$, $row\ pointer$] array stores the number of non-zeros in the i_{th} row and the pointer to the first non-zero element in the [val, col] array, as illustrated in Fig. 3(b).

merger-specific data format. During the merging process, we use a merger-specific data format to facilitate the flexible interconnects of the R-merger. As depicted in Fig. 3(c), this format consists of [em, eor, val, col] arrays. The evil-mate (em) indicates whether a row requires merging with an evil row in the P-merger. As shown in Fig. 3(c), when row b_3 corresponding to a_{03} is an evil row, the partial results c_0^1 from a_{00} in the R-merger need to merge with c_0^3, generated by a_{03} and b_3 in the P-merger. Consequently, a_{00} and a_{02} are called evil-mates (em=1). The end-of-row (eor) signifies whether a non-zero element is at the end of a row. In Fig. 3(c), the last non-zero element in a row has eor=1; otherwise, eor=0. The col and val arrays represent the column coordinate and value of a non-zero element, respectively.

IV. Hardware Design

A. Overview

Fig. 4 shows an overview of the AiSpGEMM architecture. We established N PE Groups (PEG) based on the number of memory channels in the external memory, with each channel



Input		Output	
eor_up	eor_down	assign to	(up_out, down_out)
0	0	merge	(merge, null)
0	1	merge	(merge, null)
1	0	forward	(up_in, down_in)
1	1	forward	(up_in, down_in)

Fig. 6. The Architecture of R-merger and P-merger and Sparse matrix format used in both mergers. (a) Example of merger-specific data format; (b) R-merger's architecture; (c) P-merger's architecture; (d) Details of merge tree; (e) Details of the m/f module; (f) Details of the *eor* table.

being 512 bits, to fully exploit the maximum bandwidth of HBM. Each PEG utilizes an AXI bus interface to transfer data from external memory to the on-chip memory in the matrix loaders. Next, the Matrix *A* (Mat *A*) and Matrix *B* (Mat *B*) loaders send a_{ij} and b_j to the scheduler & converter.

Specifically, as depicted in Fig. 5(a), the Mat *A* loader initially retrieves the $[val,\ col]$ of a_{ij} using the $[row\ length,\ rowpointer]$ from the optimized C^2SR. Next, as illustrated in Fig. 5(b), the Mat *B* loader uses the column coordinates of a_{ij} provided by the Mat *A* loader to locate the address of b_j in the address mapping table of Mat *B*, thereby obtaining the val and col of each element in b_j. In the scheduler, we classify each a_{ij} and b_j as either evil rows or non-evil rows based on the evil row markers (highlighted in red in Fig. 3(a)) and convert the optimized C^2SR format into the sparse data format used for merging in the optimized C^2SR decoder, as shown in Fig. 5(c). Subsequently, the scheduler & converter allocates the evil rows and non-evil rows to the P-merger and R-merger, respectively, as illustrated in Fig. 5. Due to the presence of the em (evil-mate), the R-merger needs to use a demux to forward the merged rows with the em markers to the P-merger to complete the final merging. Finally, the final merged results (*i.e.*, each c_i row) produced by the P-merger and R-merger are written back to the external memory by the Matrix *C* (Mat *C*) writer.

B. R-merger

The R-merger facilitates the merging and rapid output of partial results from multiple sets of c_i through flexible interconnects, thereby improving PE utilization and computational efficiency. Fig. 6(b) illustrates the R-merger architecture, which includes four multipliers and a merge tree. The R-merger first receives multiple (a_{ij}, b_j) pairs and performs the multiplication operations. The resulting partial results of c_i are then fed into the merge tree, fully utilizing the pipeline, even with multiple sets (Multi-sets) of c_i partial results.

Multi-sets c_i's partial results merging. To merge the partial results of multiple sets of c_i, the merger tree is automatically divided into multiple sub-merger trees, utilizing flexible m/f modules. Specifically, the merger tree comprises several independent m/f modules, as shown in Fig. 6(d). As illustrated in Fig. 6(e), the m/f module can dynamically respond (merge or forward) and output based on the *eor* table (as illustrated in Fig. 6(f)). Subsequently, multiple m/f modules will dynamically form the sub-merger tree for the corresponding c_i's partial results based on the output direction.

Rapid output of merged results. In current merger tree structures, the final result is typically output at the last node. However, R-merger is different as it employs flexible interconnects to achieve multi-set c_i's partial results merging, producing multiple outputs. In a traditional tree-based architecture, the R-merger would have to queue outputs at the final node, leading to prolonged data stalling and significant synchronization overhead in subsequent computations. To address this, we designed an upward-output merger tree structure with dual output ports (up_out, $down\_out$) at each m/f node. During the merging process, each m/f node sends the merged result to the upper layer through up_out. When both eor_up and eor_down signals of the first layer's m/f nodes are 1, the final result is output through the up_out port. If the $down\_out$ of the last m/f module in the first layer is not null, it indicates that c_i is still a partial result and is fed back to the initial input port for further merging.

Example of R-merger's dataflow. As shown in Fig. 6(b), multiple partial results of c_i enter the merger tree, where their $eors$ indicate that c_0 and c_1 are from different rows. Since c_0 is the end-of-row element (eor=1), it is immediately output, while c_1 is passed to the next m/f for output. Meanwhile, c_2^0 and c_2^1, which are on the same row, need to be merged into c_2 (with their $eors$ combining as (0,1)). After merging, c_2, being the end-of-row element, is passed to the top m/f. The subsequent m/f then sequentially outputs c_1 and c_2 according to the eor table.

C. P-merger

Fig. 6(c) and Algorithm 1 illustrate the overall architecture of the P-merger and provide details of the intra-row parallel merging algorithm. Fig. 6(c) shows that the P-merger includes four multipliers, an evil_mate_fifo, and a 4x4 comparator array. The P-merger primarily uses the comparator array for parallel sorting and merging. Specifically, $Array\ a$ consists of elements from the c_i rows obtained by multiplying a_{ij} with each element in b_j. $Array\ b$ is composed of c_i rows with em markers stored in the evil_mate_fifo. The data in the evil_mate_fifo is supplied by the R-merger and the P-merger (with em markers).

Intra-row parallel merging. The comparator array indicates the positions of $Array\ b$'s elements in relation to $Array\ a$ by determining the boundary tile. First, the comparator array compares the col of $Array\ a$ and $Array\ b$. The comparison between $Array\ a$ and $Array\ b$ is denoted by '$<$', '$=$', '$>$'. Subsequently, we identify the boundaries demarcated by the '$<$', '$>$' and '$=$'. In each column in the comparator array, the topmost '$<$' and '$=$' tile is the boundary tile (colored pink). For example, for the left-top tile, where the col of $Array\ b$ is less than of $Array\ a$ ($0 < 2$), indicating that the $Array\ b$ element ($c^4_{0,0}$) should come before the $Array\ a$ element ($c^5_{0,2}$). Additionally, since the column indices of subsequent elements in $Array\ a$ and $Array\ b$ are unpredictable, the last element of one of the arrays will be sent back to the original array for further computation according to the following cases:

(1) If no boundary tile exists in the column, the last element in $Array\ b$ (*e.g.*, $c^4_{0,20}$) is sent back to the original array for subsequent computation.

(2) If the boundary tile is at the end of the column, the last element in $Array\ a$ is sent back to the original array for subsequent computation.

Subsequently, columns that have bounded tiles and do not satisfy the above two cases are sorted and merged into the output phase. Finally, it's necessary to check if the merged data is classified as em to decide whether to send this merged data to Mat C writer or P-merger.

V. Evaluation

A. Evaluation setup

We implemented AiSpGEMM in Verilog HDL and deployed it on the AMD-Xilinx Alveo U280 FPGA, measuring both execution time and power consumption. We compared the performance of our design against the SOTA FPGA-based SpGEMM accelerator (HLS-based) [6] and the NVIDIA cuSPARSE library on an NVIDIA A6000 GPU [9]. Our evaluation used 15 matrix sets from the widely used SuiteSparse [3], including 10 balanced and 5 imbalanced matrices. The balanced matrices were selected to match those used in the SOTA FPGA-based SpGEMM accelerator, while the imbalanced matrices were sourced from HiSpMV [12], which is dedicated to effectively accelerating imbalanced SpMV (sparse matrix-dense vector) operations. As in other work, we evaluated the performance of SpGEMM by multiplying the sparse matrix with itself [13].

Algorithm 1 Intra-row parallel merging algorithm

```
Input: array_a, array_b
Output: merged_output
  ▷ Compare and load data into comparator array
1: for $i = 0 \to 3$ do
2:     for $j = 0 \to 3$ do
3:         Compare(array_a[i], array_b[j])
4:     end for
5: end for
  ▷ Identify boundary tiles and sort columns
6: for col in comparator_array.columns do
       ▷ find boundary tile
7:     for tile in col do
8:         if tile == '$<$' or tile == '$=$' then
9:             boundary_tile = tile and break
10:        end if
11:    end for
       ▷ sort columns by boundary tiles
12:    if boundary_tile then
13:        for tile in col do
14:            if tile <= boundary_tile then
15:                sorted_column.append(tile)
16:            end if
17:        end for
           ▷ case (2)
18:        if boundary_tile == col[-1] then
19:            array_a.write(array_a[col.index])
20:        end if
21:        merged_output.extend(sorted_column)
       ▷ case (1)
22:    else
23:        array_b.write(array_b[col.index])
24:    end if
25: end for
```

TABLE I
Resource Utilisation and Frequency Comparison.

	LUT	LUTRAM	FF	BRAM	URAM	DSP	Frequency
Li et al. [6]	169K	N/A	217K	185	60	54	100MHz
AiSpGEMM-R-4	71K	4K	74K	271	N/A	80	225MHz
AiSpGEMM-21	362K	13K	384K	1596	N/A	756	225MHz

B. Comparison with FPGA-based SpGEMM accelerator

To ensure a fair comparison, we evaluated our design against the SOTA FPGA-based SpGEMM accelerator using the same benchmarks (balanced matrices) and matching off-chip memory bandwidth and computational resources. Therefore, we used the AiSpGEMM-R-4 baseline for comparison, where AiSpMM-R-4's PEG employs only the R-merger (utilizing similar computational resources as the prior SpGEMM accelerator) and normalized the off-chip memory bandwidth to the same 4 memory channels. Additionally, similar to the prior SpGEMM accelerator, our design employs single-precision floating-point computation.

Resource consumption and frequency. We synthesized our design and performed place-and-route using Vivado 2020.2,

TABLE II
PERFORMANCE COMPARISON ON BALANCED MATRICES.

Balanced Matrices	Size	Nonzero	Density	Latency (ms)		Speedup
				Li et al. [6]	AiSpGEMM-R-4	
poisson3Da	13514	352762	0.19%	198.79	70.44	2.82×
raefsky1	3242	294276	2.80%	274.59	76.96	3.57×
crystk01	4875	315891	1.33%	218.85	22.59	9.69×
s3rmt3m3	5357	207695	0.72%	86.29	9.34	9.24×
t2dah_a	11445	176117	0.13%	55.82	5.89	9.47×
nasa2910	2910	174296	2.06%	127.97	13.42	9.53×
bcsstk24	3562	159910	1.26%	75.88	7.83	9.69×
cavity26	4562	138187	0.66%	66.33	22.91	2.89×
ex9	3363	99471	0.88%	40.11	4.82	8.33×
af23560	23560	484256	0.09%	154.09	59.59	2.59×

TABLE III
IMBALANCED MATRICES (SIZE: ROWS = COLUMNS).

Imbalanced Matrices	Size	Nonzero	Density	Gini coefficient
c-52	23,948	202,708	3.53E-04	0.538
language	399,130	1,216,334	7.64E-06	0.310
analytics	303,813	2,006,126	2.17E-05	0.646
poli_large	15,575	33,033	1.36E-04	0.506
hangGlider_3	10,260	92,703	8.81E-04	0.248

obtaining hardware resource utilization and frequency data from the place-and-route report. Table I presents the resource utilization and design frequency for both the SOTA FPGA-based SpGEMM accelerator and AiSpGEMM. Although AiSpMM-R-4 uses 30 more DSP resources than the prior SpGEMM accelerator, this increase occurs during the merging phase, while DSP usage remains the same during the multiplication phase. To fully utilize the maximum bandwidth of HBM at 450MHz with 512-bit ports, our core needs to achieve only a frequency of 225MHz [7]. Additionally, the prior SpGEMM accelerator is written in C++ and converted to Verilog using Vitis HLS. This conversion may introduce timing constraints, resulting in a lower design frequency of only 100MHz.

Performance comparison. Table II presents the latency for balanced matrices on both the SOTA FPGA-based SpGEMM accelerator and AiSpGEMM-R-4. It is observed that AiSpGEMM-R-4, under the same configuration as the prior SpGEMM accelerator (*e.g.*, off-chip memory bandwidth, DSP usage), is geometrically mean 5.8× faster. This improvement is attributed to the flexible interconnects of the R-merger, which enable parallel merging of multi-sets c_i's partial results and rapid output of the final results. Additionally, since the prior SpGEMM accelerator uses Vitis HLS to convert C++ to Verilog, it lacks fine-grained architectural optimizations. Since the original paper on the prior SpGEMM accelerator does not provide details on energy efficiency, we focus on discussing AiSpGEMM's energy efficiency (GFLOPS/W) compared to GPU.

C. Comparison with NVIDIA A6000 GPU

We use the NVIDIA cuSPARSE library running on an NVIDIA A6000, as a baseline for comparison. The formula we use to calculate GFLOPS is Eq. (1) [12], which directly reflects the execution time, as $Nonzero$ and ROW are derived from the matrix.

$$\text{GFLOPS} = \frac{2 \times (\text{Nonzero} + \text{ROW})}{\text{Latency}} \tag{1}$$

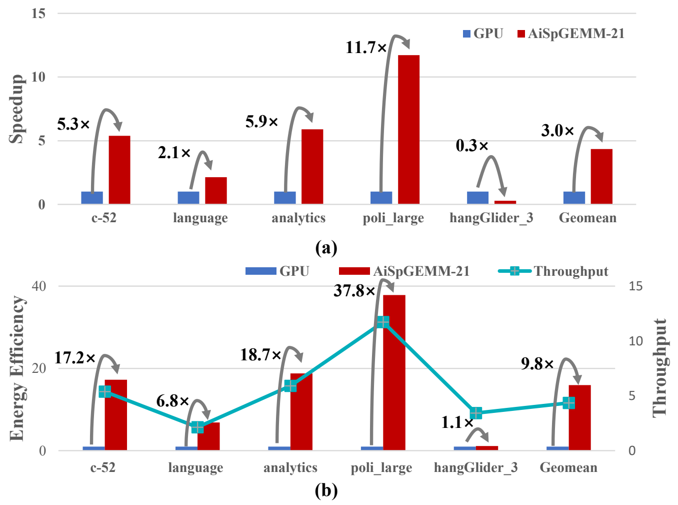


Fig. 7. Speedup and energy efficiency comparison. (a) Speedup of AiSpGEMM-21 over GPU; (b) Energy efficiency and Throughput of AiSpGEMM-21 over GPU.

Comparison Results. The NVIDIA A6000 GPU offers a maximum memory bandwidth of 768 GB/s, while the Xilinx U280 provides a maximum memory bandwidth of 460 GB/s. We also used imbalanced matrices for comparison. During kernel execution, we measured power consumption using vendor tools ($nvml$ for GPUs and Xilinx Power Estimator (xpe) for FPGAs). In terms of geomean speedup and energy efficiency improvement, AiSpGEMM achieves a 3.0× speedup and a 9.8× energy efficiency improvement over GPU, as illustrated in Fig. 7. Additionally, for matrices with smaller Gini coefficients, indicating a more balanced data distribution (*e.g.*, *hangGlider_3* in Table III), AiSpGEMM shows lower speedup. This is because, in matrices with higher Gini coefficients, the number of floating-point operations required by the hardware is reduced due to a decrease in the partial results of c_i. As a result, the GPU has lower core utilization and higher performance overhead. Additionally, AiSpGEMM-21 achieves a 4x increase in average throughput compared to the same GPU. This is because the throughput formula (Eq. (1)) used directly reflects execution time, with shorter execution times naturally resulting in higher throughput.

VI. CONCLUSION

In this work, we addressed the challenges of accelerating imbalanced SpGEMM using AiSpGEMM. First, we introduced the optimized C^2SR format to resolve memory bank conflicts and reuse matrix B. Second, we developed the R-merger to balance PE utilization and parallelism. Additionally, we designed an intra-row parallel merging algorithm and the corresponding hardware, the P-merger, to accelerate the merging of evil rows. AiSpGEMM has demonstrated significant performance gains across multiple benchmarks. Our extensive experiments demonstrated AiSpGEMM's superior performance and energy efficiency compared to the SOTA FPGA-based SpGEMM accelerator and the NVIDIA cuSPARSE library running on NVIDIA A6000 GPU.

References

[1] A. Azad, A. Buluç and J. Gilbert, ”Parallel Triangle Counting and Enumeration Using Matrix Algebra,” in *2015 IEEE IPDPSW*, Hyderabad, India, 2015, pp. 804–811.

[2] R. Bulirsch, E. Nerz, H.J. Pesch, and O. von Stryk, ”Combining Direct and Indirect Methods in Optimal Control: Range Maximization of a Hang Glider,” in *ISNM*, vol. 111, R. Bulirsch, A. Miele, J. Stoer, and K. Well, Eds., Birkhäuser Basel, 1993.

[3] T.A. Davis and Y. Hu, ”The University of Florida Sparse Matrix Collection,” *ACM TOMS*, vol. 38, no. 1, Article 1, pp. 1–25, Nov. 2011.

[4] J.R. Gilbert, S. Reinhardt, and V.B. Shah, ”A Unified Framework for Numerical and Combinatorial Computing,” *Comput. Sci. Eng.*, vol. 10, no. 2, pp. 20–25, Mar.–Apr. 2008.

[5] F.G. Gustavson, ”Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition,” *ACM TOMS*, vol. 4, no. 3, pp. 250–269, Sep. 1978.

[6] S. Li and W. Liu, ”Accelerating Gustavson-based SpMM on Embedded FPGAs with Element-wise Parallelism and Access Pattern-aware Caches,” in *2023 IEEE DATE*, Antwerp, Belgium, 2023, pp. 1–6.

[7] A. Lu, Z. Fang, W. Liu, and L. Shannon, ”Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking,” in *2021 ACM/SIGDA FPGA*, New York, NY, USA, 2021, pp. 105–115.

[8] X. Luo, M. Zhou, S. Li, Z. You, Y. Xia, and Q. Zhu, ”A Nonnegative Latent Factor Model for Large-Scale Sparse Matrices in Recommender Systems via Alternating Direction Method,” *IEEE TNNLS*, vol. 27, no. 3, pp. 579–592, Mar. 2016.

[9] Nvidia, ”Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs,” 2023. [Online]. Available: https://docs.nvidia.com/cuda-libraries/index.html.

[10] S. Pal et al., ”OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator,” in *2018 IEEE HPCA*, Vienna, Austria, 2018, pp. 724–736.

[11] E. Qin et al., ”SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training,” in *2020 IEEE HPCA*, San Diego, CA, USA, 2020, pp. 58–70.

[12] M.B. Rajashekar, X. Tian, and Z. Fang, ”HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs,” in *2024 ACM/SIGDA FPGA*, New York, NY, USA, 2024, pp. 154–164.

[13] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, ”MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product,” in *2020 IEEE/ACM MICRO*, Athens, Greece, 2020, pp. 766–780.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, ”Learning Structured Sparsity in Deep Neural Networks,” in *NIPS ’16*, Red Hook, NY, USA, 2016, pp. 2082–2090.

[15] G. Zhang, N. Attaluri, J.S. Emer, and D. Sanchez, ”Gamma: Leveraging Gustavson’s Algorithm to Accelerate Sparse Matrix Multiplication,” in *ASPLOS ’21*, New York, NY, USA, 2021, pp. 687–701.

[16] Z. Zhang, H. Wang, S. Han, and W.J. Dally, ”SpArch: Efficient Architecture for Sparse Matrix Multiplication,” in *2020 IEEE HPCA*, San Diego, CA, USA, 2020, pp. 261–274.

[17] Q. Wu, L. Zhao, Y. Gui, H. Liang, X. Wang, and X. Jin, ”Efficient Message Passing Architecture for GCN Training on HBM-based FPGAs with Orthogonal Topology On-Chip Networks,” in *2024 ACM/SIGDA FPGA*, New York, NY, USA, 2024, pp. 187.

[18] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi and C. Hao, ”FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference,” in *2023 IEEE HPCA*, Montreal, QC, Canada, 2023, pp. 1099-1112.

[19] Y. Liu, R. Chen, S. Li, J. Yang, S. Li, and B. da Silva, ”FPGA-Based Sparse Matrix Multiplication Accelerators: From State-of-the-Art to Future Opportunities,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 4, Art. no. 59, Dec. 2024, pp. 1-37.

[20] R. Chen, H. Zhang, S. Li, E. Tang, J. Yu, and K. Wang, ”Graph-OPU: A Highly Integrated FPGA-Based Overlay Processor for Graph Neural Networks,” in *2023 IEEE FPL*, Gothenburg, Sweden, 2023, pp. 228–234.