SpaHet: A Software/Hardware Co-design for Accelerating Heterogeneous-Sparsity based Sparse Matrix Multiplication

Haoqin Huang[†], Pengcheng Yao[†‡*], Zhaozeng An[†], Yufei Sun[†], Ao Hu[†],
Peng Xu[†‡], Long Zheng[†], Xiaofei Liao[†], Hai Jin[†]
[†]National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, 430074, China
[‡]Zhejiang Lab, Hangzhou, 311121, China
{hqhuang,pcyao,anzz,yfsun,ahu,longzh,xfliao,hjin}@hust.edu.cn,xup@zhejianglab.com

ABSTRACT

Sparse general matrix-matrix multiplication is widely used in data mining applications. Its irregular memory access patterns limit the performance of general-purpose processors, thus motivating many FPGA-based hardware innovations in recent years. Nevertheless, existing accelerators fail to efficiently support heterogeneous input matrix sparsity, which is universal in various real-world applications. With in-depth experimental analysis, we observe that their performance is bottlenecked by their fixed tiling mechanisms, which only alleviate the irregularity of one input matrix. Based on the observation, we propose SpaHet, a software/hardware co-design to accelerate heterogeneous-sparsity based sparse matrix multiplication. SpaHet adopts a dual-adaptive sliding window mechanism to cover the reuse characteristics of both input matrices simultaneously. With a specialized exploration algorithm, the window-based mechanism can automatically find the optimal tiling strategy instead of applying a fixed one based on empirical experience. A sparsity-aware merge tree is also proposed to maximize the output matrix reuse via accumulating intermediate results thoroughly. Our results on a Xilinx Alveo U280 accelerator card show that SpaHet outperforms state-of-the-art CPU-, GPU- and FPGA-based solutions by 7.71×, 1.1×, and 2.74× in performance, respectively.

1 INTRODUCTION

Sparse general matrix-matrix multiplication (SpGEMM) plays an essential role in a broad range of real-world applications, including graph analytics [11], linear algebra [8], and large language models [3]. Through skipping ineffectual zero elements, SpGEMM can efficiently reduce computation operations compared to dense matrix multiplication. However, SpGEMM generally suffers from unpredictable traversal of sparse non-zero elements when running on general-purpose processors (e.g., CPUs) [6]. The memory access latency increases due to frequent cache misses, resulting in deficient memory performance.

As Dennard scaling ends, these inherent inefficiencies make general-purpose processors unlikely to provide considerable performance for SpGEMM. Instead, FPGA is considered to be a promising candidate because of its high reconfigurability and energy efficiency. In the past few years, lots of research efforts have been made to develop FPGA-based SpGEMM accelerators [4, 7, 9]. With dedicated memory optimizations, SpGEMM accelerators can deliver tenfold performance gains over general-purpose processors.

With the development of SpGEMM applications, the need for hardware accelerators to efficiently support *heterogeneous sparsity* is increasing. In real-world scenarios, the sparsity of two matrices multiplied can vary significantly. For example, the graph convolution network performs sparse matrix multiplication $A \times X$ to extract vertex features in the aggregation phase [12]. The graph matrix A is highly sparse and usually has less than 0.0001% nonzeros. In contrast, the feature matrix X is much denser, with only moderate sparsity around 1% to 90% [5]. This heterogeneity sparsity results in different data locality of two input matrices, thus requiring differential on-chip memory management.

However, existing SpGEMM accelerators are orchestrated to perform well only when the sparsity of two input matrices is similar. The core reason is that each of them adopts a fixed tiling mechanism that maximizes data reuse of one input matrix but sacrifices the other. As a result, the performance would suffer from bad input matrix reuse if the workload violates this rigid design assumption. For instance, Spaghetti [7] splits the first input matrix into multiple horizontal tiles and stores nonzeros of the tile processed on-chip to improve memory efficiency. Although Spaghetti maximizes the input reuse of the first input matrix, the nonzeros in the second input matrix are seldom reused. From the experimental results in Section 2.2, we observe that Spaghetti performs 4.7× more off-chip memory accesses compared to the optimal case, i.e., on-chip memory is large enough to reuse all data.

In this paper, we propose SpaHet, a software/hardware co-design to efficiently accelerate SpGEMM with heterogeneous matrix sparsity. In software, SpaHet introduces a *dual-adaptive sliding window* (DASW) mechanism to improve the memory efficiency of both input matrices. Instead of adopting a fixed tiling strategy based on empirical experience, DASW tiles two input matrices differentially via sliding their windows upon a generalized matrix multiplication model. In order to cover diverse reuse characteristics, DASW also adopts an exploration method to automatically find the optimal tiling strategy of any input matrices given. Moreover, SpaHet proposes a novel memory allocation-specific reordering algorithm to improve the load balance of DASW. In hardware, SpaHet designs a

*Pengcheng Yao is the corresponding author. This work is supported by the National Key Research and Development Program of China under Grant No. 2023YFB4502300.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655944

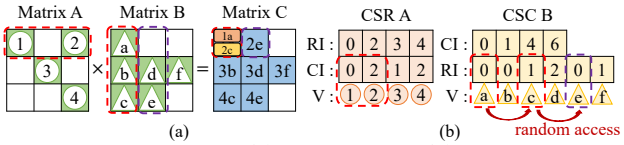


Figure 1: An example of (a) SpGEMM and (b) random memory accessed in SpGEMM

sparsity-aware merge tree to improve the memory efficiency of the output matrix further. The multiplication results destinated to the same memory address are completely accumulated before writing back to off-chip memory, thus maximizing output reuse.

In summary, this paper makes the following contributions:

- We revisit existing SpGEMM accelerators in heterogeneous matrix sparsity scenarios and identify their fixed tiling strategy as the main reason for low memory efficiency.
- We propose SpaHet, a software/hardware co-design that can achieve near-optimal data reuse for input and output matrices via automatic adaptations.
- We implement SpaHet in RTL and evaluate it with a broad range of matrix datasets. Our experimental results indicate that SpaHet achieves 7.71×, 1.1×, and 2.74× performance speedups over the state-of-the-art CPU-, GPU-, and FPGA-based solutions, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Sparse-Matrix Sparse-Matrix Multiplication

SpGEMM performs multiplication between two sparse matrices A and B to generate a sparse matrix C. The computation is conducted by the following formula $C_{ij} = \sum_k A_{ik} \times B_{kj}, i \in [1, N], j \in [1, M]$, where N is the row size of A and M is the column size of B. Figure 1(a) illustrates an example of SpGEMM where white and green cubes represent zero and nonzero elements, respectively. Compared to dense matrix multiplication, SpGEMM skips multiplication of any $A_{ik} = 0$ or $B_{kj} = 0$ to minimize computation workloads. The compressed data structures, like *Compressed Sparse Row* (CSR), are also leveraged to reduce redundant memory accessed on zero elements.

Although SpGEMM is efficient in computation complexity, its irregular traverse on nonzero elements introduces tremendous random memory accesses. Figure 1(b) shows compressed data of two input matrices in Figure 1(a), including *row index* (RI), *column index* (CI), and *Values* (V) arrays. When multiplying the first A row with the first B column, a random access is made between VB_{11} and VB_{31} because the multiplication of $0 \times VB_{21}$ is skipped. Similarly, performing $\sum_k VA_{13} \times VB_{32}$ will generate another random access between VB_{31} and VB_{32}. Such irregular memory behavior results in frequent cache misses, which dominates the performance of SpGEMM. Consequently, general-purpose processors often deliver extremely low throughput despite their high theoretical peaks (e.g., 1GFLOPS vs. 4TFLOPS in GPU) [11].

2.2 Pitfalls of Existing SpGEMM Accelerators

To improve memory efficiency, previous SpGEMM accelerators employ a tiling-based execution model. As shown in Figure 2(a), the large input matrices A and B are partitioned into multiple independent tiles, which are relatively small (e.g., several megabytes). SpGEMM accelerators only perform one tile of computation at a time to ensure that the majority of input/output data can be stored

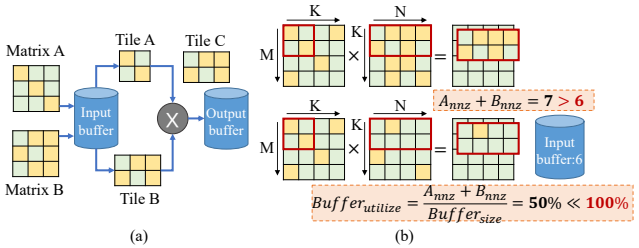


Figure 2: An example of (a) a typical design for existing SpGEMM accelerators and (b) their limitations

Datasets	N	Sparsity
RMAT14	16384	10^{-7}~10^{-1}
RMAT15	32768	10^{-7}~10^{-1}
RMAT16	65536	10^{-7}~10^{-1}
RMAT17	131072	10^{-7}~10^{-1}

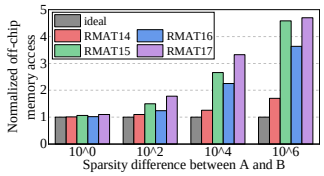


Figure 3: The off-chip communications of SpGEMM accelerators on different input matrices

in specialized on-chip memories, i.e., input/output buffers in Figure 2(a). All irregular nonzero elements are accessed on-chip, thus reducing off-chip communications effectively.

When tiling matrices, existing works adopt a rigid assumption that two input matrices follow similar structures. As a result, they usually adopt a fixed tiling mechanism with the same strategy for all matrices. For instance, Spaghetti [7] determines the tile size of both input matrices based on the data reuse of matrix A only. Nevertheless, the sparsity of input matrices in real-world applications varies greatly. This heterogeneous sparsity results in distinct data locality, which requires differential on-chip memory management. In other words, a tiling mechanism that works well on reusing matrix A will always fall short on the other. Therefore, existing works suffer from performance degradation or even negative acceleration when processing heterogeneous input matrices.

Figure 2(b) illustrates the challenges introduced by heterogeneous sparsity. We take a fixed grid tiling strategy, which is widely adopted in SpGEMM accelerators, as an example. When matrix B is denser than matrix A, partitioning matrix B based on the row/column size of matrix A will result in a huge number of nonzeros that exceed the cache capacity. Consequently, frequent cache misses on matrix B will occur, resulting in suboptimal data reuse. Conversely, if matrix A is denser than matrix B, reducing the tile size to allocate more space for reading matrix B can lead to inadequate cache utilization, requiring more cycles for computation.

To investigate the performance impact arising from heterogeneous sparsity, we conduct several experiments on a Xilinx U280 accelerator card. The state-of-the-art SpGEMM accelerator, Spaghetti, is evaluated by enumerating off-chip memory accesses on the input matrices listed in Figure 3. As shown in Figure 3, Spaghetti performs well when two input matrices are similar. The number of off-chip communications is close to the ideal case where the on-chip memory is large enough to reuse all data. However, when the sparsity difference increases, Spaghetti suffers from 4.7× more off-chip memory communications than the optimal case, demonstrating the limitations of using a fixed tiling method for all matrices.

In summary, existing SpGEMM accelerators adopt fixed tiling mechanisms that only maximize on-chip data reuse of either matrix

A or B. Such design philosophy suffers when processing input matrices with heterogeneous sparsity. As a result, it is of great necessity to consider the reuse characteristics of both input matrices and find an appropriate tradeoff between them. To this end, we propose a novel SpGEMM accelerator, SpaHet, which dynamically and separately tiles input matrix A and B, achieving minimal off-chip communications.

3 DUAL-ADAPTIVE SLIDING WINDOW

3.1 Dual-Window Based Tiling space

In general, the data reuse of existing accelerators can be described by the following formula, the date reuse A_{reuse}, B_{reuse}, C_{reuse} of matrices A, B, C is $\frac{T_n}{N}$, $\frac{T_m}{M}$, and $\frac{T_k}{K}$, respectively. As shown in Figure 4(a), T_n, T_m, T_k respectively represent the length of the three dimensions N, M, K of the matrix. Because the tile data being processed must be accessed on-chip, the nonzero elements in three matrices have to satisfy the following formula.

$$\alpha \cdot T_n \cdot T_m + \beta \cdot T_m \cdot T_k + \gamma \cdot T_n \cdot T_k < \sigma \tag{1}$$

where α, β, γ are non-variables and σ is the on-chip cache capacity.

Since existing accelerators assume that both input matrices follow similar structures, they simply change the tile size of A to maximize the value in Equation 1. When implemented on a two-dimensional matrix, such design acts as a window sliding vertically and horizontally to find the best tile size for matrix A. While such sliding window mechanism can maximize the data reuse of matrix A, it always sacrifices the data reuse of B, as discussed in Section 2.

To comprehensively alleviate the locality characteristics in both input matrices, we introduce a dual-adaptive sliding window mechanism for the exploration of the optimal tiling strategy, which is a search space that not only encompasses all previously proposed tiling methods but also reveals novel tiling approaches. As shown in Figure 4(a), it contains a $T_n \times T_{k1}$ sliding window in matrix A and a $T_{k2} \times T_m$ sliding window in matrix B. When exploring tiling methods, both windows initially slide along dimension K to determine T_{k1} and T_{k2}, and then independently slide along dimensions N and M to determine T_n and T_m according to the total *non-zero* (nnz) numbers in the sliding windows.

For instance, as shown in Figure 4(b), we assume that the cache capacity is 6. Initially, we slide along the dimension K to determine the dimensions T_{k1} and T_{k2}, both of which are set as 4. Next, we slide along dimensions N and M to determine a pair of T_n and T_m, ensuring total nonzero elements in sliding windows $T_n \times T_{k1}$ and $T_{k2} \times T_m$ can be fitted into the cache. In our example, we set T_n as 4 and T_m as 3, generating a tile pair in matrix A and B. Subsequently, we repeatedly slide and generate tiles for the selected T_{k1} and T_{k2}, until all nonzero elements in A and B are handled. This design ensures that, during sliding along dimensions N and M, each generated tile cannot exceed the cache capacity. Afterward, the two sliding windows continue to slide along dimension K, initiating the next round of bidirectional sliding.

Nevertheless, an unrestricted exhaustive search is excessively time-consuming, with time complexity reaching $O(n^4)$. Consequently, pruning operations become imperative. Initially, we align the dimensions T_k of both sliding windows, reducing the time complexity to $O(n^3)$. Regardless of how we tile the K dimension, it does not affect matrix reuse. Maintaining different dimensions T_{k1} and T_{k2}

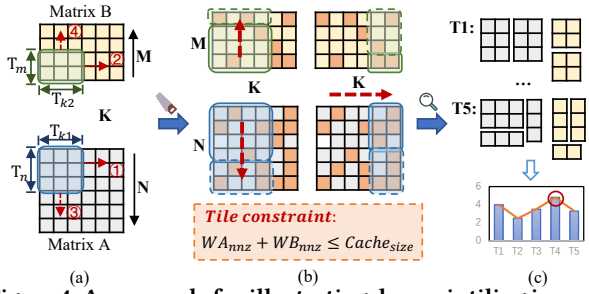


Figure 4: An example for illustrating dynamic tiling in matrix multiplication, including (a) dual-adaptive sliding windows, (b) sliding windows to tiles, and (c) performance evaluation model to find optimal tiling

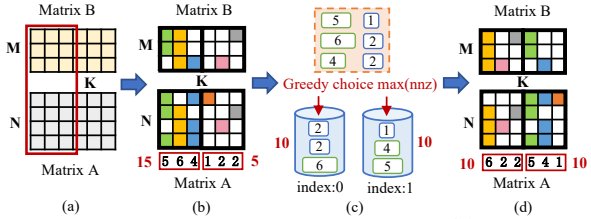


Figure 5: An example of reordering with (a) segment, (b) before reordering, (c) greedy matching strategy, and (d) after reordering

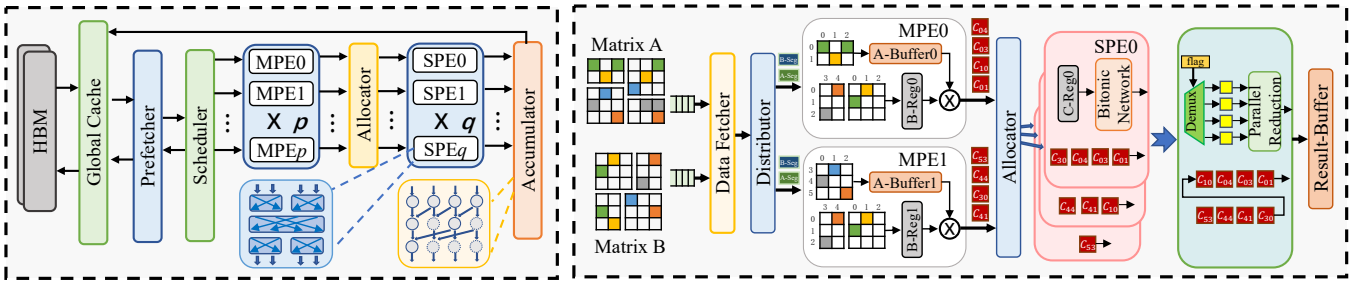
for the two sliding windows results in unnecessary data loading, wastage of memory, and poor input reusability.

Additionally, we limit the sum of nnz within each sliding window to stay below the cache capacity. Exceeding the capacity leads to frequent cache evictions and numerous off-chip memory accesses, as each PE requires on-chip memory loading during computation. We introduce a threshold for difference in the number of multiplication operations between various groups of sliding windows, ensuring effective load balancing for different PEs during computation.

3.2 Automatic Tiling Space Exploration

Automatic tiling can generate numerous tiling methods, so it is necessary to cost-effectively select the optimal one that maximizes the overall performance, as shown in Figure 4(c). An intuitive idea is to execute the program on FPGA to measure actual performance for evaluation. However, this will incur substantial overhead, including execution time and energy consumption. Using an analytical model to predict performance and extract the optimal tiling methods is a better alternative. Nonetheless, it is challenging to construct an analytical model that depends on multiple parameters, such as algorithm characteristics, matrix properties, and hardware specifics.

We propose a performance analysis model based on data access patterns. SpGEMM is a memory-bound application, where data movement occupies a significant amount of execution time. Furthermore, various tiling methods have almost no impact on the computational cost for SpGEMM. As a result, we can use the memory access patterns to model performance for predicting the relationship between tiling methods and performance. In fact, matrix A is always accessed once from off-chip HBM to on-chip buffer, so we can ignore the access to matrix A, and only take matrices B and C into consideration to simplify the model and improve exploration efficiency. Specifically, the read operation on B and read/write the operation on C are the key factors in determining performance.



(a) Architecture overview of SpaHet

(b) An example of the workflow

Figure 6: The SpaHet architecture

Performance can be predicted by accumulating the memory access time of each pair of tiles, as shown in Equation. 2. Since the memory accesses are pipelined, the time for each tile can be represented as the multiplication between the average operation counts of all PEs and the maximum memory access time among *reading B* (R_B), *reading C* (R_C), and *writing C* (W_C).

$$ExT = \sum_{i=0}^{N_{cycles}} [\frac{WA_{cnt} \times WB_{cnt}}{PE_{cnt}} \times max(R_B + R_C + W_C)] \quad (2)$$

where

$$\begin{cases} R_B = WB_{nnz} \\ R_C = W_{C_{i-1}} + \sum_{j=0}^{T_k} WA_{col} \cdot WB_{row} \times (1-\lambda) \\ W_C = R_{C_i} \times (\alpha + \beta \cdot i) \end{cases} \quad (3)$$

Specifically, N_{cycles} represents the number of execution rounds, WA_{cnt} and WB_{cnt} indicate the sliding window count in matrices A and B for each round, and PE_{cnt} represents the number of parallel *processing elements* (PEs), R denotes data reading time, W represents data writing time, and WB_{nnz} quantifies the number of nnz in the sliding window of matrix B.

R_C is determined by the calculation of the intermediate matrix, which involves elements from both sliding windows and the intermediate matrix produced by reading the merged outcome from the previous round. To estimate the number of nnz in the intermediate matrix, we multiply the MAC count by the ratio of unmerged elements. WA_{col} and WB_{row} represent the number of nnz in a column and a row of the sliding window, respectively, while λ denotes the merging ratio, determined based on output affinity between elements. W_C is contingent upon the result achieved after merging two intermediate matrices. As the number of merging iterations increases, the merging ratio gradually decreases. We introduce α, whose value changes with an increase in the number of merge iterations, while β remains a constant.

3.3 Affinity and Distribution-based Reordering

The difference between the unknown matrix data output size and static memory allocation in RTL leads to memory allocation based on conservative nnz estimates, causing unevenness due to the power-law distribution. This unevenness limits memory size to the matrix with the maximum nnz, resulting in a long-tail effect and reduced runtime memory utilization. To address this, we alleviate the long-tail effect through tiling and reordering for improved matrix uniformity and enhance output reuse by increasing data reuse between consecutive rows, thereby boosting performance.

We initially treat a column from matrix A and a row from matrix B as a single data unit. Subsequently, as shown in Figure 5(a), we implement a cost-effective vertical tiling strategy to tile the shared dimension K of the matrices into segments of length W. As illustrated in Figure 5(b), this results in K/W partitions, where the index i^{th} range spans from $(i-1) \times W$ to $i \times W$. Subsequently, as depicted in Figure 5(c), we iteratively choose the segment with the lowest total nnz count. Greedily, we assign the remaining data unit with the highest nnz count to this segment. When multiple units share the same maximum nnz count, and their nnz distribution positions vary, selecting positions with more common nnz between two units results in a smaller edit distance and higher similarity. We define this as affinity, represented by Equation 4, where D_i denotes the nnz count in the i^{th} data unit. By assessing the affinity between each data unit and the segment, we choose the data unit with the highest affinity to integrate into the segment. This strategy facilitates the merging of more data during computation, enhancing output reuse. Allocation to a segment stops when its total nnz count exceeds that of other segments, and we proceed to smaller segments until all elements are allocated.

$$Affinity(D_i, D_j) = \frac{|D_i \cap D_j|}{|D_i \bigcup D_j|} \quad (4)$$

As seen in Figure 5(d), through this reordering method, we locally optimize matrix uniformity, naturally alleviating the long-tail effects and uneven load issues caused by a power-law distribution.

4 HARDWARE DESIGNS

4.1 Hardware Overview

Figure 6(a) shows the architecture overview of SpaHet, including the prefetcher, scheduler, *multiplication processing elements* (MPE), *sorting processing elements* (SPE), and accumulator. The prefetcher is responsible for prefetching matrix data from off-chip memory to a scratchpad-based global cache based on the tiling strategy generated by the dual sliding window mechanism. Once matrix data is available for execution, the schedule dispatches it to MPEs evenly. The architecture consists of multiple MPEs connected with a crossbar switch. Each MPE contains a multiplication unit for performing multiplication operations. The multiplication results destinated to the same output location are merged by the accumulator, and written back to the global cache.

4.2 Sparsity-aware Merge Tree

Although DASW maximizes the reuse of matrix data, it increases the irregularity because of the non-uniform tiling structure. Specifically, the number of addition operations on different C_{ij} varies significantly. Consequently, the results of MPEs in the same cycle might update random locations in matrix C. When multiple results

Table 1: Matrix DataSets used in the Experiments

Matrix	n	nnz	Sparsity
exdata_1 (EX)	6.0k	1.14M	3.16×10^{-2}
mycielskian13 (MC)	6.1k	0.61M	1.63×10^{-2}
ship_001 (SP)	34.9k	2.34M	1.91×10^{-3}
Chebyshev4 (C4)	68.1k	5.38M	1.15×10^{-3}
filter3D (FL)	106.4k	1.41M	1.24×10^{-4}
cage12 (CG)	130.2k	2.03M	1.20×10^{-4}
amazon0312 (AM)	400.7k	3.20M	1.99×10^{-5}
com-DBLP (CD)	317.1k	1.05M	1.04×10^{-5}
com-Youtube (YT)	1.1M	2.99M	2.32×10^{-6}
com-LiveJournal (LJ)	4.0M	34.68M	2.17×10^{-6}
RMAT15-y (R15-y)	32.8k	$n^2 \times 10^{-y}$	10^{-y}
RMAT16-y (R16-y)	65.5k	$n^2 \times 10^{-y}$	10^{-y}
RMAT17-y (R17-y)	131.1k	$n^2 \times 10^{-y}$	10^{-y}

try to update the same C_{ij}, the pipeline must stall to avoid data conflicts, resulting in significant performance degradation.

To cope with this problem, we propose a sparsity-aware merge tree that can merge random multiplication results without pipeline stalls. Specifically, it features a two-stage design that is fully pipelined. In the first stage, we adopt a lightweight bitonic sorting network to sort the multiplication results in an ascending order based on their destination index. With an ordered input sequence, the merging problem can be simplified to a prefix sum problem by restarting the addition when finding a new destination index. Therefore, we adopt a prefix sum network in the second stage to merge all multiplication results destinated to the same locality.

Figure 6(b) shows an example of the workflow. The scheduler dispatches the input matrix tiles prefetched from off-chip memory to the MPEs for multiplication. The multiplication results are forwarded to the allocator, where they are assigned to the sorter corresponding to the remainder group based on their row indices. The sorter facilitates data flow to the bitonic sort network which sorts data based on row and column numbers. The sorted results are transmitted to the merger's queue and introduced into the parallel reduction tree, which merges data with matching row and column numbers. Finally, the merged results are written back to the HBM.

5 EVALUATION

5.1 Experimental Setup

SpaHet Settings. We implement SpaHet on a Xilinx Alveo U280 FPGA accelerator card. The FPGA chip in U280 provides 1.3M LUTs, 2.6M Registers, and 9MB BRAM reconfigurable hardware resources. SpaHet consists 16 PEs connected to 2 HBM stacks, which provide 460GB/s aggregated memory bandwidth. We use BRAM resources to implement a 4.8MB on-chip memory, which is evenly sliced to all PEs. The clock rate and resource utilization of SpaHet are obtained by Xilinx Vitis 2021.2. SpaHet runs at 200MHz in all experiments.

Baselines. For general-purpose processors, we compare SpaHet with CPU-based Intel MKL and GPU-based cuSPARSE. Both of them are well-optimized for linear algebra and widely adopted in related works. For specialized accelerators, we compare SpaHet with a state-of-the-art SpGEMM accelerator Spaghetti [7]. All experiments are conducted on a server with two 28-core Intel Xeon Gold 6330 CPUs clocked at 2GHz, 1TB DDR4 memory, an Nvidia A100 GPU, and a Xilinx Alveo U280 FPGA.

Datasets We perform SpGEMM on both real-world and synthetic matrix datasets listed in Table 1. The real-world datasets are collected from SuiteSparse Matrix Collection [2], and synthetic

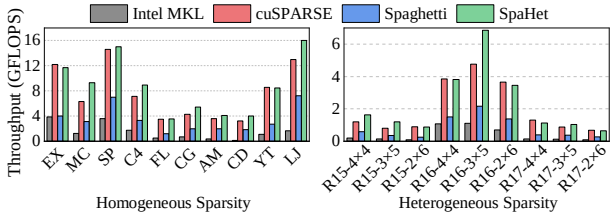


Figure 7: Throughput of SpaHet against Intel MKL, cuSPARSE, and Spaghetti

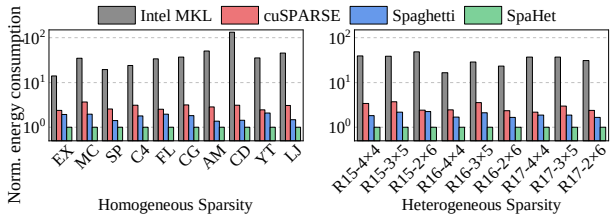


Figure 8: Energy consumption of Intel MKL, cuSPARSE, Spaghetti, and SpaHet, normalized to SpaHet

datasets are produced by the Kronecker Generator [1]. These datasets have different scales and sparsity, thus representing a broad range of real-world demand on SpGEMM. For SpGEMM with homogeneous sparsity, we utilize real-world datasets and perform $A \times A$ for evaluation. For heterogeneous sparsity, we generate synthetic datasets with various sparsity (from 10^{-2} to 10^{-6}) for each scale (from 15 to 17), and evaluate the computation of $A \times B$. For example, R15-y in our datasets represents a generated dataset whose n is 2^{15} and sparsity is 10^{-y}, and R15-y×z in our evaluation results represents the multiplication between R15-y and R15-z.

5.2 Overall Performance

Figure 7 depicts the throughput results. We use *Giga Floating-point Operations per Second* (GFLOPS) for throughput, which is calculated by |Operations| / Time. Each $C_{ij} += A_{ik} \times B_{kj}$ is counted as two operations, including one multiplication and one addition.

SpaHet vs. General-Purpose Processors. SpaHet outperforms Intel MKL and cuSPARSE by 7.71× and 1.1× on average, respectively. The performance improvement mainly comes from the reduced memory access latency and off-chip communications. First, the random accesses on nonzero elements cause frequent cache misses on CPUs and GPUs, resulting in significant miss penalty. In contrast, SpaHet can access the majority of the nonzero elements through tiling-based on-chip memory management, thus minimizing memory access latency. Second, SpaHet adopts an automatic tiling design to reduce off-chip communications via reusing input matrix data. Specifically, SpaHet achieves the highest speedup on MC because MC has a relatively higher matrix sparsity, leaving more memory inefficiencies that can be improved.

SpaHet vs Spaghetti. SpaHet achieves 2.74× performance improvement over Spaghetti on average. The main reason is the better data reuse on both input matrices because of a more flexible tiling mechanism. Spaghetti adopts a fixed tiling mechanism that partitions both input matrices based on the data reuse of matrix A. It has to redundantly access matrix B from off-chip memory, resulting in significant memory inefficiencies. As a comparison, SpaHet uses a flexible sliding window mechanism that automatically searches an optimal tiling strategy with maximal data reuse for all input

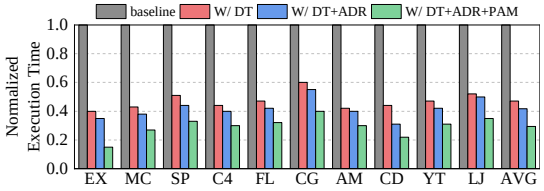


Figure 9: Effectiveness of DT, ADR, and PAM

matrices. Specifically, SpaHet obtains the highest performance improvement when multiplying RMAT15-2 with RMAT15-6 because of the significantly different sparsity between two matrices.

5.3 Resource Utilization and Energy Savings

Resource Utilization. Because it is hard to compare the area of two FPGA implementations directly, we use hardware resource utilization as an alternative. Table 2 shows the resource utilization of SpaHet and Spaghetti. When implementing with the same number of PEs, SpaHet uses slightly more resources compared to Spaghetti. This is because the flexible tiling mechanism adopted in SpaHet generates many partitions with non-uniform structures. Consequently, SpaHet has to introduce a mapping unit to map flexible partitions to a fixed on-chip memory at runtime. Fortunately, the mapping unit is efficient and introduces only negligible overheads (about 2.7% LUTs and 4.3% REGs).

Table 2: Resource Utilization of Spaghetti and SpaHet

Accelerators	LUT	REG	BRAM
Spaghetti	44.5%	37.1%	47.8%
SpaHet	51.7%	47.7%	58.6%

Energy Savings. Figure 8 shows energy consumption results of SpaHet and baselines. Compared to general-purpose processors, SpaHet consumes 38.3× and 2.94× less energy against MKL and cuSPRASE on average. This is because of the energy-efficient characteristics of the FPGA platform and domain-specific specializations in SpaHet. Compared to SpGEMM accelerator Spaghetti, SpaHet saves 1.87× energy on average. The main reason is that the flexible tiling mechanism adopted in SpaHet significantly reduces off-chip communications, which consume most of the energy.

5.4 Effectiveness

Figure 9 shows the benefit breakdown of software/hardware designs proposed in this paper, including *dynamic tiling* (DT), *affinity and distribution-based reordering* (ADR), and *parallel accumulation and merge* (PAM). For comparison, we adopt a baseline design with a fixed grid tiling mechanism, which is widely adopted in SpGEMM accelerators [10]. Based on the baseline, we incrementally integrate the three designs to analyze the benefits brought by them.

As shown in Figure 9, most of the speedups come from DT, which achieves 2.13× speedups compared to the baseline. This is because the grid tiling mechanism in the baseline always sacrifices the data reuse of one input matrix. In contrast, DT comprehensively analyzes the overall data reuse based on a performance prediction model. As shown in Figure 10, DT provides accurate performance predictions with a deviation of 4.27%. Based on predictions, DT automatically searches for the optimal tiling strategy, thus maximizing overall data reuse compared to the baseline.

Based on DT, ADR further improves local uniformity to alleviate the waste of memory allocation caused by uneven distribution.

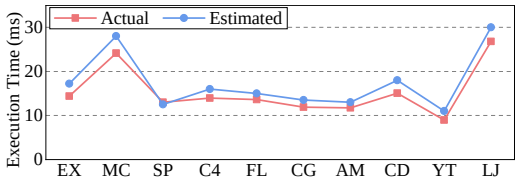


Figure 10: Comparison of estimated execution time and actual running time of performance evaluation model in DT

However, ADR comes at the cost of significant time consumption, resulting in only 1.13× performance improvement. Due to the performance mismatch between computing and merging units in hardware designs, PAM improves the efficiency of merging significantly, thus improving the overall performance by 1.42×.

6 CONCLUSIONS

We introduce SpaHet, a software/hardware co-design for accelerator SpGEMM with heterogeneous matrix sparsity. SpaHet first proposes a dual-adaptive sliding window to automatically search for an optimal tiling strategy that maximizes the data reuse of all matrices. In addition, SpaHet provides a reordering algorithm that combines partition and affinity to alleviate the long-tail effect associated with memory allocation. Based on the software designs, SpaHet architects a hardware template and designs a sparsity-aware merge tree to mitigate pipeline stalls by maximizing the data reuse of the output matrix. Our evaluation on various input matrices shows that SpaHet outperforms the CPU-based library Intel MKL, the GPU-based library cuSPARSE, and the FPGA-based accelerator Spaghetti by 7.71×, 1.1×, and 2.74×, respectively.

REFERENCES

[1] James Alfred Ang, Brian W. Barrett, Kyle Bruce Wheeler, and Richard C. Murphy. 2010. Introducing the graph 500. *Cray User's Group* (2010).
[2] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM TOMS* 38, 1 (2011), 1–25.
[3] Hongxiang Fan, Thomas Chau, Stylianos I. Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D. Lane, and Mohamed S. Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *Proceedings of MICRO*. 599–615.
[4] Noble G, Nalesh S, and Kala S. 2023. MOSCON: Modified Outer Product based Sparse Matrix-Matrix Multiplication Accelerator with Configurable Tiles. In *Proceedings of VLSID*. 264–269.
[5] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *Proceedings of MICRO*. 922–936.
[6] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2008. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of PDP*. 283–292.
[7] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *Proceedings of HPCA*. 84–96.
[8] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. In *Proceedings of OOPSLA*. 77:1–77:29.
[9] Shiqing Li, Shuo Huai, and Weichen Liu. 2023. An Efficient Gustavson-based Sparse Matrix-matrix Multiplication Accelerator on Embedded FPGAs. *IEEE TCAD* 42, 12 (2023), 4671–4680.
[10] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *Proceedings of PPoPP*. 90–106.
[11] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *Proceedings of HPCA*. 724–736.
[12] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *Proceedings of HPCA*. 15–29.