2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 979-8-3503-4963-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASAP61560.2024.00034

Sparm: A Sparse Matrix Multiplication Accelerator Supporting Multiple Dataflows

Shengbai Luo, Bo Wang, Yihao Shi, Xueyi Zhang, Qingshan Xue, and Sheng Ma

National University of Defense Technology, Changsha, China
{luoshengbai, bowang, shiyihao, z, xueqingshan23, masheng}@nudt.edu.cn

***Abstract*—As the main workload of many scientific and machine learning applications, sparse matrix-matrix multiplication (spGEMM) has become a hot research field. The current spGEMM workloads exhibit sparsity and irregularity, leading to computational inefficiencies on traditional hardware platforms and motivating numerous customized accelerators. These specialized hardware designs typically accelerate only one type of spGEMM dataflow (such as Inner-Product, Outer-Product, or Gustavson), yet the computational efficiency of the same spGEMM kernel can vary significantly under different dataflows. Flexagon is the first spGEMM accelerator to support multiple dataflows, but its MRN (Merger-Reduction Network) design causes a lot of data blocking and load imbalance, which limits its performance.**

In this work, we propose Sparm, which achieves efficient merging of psums (partial sums) for different dataflows through a specialized indexing unit. Sparm addresses the performance bottlenecks encountered by Flexagon when facing highly sparse matrices. Furthermore, Sparm employs a row/column prefetcher to load the streaming matrices proactively and thus significantly reduces the amount of DRAM access. We conduct simulations using a cycle-accurate simulator on workloads from various application domains, and the results demonstrate that Sparm achieves average performance gains of 2.62× and 1.35× compared to state-of-the-art spGEMM accelerators SIGMA and Flexagon. Meanwhile, Sparm brings only a small amount of additional hardware overhead over Flexagon.

***Index Terms*—Sparse Matrix Multiplication, Domain-Specific accelerator, Dataflow, Data Reuse**

I. Introduction

Sparsity has become increasingly important in modern science and machine learning applications. As a crucial computational kernel in numerous algorithms, such as object detection and natural language processing, sparse matrix multiplication (spGEMM) stands as a focal point of customized accelerators. However, in many real-world scenarios, the structure of sparse matrices is highly irregular, especially with the emergence of large-scale graph analytics and various model compression techniques in recent years [17]. For instance, Many CNNs employ the ReLU activation function to convert negative values to zeros [8] [19]. Pruning algorithms also introduce substantial sparsity, where in some cases, over 60% of convolutional layer weights and over 90% of fully connected layer weights can be removed [3]. Real-world graph datasets often feature enormous dimensions with extremely high sparsity, sometimes reaching dimensions as large as 10^{13} while having less than 0.0001% of non-zero values [9]. These realistic workloads exhibit diverse sparsity patterns, leading to highly irregular memory accesses. Therefore, tremendous challenges are posed in efficiently handling sparse matrix multiplication.

There are three existing spGEMM dataflows, including Inner-Product, Outer-Product, and Gustavson. These three dataflows make different trade-offs between input reuse and output reuse, and thus none of them can be suitable for all types of workloads. Previous works have proposed numerous spGEMM accelerators that often optimize for input or output data reuse based on a fixed dataflow. These accelerators tend to perform well only within specific sparsity ranges or under particular sparsity patterns because their fixed dataflow strategies struggle to adapt to the varying memory access characteristics of different workloads [7].

Flexagon [10] is the first accelerator to support these three dataflows. It exposes the fact that the optimal dataflow not only can change between DNN models, but also from layer to layer within a particular DNN model. Flexagon hinges on a novel network topology (called MRN) and a new L1 on-chip memory organization to effectively capture the memory access patterns that each dataflow exhibits for input, output, and psums. However, while MRN has a high hardware utilization rate, it comes at the cost of blocking a significant amount of data.

Therefore, we propose Sparm, a highly efficient spGEMM accelerator that can match the optimal dataflow for workloads with varying dimensions and sparsity patterns. In this work, we first conduct a detailed analysis of the characteristics of three existing dataflow schemes. We identify the limitations of Flexagon, which bottleneck its performance when performing merge operations. To address these limitations, we design a specialized control unit named Merge Manager and an efficient reduction tree that enables regular accumulation of psums for different dataflows. We retain Flexagon's three on-chip memory structures, including a read-only FIFO customized for the stationary matrix, a cache to store the streaming matrix, and a PSRAM for storing the large number of psums generated in Outer-Product and Gustavson dataflows. Additionally, we introduce a memory structure called Bitindex SRAM, which loads index information for the intersection of the stationary matrix's rows and the streaming matrix's columns. The Merge Manager then utilizes this index information to control psums for regular merging. Furthermore, we observe that input matrices have different loading orders across various dataflows

2160-052X/24/$31.00 ©2024 IEEE
DOI 10.1109/ASAP61560.2024.00034

(row-wise, column-wise, or irregular), we employ tailored data prefetching strategies for each dataflow, implemented using the Look-Ahead FIFO and RC (row/col) Prefetcher.

In summary, the contributions of this paper are:

- We demonstrate that spGEMM operation exhibits different memory access characteristics under different dataflows and detailedly analyze the limitations of the Merging Reduction Tree in Flexagon.
- We design Sparm, which relies on a novel reduction network that supports regularized merging for spGEMM operations under three different dataflows.
- We introduce the RC Prefetcher, which significantly reduces DRAM accesses.
- We evaluate Sparm using an extensive dataset of real-world sparse matrices from SuiteSparse. Our observations indicate that, compared to SIGMA and Flexagon, Sparm achieves average performance gains of 2.62x and 1.35x, respectively.

II. Background

A. Compressed Storage Formats

SpGEMM operates compressed sparse data, with the most common formats like CSR (compressed sparse row), CSC (compressed sparse column), and bitmap. Fig. 1 illustrates sparse matrices encoded in these formats. CSR format consists of three components: rowptr, colindex, and value. Rowptr represents the compressed row indices. The difference between rowptr[i+1] and rowptr[i] indicates the number of non-zero elements in the i-th row of the matrix. Value stores the non-zero values in row-major order, and colindex indicates the corresponding column coordinates. In principle, CSC is similar to CSR, but it compresses the matrix by columns. Bitmap compression format uses a bitmap consisting of 0s and 1s to index the positions of non-zero values, so it is hardware-friendly.

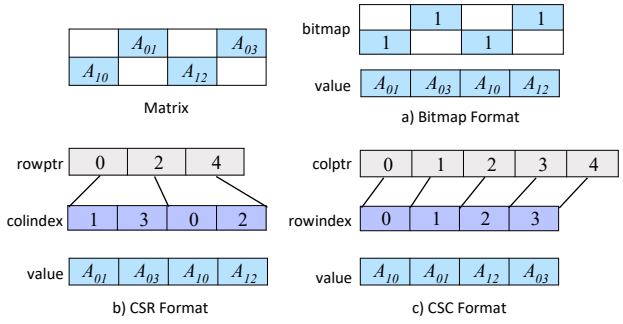

Fig. 1. Compressed sparse matrix formats.

B. SpGEMM Dataflows

Fig. 2 illustrates three major SpGEMM dataflow schemes: Inner-Product (IP), Outer-Product (OP), and Gustavson (Gust). From an algorithmic perspective, these dataflows are merely variations of a triple nested loop. This triple loop iterates over the dimension M of matrix A, the dimension N of matrix B, and their common dimension K. The dataflow is determined by the position of the common iteration K. When K appears in the innermost loop, outermost loop, or middle loop, it corresponds to the IP, OP, and Gustavson dataflows, respectively.

Inner-product is an output-stationary dataflow, a well-known matrix multiplication pattern. It computes an element of the output matrix by performing an inner product between a row of matrix A and a column of matrix B. This computation pattern achieves good output reuse but poor input reuse. Inner-product must traverse all elements of the rows and columns, even though there are few effectual intersections. Therefore, it is efficient for highly dense matrices [18], but the performance decline caused by its inefficient intersections will be gradually amplified as the sparsity of the input matrices increases. all elements of the rows and columns must be traversed, even though there are few effectual intersections.

In contrast, the Outer-Product dataflow computes a partial sum matrix of size $M \times N$ by iterating over a column of matrix A and a row of matrix B. Each partial sum matrix corresponds to a k-iteration in range [0:K], and the final output matrix is generated by merging the K partial sum matrices. OP achieves good input reuse as the columns of matrix A and corresponding rows of matrix B are traversed once in sequence and will not be reused in the future. This pattern is effective for highly sparse matrices. However, for dense or large-scale matrices, its performance degrades significantly due to the significant memory traffic brought by the K large partial sum matrices.

Gustavson, as a compromise between the first two dataflows, computes a row of the output matrix by operating a row of matrix A and the corresponding rows of matrix B. This dataflow has the advantage of generating fewer partial sums compared to OP, making it easier to store them on-chip. Additionally, computations for different rows of matrix C are isolated, allowing for parallelization to improve computational efficiency. However, the reuse of rows of matrix B is poor as they are repeatedly and irregularly fetched. For example, in Fig. 2c), the first row of matrix B will be accessed twice.

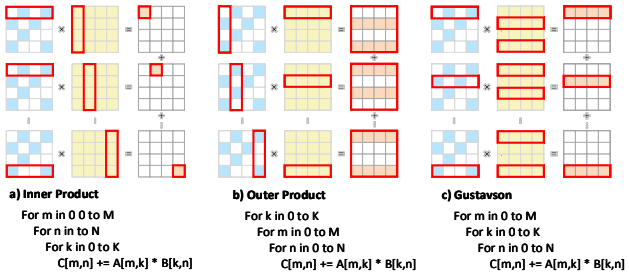

Fig. 2. Comparison of three spGEMM dataflows.

C. SpGEMM Accelerators

Many spGEMM accelerators have been previously proposed for the aforementioned three dataflows. UCNN [5]

and SIGMA [14] have implemented spGEMM based on the IP dataflow. To alleviate the inherent problem of inefficient intersections in the IP dataflow, UCNN introduced dot product decomposition and weight repetition to reduce on-chip memory reads [5]. SIGMA adopts a hardware-friendly bitmap to accelerate intersections [14]. However, these approaches do not address the poor input reuse inherent in the IP dataflow.

Accelerators like OuterSpace [13] and Sparch [21] have implemented the OP dataflow. To mitigate the substantial storage and data movement overhead associated with numerous partial sum matrices, these accelerators have made several innovations. OuterSpace partitions the partial sum matrices by rows and performs merging separately between rows, which reduces the complexity of merging [13]. Sparch utilizes a comparator array to highly parallelize merging and employs a special matrix compression method to reduce the number of partial sum matrices by three orders of magnitude [21]. Sparch performs impressively when handling highly sparse matrices. Nevertheless, when dealing with less sparse or large-scale matrices, there are still significant memory access overheads due to the need to send a large number of partial matrices off-chip.

Gamma [20] and MatRaptor [16] have leveraged the Gustavson dataflow. The hardware in these accelerators emphasizes accelerating memory access to minimize memory traffic and facilitate the merging of partial output rows. Gamma utilizes the fact that rows of the output matrix can be generated independently from each other. It uses specialized PEs with high-radix mergers to parallelize many independent merges, which achieves high throughput with linear cost [20]. Additionally, it enhances the reuse of matrix B through an on-chip storage structure called FiberCache. MatRaptor introduces a new storage structure called C^2SR for sparse matrices, which minimizes memory channel conflicts and improves the parallelism of memory reads [16]. It also designs specialized sorting hardware to enhance merge performance. However, ultimately, these designs can only alleviate the problem of poor input reuse for matrix B to a certain extent. If the size of matrix B is too large and the on-chip cache cannot fully accommodate the frequently accessed rows of matrix B, the performance will deteriorate sharply.

Spada [9] and Flexagon [10] are among the first accelerators to flexibly support all three dataflows. Spada proposes an adaptive window-based dataflow that can flexibly adapt to different sparsity patterns, optimally matching data distributions and achieving various reuse benefits [9]. Flexagon introduces a novel inter-layer dataflow mechanism that matches the optimal dataflow for different DNN layers without requiring explicit hardware modules for compression format conversion. Flexagon features a new reduction network (MRN) to support the reduction of dot products and the merging of partial sum matrices. Furthermore, Flexagon includes a novel L1 on-chip memory organization to effectively capture the memory access patterns of inputs, outputs, and partial sums exhibited by each dataflow [10].

However, the MRN's merging mechanism based on coordinate matching has certain drawbacks. Fig. 3b) illustrates how the MRN works under the Gustavson dataflow, using the spGEMM example from Fig. 3a). Ideally, partial sums $*C_{00}$ from the first multiplier and the second multiplier have matched coordinates and are merged. However, when partial sums $*C_{11}$ and $*C_{10}$ from the third and fourth multipliers have mismatched coordinates, the one with the smaller column coordinate, $*C_{10}$, is sent to the parent node while $*C_{11}$ remains stalled. It is worth noting that this can trigger a chain reaction: blocking subsequent partial sums like $*C_{12}$ resulting from the multiplication of A_{10} and B_{02}. We refer to this coordinate mismatch and subsequent stalling as "blockage". In our experiments on a spGEMM example of size 256×256×256, as shown in Fig. 3c), the proportion of blockages increases significantly with the sparsity of the streaming matrix (matrix B), strongly impacting accelerator performance.

Sparm leverages the efficient indexing intersection of bitmaps, pre-computes the bitindex of partial sums, and regularizes the accumulation of partial sums. This approach avoids frequent blockages during the merging phase. Additionally, regularized accumulation eliminates the need for psums to carry additional coordinate information during merging, significantly reducing on-chip traffic (as Flexagon uses 16 bits to store values and 16 bits to store coordinates by default).

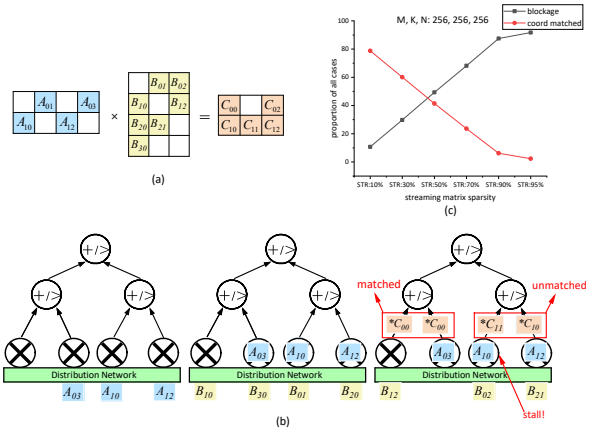

Fig. 3. a)SpGEMM example. b)Example of Flexagon running spGEMM. c)Proportion of blockage and coordinate matched in all cases.

III. Sparm Architecture

A. Architecture Overview

Fig. 4 illustrates the architecture of Sparm. We retain the three customized memory structures of Flexagon: StaFIFO, StrCache, and PSRAM, as they demonstrate excellent adaptability. They store the stationary matrix, the streaming matrix, and the partial sums respectively. When the StaFIFO loads the stationary matrix, the Bitindex SRAM simultaneously loads the corresponding bitindex table. Additionally, we incorporate a small capacity FIFO for each multiplier, along with a control unit called Merge Manager that orchestrates the orderly reading of these FIFOs. These on-chip components

are interconnected through a generic three-tier reconfigurable Network-on-Chip (NoC) composed of a distribution network, a multiplier network, and a reduction network.

In this work, our primary focus is on the accelerator's computational approach for three distinct dataflows. The determination of the optimal dataflow, (i.e. the analysis of matrix sparsity patterns and dimensions), will be reserved for future endeavors.

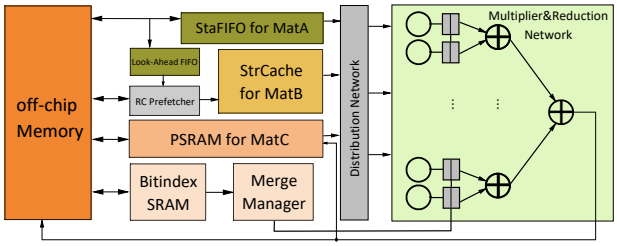

Fig. 4. Schematic of the Sparm architecture.

B. Regularized Merging

The main difference between Sparm's merging tree and MRN lies in the regularization of the merging process. The results of multiplication operations will first be sent to the FIFOs behind the multiplier. Then, based on the predefined dataflow, the Merge Manager will traverse the entries of the Bitindex table in a particular order and control the data in the corresponding FIFO pop. Multiplication operations are relatively intensive, and we only read some of the FIFOs at a time, so basically there will be no situation where the FIFO is empty and needs to wait. This merging strategy is highly regularized because we only perform merging operations for some k iterations per cycle, rather than sending all the psums directly to the merging tree and inefficiently comparing their coordinates as Flexagon does. In addition, we can remove the comparator and only need to introduce a small-scale FIFO array, which saves a lot of energy consumption. The examples in the next section will detail the process of regularized merging without comparing the coordinates of psums.

C. Walk-through Examples

We have prepared three illustrative examples to demonstrate how Sparm handles these three dataflows for the multiplication of matrices A and B from Fig. 3a). In these examples, we configure the accelerator with four multipliers and showcase how data flows through the multiplier network and reduction network. In these cases, values with "*" represent the psums that need to be reduced by the reduction network to produce the final output of matrix C. We assume that matrix A is the stationary matrix and matrix B is the streaming matrix.

1) Example of Inner-Product dataflow: Fig. 5 details the operational steps of Sparm under the IP dataflow. During the offline phase, the row bitmaps of matrix A are bitwise ANDed with the column bitmaps of matrix B. The intersections of their indices are recorded to generate the bitindex table. Concurrently, the optimal dataflow is determined through the analysis of the input matrices' bitmaps.

During the runtime computation phase, as many rows of matrix A as possible are allocated to the multipliers, given that matrix A is stored in CSR format in the IP dataflow (If a row has more non-zero elements than the number of multipliers, it will be allocated in multiple batches). Correspondingly, the associated bitindexes are sent to the Merge Manager for parsing. Based on the column coordinates of the data kept in the multipliers, each connected FIFO is assigned a K_ID (indicate which k iteration the data in the FIFO belongs to).

In step 2, matrix B stored in CSC format is multicast column-wise to the corresponding multipliers. The multipliers enclosed by the red box in Fig. 5 is a multipliers cluster. Each cluster corresponds to a row of matrix A and performs dot product operations. The resulting partial sums (psums) from the multiplication operations are then sent to the FIFOs, as shown in step 3. The Merge Manager will control the read of these FIFOs which have the same K_ID as in the table entry. In the IP dataflow, this step can be executed in parallel for multiple rows. Since two rows of matrix A are sent to the multiplier network in this example, the Merge Manager can simultaneously parse the bitindex table for two rows and control the independent reading operations of the two clusters of FIFOs.

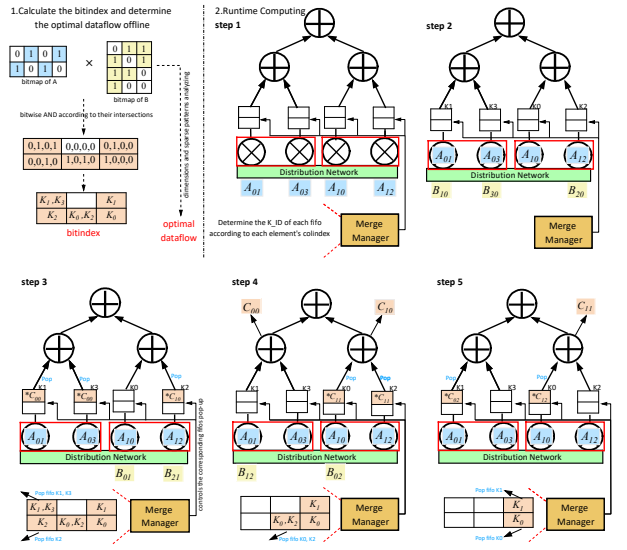

Fig. 5. Example of Sparm running SpGEMM using an Inner-Product dataflow.

2) Example of Outer-Product dataflow: Fig. 6 illustrates the detailed operational steps of Sparm when processing the OP dataflow. For brevity, we omit the offline stage operations as they remain consistent across different dataflows and have already been introduced in the previous section. In the OP dataflow, the matrix A is loaded in CSC format, meaning that the non-zero values of each column are loaded sequentially and mapped to multipliers. In our particular example, since

the first four columns each contain exactly one element, all four columns of the matrix A can be loaded to the multiplier network. The four columns of matrix A will be operated with the corresponding four rows of matrix B, so the non-zero values of these four rows will be sequentially allocated to the corresponding multipliers. It's worth noting that data can only be generated for a few k iterations or one k iteration at a time, and we cannot merge data belonging to the same k iteration. Flexagon's approach is to send all psums back to PSRAM first, and after all multiplication operations are completed, these psums are loaded from PSRAM for merging. Although this method is highly efficient for merging, it results in a significant amount of on-chip and even off-chip traffic. While for Sparm, when the accelerator generates psums for multiple k iterations at a time, it will first merge the psums that can be merged within these k iterations, and then perform a final merge after all k iterations are complete. This two-stage merging can better leverage PSRAM.

In the example shown in Fig. 6, the first step is to send the four columns of matrix A to the multiplier network and determine the corresponding K_ID for the four FIFOs. Then, the first element of each of the four rows in matrix B is sent to the corresponding multiplier. In step 3, the Merge Manager iterates through each entry of the bitindex table in row order and searches for values between k0 and k3. If found, it controls the corresponding FIFOs to be read. Note that there may be multiple FIFOs with the same K_ID, in which case only the leftmost non-empty one will be read. In this step, two *C_{00}s are popped and merged in step 4. Then, in step 5, the merged data C_{00} is sent to PSRAM. Note that we send the result to PSRAM instead of directly writing it to DRAM because in our particular example, all the psums belonging to k0 to k3 are calculated after a one-time matrix A distribution, and the controller cannot determine if there will be other k iterations like k4, k5, or more. Only when all k iterations are complete will the PSRAM perform the second stage of merging.

3) Example of Gustavson dataflow: Fig. 7 illustrates how Sparm works when the Gust dataflow is selected. Similar to the IP dataflow, the first step is to allocate as many rows of matrix A as possible to the multiplier network. However, in the Gust dataflow, both input matrices are in CSR format, so the required rows of matrix B need to be loaded into the StrCache. In our example, data A_{01} and A_{03} from the first row of matrix A need to be operated with the first and third rows of matrix B, while data A_{10} and A_{12} from the second row of matrix A need to be operated with the zeroth and second rows of matrix B. These four rows of matrix B will be preloaded into the StrCache in advance, which is achieved through the RC Prefetcher and will be introduced in detail in the next section.

Similar to the OP dataflow, each time a row of matrix B is loaded, the first element is always sent to the corresponding multiplier. However, the merging phase is similar to the IP dataflow where different multipliers clusters perform their merging operations independently, whereas there is no concept

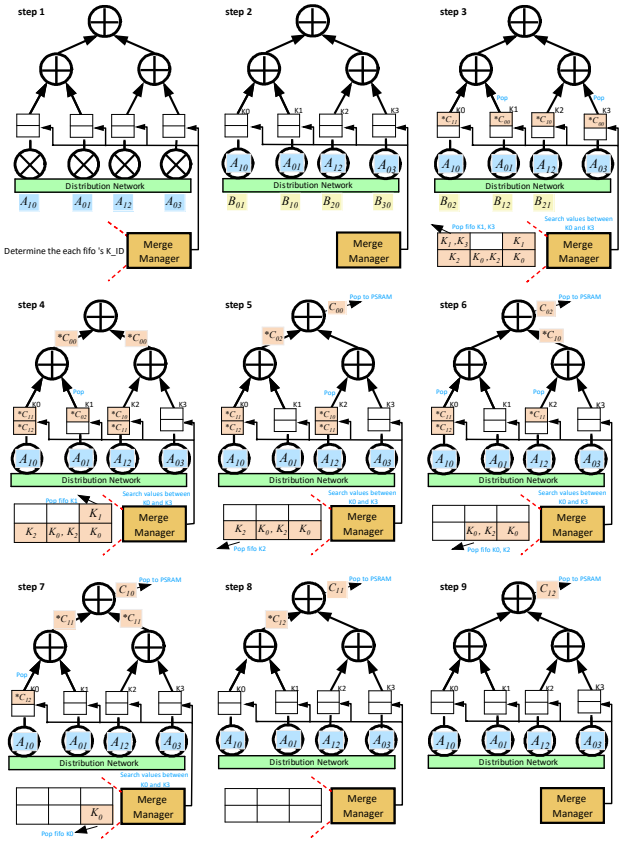

Fig. 6. Example of Sparm running SpGEMM using an Outer-Product dataflow.

of multipliers cluster in the OP dataflow. Each multipliers cluster generates a row of the output matrix. For example, from step 4 to step 6, the left multipliers cluster generates the first row of the output matrix: C_{00} and C_{02}, while the right multipliers cluster generates the second row of the output matrix: C_{10}, C_{11}, and C_{12}.

D. RC Prefetcher

Prefetching input matrices in spGEMM can significantly reduce DRAM accesses and improve performance. Many accelerators have previously implemented prefetching functions tailored to a single dataflow or their customized architectures. However, for different spGEMM dataflows, memory access patterns vary significantly, necessitating distinct prefetching strategies for each dataflow. In Sparm, since matrix A is loaded into StaFIFO in a row-by-row or column-by-column order, we primarily focus on prefetching matrix B. As observed from Fig. 2, the IP and OP dataflows load matrix B in a column-by-column and row-by-row order, respectively. Therefore, a simple prefetching strategy of sequentially prefetching matrix B's columns/rows suffices.

However, the same prefetching strategy cannot be applied to the Gustavson dataflow, as each row of matrix A requires

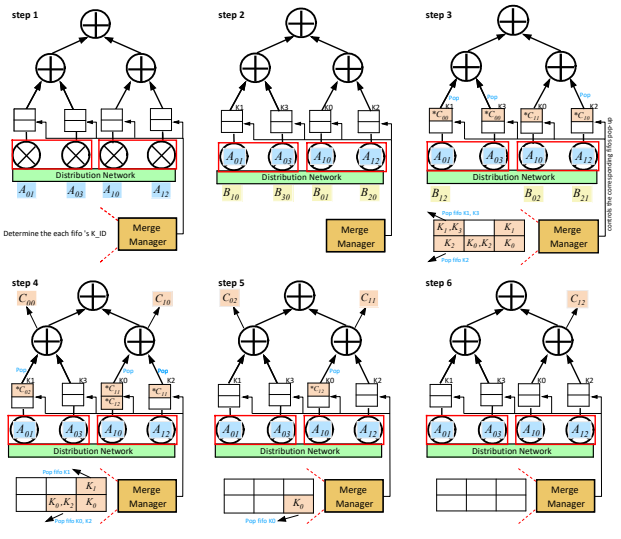

Fig. 7. Example of Sparm running SpGEMM using a Gustavson dataflow.

different rows of matrix B. For instance, assuming the accelerator is configured with only two multipliers when executing the Gust dataflow in Fig. 2c), the first iteration requires rows 1 and 3 of matrix B, while the second iteration requires rows 0 and 2. The required rows of matrix B vary across iterations. Nonetheless, by recording the column indices of matrix A data in subsequent iterations, we can predict which rows of matrix B will be needed in the future. Fig. 8 illustrates how Sparm implements prefetching under the Gustavson dataflow. As StaFIFO loads matrix A row by row, their corresponding column indices are loaded into the Look-Ahead FIFO in the same order. The RC Prefetcher then prefetches the corresponding rows of matrix B based on these column indices. In this way, the loading of matrix B is transparent for computation, and we can ignore the overhead of this process, which is hidden by computation latency.

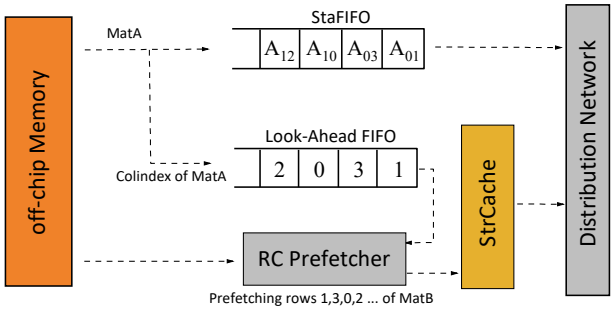

Fig. 8. Example of how RC Prefetcher works under Gustavson dataflow.

IV. Evaluation

A. Experimental Setup

1) Experimental Platform: To accurately evaluate Sparm, we have implemented a cycle-accurate microarchitecture simulator by modifying the STONNE framework [11]. We have compared the performance of SIGMA, Flexagon, and Sparm when running different types of spGEMM operations. All accelerators use the same number of multipliers and the same cache size (specifically 64 multipliers and a 1MB StrCache). For a fair comparison, Sparm employs the same hardware configuration parameters as Flexagon, namely 256 KB of PSRAM and 256 bytes of StaFIFO. Our Look-Ahead FIFO is also set to 256 bytes because of its synchronism with StaFIFO. The size of the Bitindex SRAM and the FIFO behind each multiplier is set to 128 KB and 64 bytes, respectively. Sparm is connected to a high-bandwidth memory with a bandwidth of 256GB/s and runs at a frequency of 800Mhz. We assume the cache access latency and DRAM access latency to be 1ns and 100ns, respectively. We implement our reduction network's RTL design and synthesize it using Synopsys DC on the TSMC 28 nm technology. We use CACTI 7.0 [1] to obtain the area and power results of the Bitindex SRAM. For the other components in Sparm that are identical to Flexagon, we adopt the results published in [10].

2) Workloads: To demonstrate the performance differences between Sparm and other state-of-the-art accelerators when matrices possess different sparse patterns and sparsity levels, we select some synthetic workloads. Firstly, we have evaluated SIGMA, Flexagon, and Sparm using randomly generated matrices with the same dimensions but varying sparsity levels, to demonstrate the performance gains brought by Sparm across different sparsity ranges. Secondly, we choose real-world tensors from the SuiteSparse Matrix Collection [2], as shown in Table I (set1-set6). we also use some common spGEMM workloads from typical transformer models, as indicated in Table I (set7-set9).

TABLE I
CHARACTERISTICS OF EVALUATED SPGEMM WORKLOADS.

	Workloads	*M*, *K*, *N*	Sparsity (MK, KN)
set1	ch5-5-b3*ch5-5-b2	600,600,200	99.33%,99.51%
set2	n3c6-b9*n3c6-b8	2511,4935,6435	99.80%,99.86
set3	n3c6-b4*n3c6-b3	3003,1365,455	99.63%,99.12%
set4	qc324	324,324,324	74.54%,74.54%
set5	bcsstk34	588,588,588	93.81%,93.81%
set6	mcfe	765,765,765	95.83%95.83%
set7	Transformer	256,512,64	80%90%
set8	DeiT-B	256,768,64	70%,80%
set9	DeiT-S	1024,384,64	60%,80%

B. Performance

To demonstrate the effectiveness of Sparm's merge strategy based on the Merge Manager, we have compared the runtimes of SIGMA's FAN (the reduction network in SIGMA),

Flexagon's MRN, and Sparm's merging tree when running ten spGEMM operations with input matrices of varying sparsity. The results are presented in Fig. 9 and Fig. 10. By default, we fix matrix A as the stationary matrix and matrix B as the streaming matrix.

Firstly, we observe that Sparm consistently exhibits a high speedup compared to SIGMA, a fixed IP dataflow accelerator. As the sparsity level of the streaming matrix increases, the performance gap between Sparm and SIGMA gradually widens. When the sparsity level of the streaming matrix reaches above 95%, the speedup of Sparm over SIGMA rapidly increases, reaching a maximum of 223x in our experiments. Sparm performs comparably to Flexagon within the sparsity range of 50%-95%. Similarly, when the sparsity level of the streaming matrix exceeds 95%, Sparm achieves significant performance improvements. In this experiment, Sparm's merging tree achieves an average speedup of 2.62x and 1.13x compared to SIGMA's FAN and Flexagon's MRN, respectively.

To further evaluate the effectiveness of the RC Prefetcher, we deploy our prefetching strategy on Flexagon and include it in the comparison. Fig .11 shows the runtimes of Sparm, vanilla Flexagon, and Flexagon with Prefetching for the nine different workloads in Table I. We observe that RC Prefetcher brings significant performance gains across all nine workloads, as it significantly improves the hit rate of the StrCache and reduces off-chip memory accesses. Notably, in sets 2 and 3, Sparm outperforms Flexagon with prefetching, demonstrating the superiority of our merging strategy when handling large-scale workloads with high sparsity. Across these nine real-world workloads, Sparm achieves an average speedup of 1.35x compared to the vanilla Flexagon and 1.10x compared to Flexagon with Prefetching.

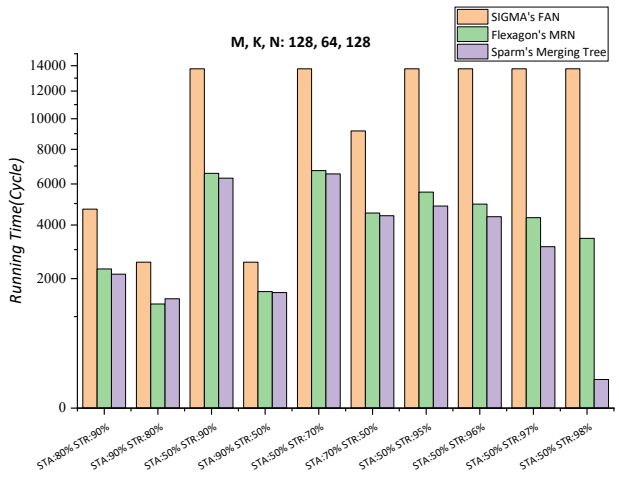

Fig. 9. Performance comparison between SIGMA's FAN, Flexagon's MRN, and Sparm's Merging Tree when running spGEMM with MKN size 128*64*128.

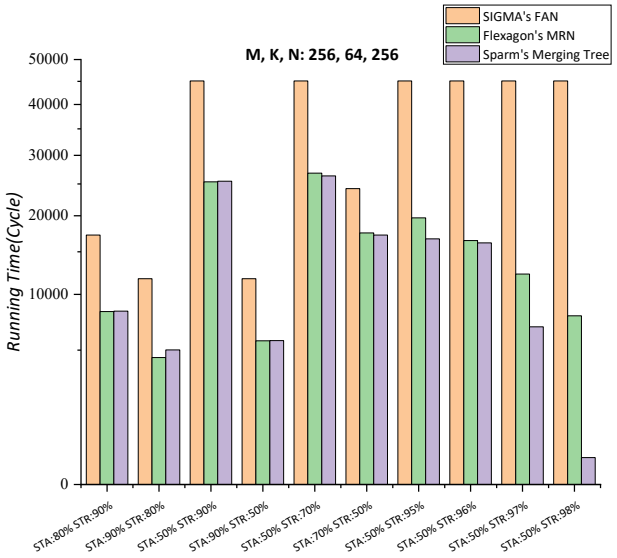

Fig. 10. Performance comparison between SIGMA's FAN, Flexagon's MRN, and Sparm's Merging Tree when running spGEMM with MKN size 256*64*256.

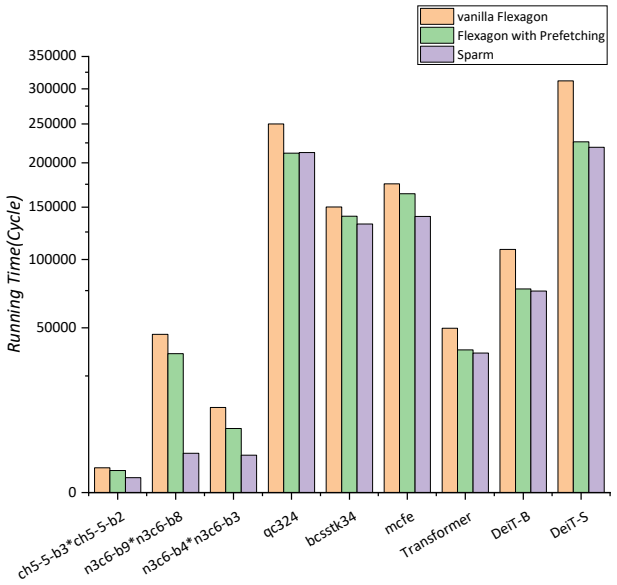

Fig. 11. Performance comparison between vanilla Flexagon, Flexagon with prefetching, and Sparm across 9 sets of real-world workloads.

C. Area and Power

Table II shows the breakdown of area and power for 64-multiplier Sparm. We showcase the results for the main microarchitecture components, including the distribution network, multiplier network, reduction network, StrCache, PSRAM, and Bitindex SRAM. It can be observed that the on-chip memory organization occupies the majority of the accelerator's area, as they are required to store a large number of compressed data and index information. Because the capacity of the StaFIFO and our additional Look-Ahead FIFO is small enough, their area is negligible. It is worth noting that the area of Sparm's merging tree is smaller than that of the MRN. This is because our design does not need to save additional coordinate information and removes the comparators integrated within the adders.

TABLE II
AREA&POWER.

	Area (mm^2)		**Power (mW)**	
Component	**Flexagon**	**Sparm**	**Flexagon**	**Sparm**
Distribution Network	0.04	0.04	2.18	2.18
Multiplier Network	0.07	0.07	3.29	3.29
Reduction Network	0.21	0.19	312	73
StrCache	3.93	3.93	2142	2142
PSRAM	1.03	1.03	538	538
Bitindex SRAM	-	0.51	-	269
Total	5.28	5.77	2998	3027

V. Related Work

The acceleration of sparse matrix multiplication operations has been extensively studied on CPUs and GPUs. For example, Intel MKL [6] and cuSPARSE [12] are widely adopted on CPUs and GPUs, respectively. Most CPUs and GPUs typically only support the Gustavson dataflow and employ different strategies to merge rows of matrix B. Additionally, general-purpose caches lack optimizations for sparse rows/columns. However, Sparm supports multiple merging strategies, and its highly customized memory structure effectively captures the memory access characteristics exhibited by different workloads.

Numerous sparse matrix preprocessing methods have been proposed for both traditional hardware platforms and domain-specific accelerators. Matrix preprocessing on CPUs and GPUs typically aims to create dense tiles to reduce the irregularity of partial outputs, disjoint tiles to minimize communication, or balanced tiles to alleviate load-imbalance issues [20]. Sparse-TPU [4] proposes packing algorithms and index-matching mechanisms to efficiently process sparse matrices in a condensed format. To alleviate the burden caused by the complexity of the greedy search algorithm in Sparse-TPU, HIRAC [15] proposes a rapid packing algorithm named SorPack, which brings the psums that need to be merged closer to each other. Specialized hardware is also designed to leverage SorPack more effectively. Therefore, in our future work, we can integrate Sparm with these preprocessing techniques to support a wider range of applications.

VI. Conclusion

In this paper, we propose Sparm, a novel accelerator that can support multiple spGEMM dataflows. We first conduct a detailed analysis of Flexagon, a prior multi-dataflow spGEMM accelerator. Addressing its inefficient merging strategy based on coordinate matching, we introduce a specialized sparse indexing unit, the Merge Manager. This unit enables regularized accumulation of partial sums for different dataflows, significantly mitigating the frequent congestion issues observed in Flexagon's MRN. Furthermore, to improve the cache hit rate of StrCache when processing large matrices, we developed the RC Prefetcher. This prefetcher adapts to the memory access patterns exhibited by different dataflows, employing different prefetching strategies to preload rows or columns of matrix B. The results demonstrate that Sparm achieves significant performance improvements over prior SOTA spGEMM accelerators on workloads with high sparsity.

Acknowledgment

This work is supported in part by the National Key R&D Project No. 2021YFB0300300, the NSFC 62172430, the STIP of Hunan Province 2022RC3065, and the Foundation of PDL 2023-JKWPDL-02, and the Key Laboratory of Advanced Microprocessor Chips and Systems.

References

[1] Rajeev Balasubramonian et al. "CACTI 7: New tools for interconnect exploration in innovative off-chip memories". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.2 (2017), pp. 1–25.

[2] Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[3] Song Han et al. "Learning both weights and connections for efficient neural network". In: *Advances in neural information processing systems* 28 (2015).

[4] Xin He et al. "Sparse-TPU: Adapting systolic arrays for sparse matrices". In: *Proceedings of the 34th ACM international conference on supercomputing*. 2020, pp. 1–12.

[5] Kartik Hegde et al. "UCNN: Exploiting computational reuse in deep neural networks via weight repetition". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 674–687.

[6] Intel. *Accelerate Fast Math with Intel ONEAPI Math Kernel Library*. 2022. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[7] Valentin Isaac–Chassande et al. "Dedicated hardware accelerators for processing of sparse matrices and vectors: A survey". In: *ACM Transactions on Architecture and Code Optimization* 21.2 (2024), pp. 1–26.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[9] Zhiyao Li et al. "Spada: accelerating sparse matrix multiplication with adaptive dataflow". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 747–761.

[10] Francisco Muñoz-Martínez et al. "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 252–265.

[11] Francisco Muñoz-Martínez et al. "Stonne: Enabling cycle-level microarchitectural simulation for dnn inference accelerators". In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2021, pp. 201–213.

[12] NVIDIA. *cuSPARSE :: CUDA Toolkit Documentation*. 2022. URL: https://docs.nvidia.com/cuda/cusparse/index.html.
[13] Subhankar Pal et al. "Outerspace: An outer product based sparse matrix multiplication accelerator". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 724–736.
[14] Eric Qin et al. "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 58–70.
[15] Hesam Shabani et al. "Hirac: A hierarchical accelerator with sorting-based packing for spgemms in dnn applications". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 247–258.
[16] Nitish Srivastava et al. "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.
[17] Bo Wang et al. "SparGD: A Sparse GEMM Accelerator with Dynamic Dataflow". In: *ACM Transactions on Design Automation of Electronic Systems* 29.2 (2024), pp. 1–32.
[18] Rui Xu et al. "A Survey of Design and Optimization for Systolic Array-Based DNN Accelerators". In: *ACM Computing Surveys* 56.1 (2023), pp. 1–37.
[19] Rui Xu et al. "Heterogeneous systolic array architecture for compact cnns hardware accelerators". In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2021), pp. 2860–2871.
[20] Guowei Zhang et al. "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 687–701.
[21] Zhekai Zhang et al. "Sparch: Efficient architecture for sparse matrix multiplication". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 261–274.