HiSpMM: High Performance High Bandwidth Sparse-Dense Matrix Multiplication on HBM-equipped FPGAs

AHMAD SEDIGH BAROUGHI, MANOJ B. RAJASHEKAR, AKHIL R. BARANWAL, and ZHENMAN FANG, Simon Fraser University, Canada

Sparse Matrix-Dense Matrix Multiplication (SpMM) is a critical operation in scientific computing, machine learning, and graph analytics. However, accelerating SpMM on FPGAs presents major challenges due to irregular memory access patterns and imbalanced workload distribution. In this work, we address a fundamental bottleneck in SpMM acceleration on High Bandwidth Memory (HBM)-equipped FPGAs: workload imbalance among processing elements (PEs). Additionally, we mitigate a scalability barrier present in state-of-the-art designs—namely, the tight coupling between PEs and HBM channels for dense matrix access. Furthermore, we provide an automated design space exploration framework.

We propose HiSpMM, a high performance SpMM accelerator architecture that introduces Dense Row Sharing to mitigate PE under-utilization by distributing heavy-row computations, a decoupled HBM access mechanism to allow independent scaling of PEs and memory bandwidth, and an automation tool that optimizes design parameters according to matrix structure-specific properties and user-defined hardware constraints. Our design achieves a geomean of 5.81× speedup and 5.75× energy efficiency improvement for imbalanced matrices compared to state-of-the-art designs, while also maintaining competitive performance for balanced matrices on AMD/Xilinx U280 HBM FPGA board. Our HiSpMM project will be open sourced in near future at https://github.com/SFU-HiAccel/HiSpMM.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

Additional Key Words and Phrases: SpMM; Imbalanced Workload; FPGA Accelerator; High Level Synthesis; Design Space Exploration

ACM Reference Format:
Ahmad Sedigh Baroughi, Manoj B. Rajashekar, Akhil R. Baranwal, and Zhenman Fang. 2025. HiSpMM: High Performance High Bandwidth Sparse-Dense Matrix Multiplication on HBM-equipped FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (October 2025), 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Sparse Matrix-Dense Matrix Multiplication (SpMM) is a fundamental mathematical operation that finds widespread use in many areas, including scientific computing [2], linear algebra solvers [5, 18], machine learning [19], neural networks [13], graph neural networks [1, 4, 6], and graph analytics [7, 12, 16]. It is expressed as multiplication of a sparse matrix (A) and a dense matrix (B), resulting in a new dense matrix (C): $\mathbf{C} = \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}$.

Sparse matrices are characterized by irregular and unpredictable memory access patterns arising from nonzero elements scattered across arbitrary indices. The random memory access hinders efficient data movement and parallel execution. High throughput processing requires reorganizing and distributing nonzero elements across multiple High Bandwidth Memory (HBM) channels. For high performance SpMM, maintaining balanced workloads across processing elements (PEs) is a core challenge. An uneven distribution of nonzero elements can significantly degrade PE utilization, as idle time accumulates when lightly loaded PEs must wait for the most loaded PE to complete. This is the main bottleneck that needs careful scheduling and load balancing strategies to fully benefit from compute resources and memory bandwidth.

Another limitation in prior architectures is the tight coupling between the number of HBM channels used for accessing the input/output dense matrix C and the number of PEs. This tight coupling constrains the number of compute units and utilized HBM channels, limiting the available bandwidth for the sparse matrix A or the dense matrix C. Efficient SpMM acceleration requires a more flexible approach to the allocation of HBM channels to compute units to maximize parallelism and resource utilization.

Authors' Contact Information: Ahmad Sedigh Baroughi, asa582@sfu.ca; Manoj B. Rajashekar, mba151@sfu.ca; Akhil R. Baranwal, akhil_baranwal@sfu.ca; Zhenman Fang, Simon Fraser University, Burnaby, Canada, zhenman@sfu.ca.

Manuscript submitted to ACM

The process of automatically determining the optimal architecture design for SpMM involves complex trade-offs among the matrix properties and user-defined hardware constraints such as BRAM/URAM/LUT/DSP usage and the number of available HBM channels. Manual design effort for such optimizations is laborious, which shows the critical need for an automated approach to achieve near-optimal utilization of memory bandwidth and compute resources for a given sparse matrix structure.

To address these challenges, we propose HiSpMM, a high performance, high bandwidth SpMM accelerator for HBM-equipped FPGAs, incorporating the following solutions:

- To tackle the workload imbalance issue, we introduce the Dense Row Sharing strategy. We target rows in the sparse matrix that contain a disproportionately high number of nonzero elements by distributing their computationally expensive nonzero elements across multiple PEs and supporting hybrid inter-row and intra-row parallelism. This strategy successfully reduces workload variance across the PEs, thus enhancing PE utilization and minimizing idle time.
- In order to solve the limitation associated with HBM channel coupling, we decouple the HBM channel allocation for the dense matrix C from the processing elements (PEs). This decoupling allows for increased bandwidth available to the sparse matrix A, thereby enhancing parallelism or alternatively providing increased bandwidth for the dense matrix C if needed. This also allows operation under user-defined constraints and presents the possibility of achieving near-optimal solutions in scenarios where the user has imposed certain resource limitations.
- Finally, we present an automation tool that navigates matrix properties and user-defined hardware constraints to automatically generate customized and high-performing configurations, reducing manual design effort and achieving near-optimal utilization of memory bandwidth and compute resources.

Our experiment setup compares performance with state-of-the-art (SoTA) studies such as Sextans[9, 15] and Leda[20], across various dimensions like throughput and energy efficiency for imbalanced as well as balanced matrices on Alveo U280 HBM-based FPGA. Our proposed designs consistently outperform SoTA SpMM designs on FPGAs, while also providing a higher energy efficiency (measured in GFLOPS per Watt). For imbalanced matrices, we achieve a geomean speedup of 5.81× against Sextans and 5.49× against Leda, while being 5.75× and 4.97× more energy efficient respectively. For balanced matrices, we achieve a geomean speedup of 1.03× against Sextans and 1.06× against Leda, while being 1.07× and 1.14× more energy efficient respectively. Our design consistently maintains high PE utilization due to better scheduling for imbalanced matrices and provides more flexibility to allocate the number of PEs and HBM channels due to our decoupled design. Additionally, the automated design space exploration framework selects the most suitable design configuration, resulting in adaptability across a wide range of sparse matrix workloads.

The remainder of the paper is structured as follows. Section 2 presents the background, challenges, and a high-level comparison of our work over prior studies. Section 3 discusses the architectural aspects of our proposed design, with a load balancing strategy in the pre-processing phase outlined in Section 4. Our proposed automation tool is detailed in Section 5. Section 6 discusses the experimental setup, performance and energy-efficiency for balanced and imbalanced workloads compared to SoTA designs. Additionally, it presents an ablation study and impact of our proposed lightweight DSE. Lastly, Section 7 concludes the paper.

2 Background and Related Work

This section discusses the tiled SpMM algorithm and the associated challenges. Following this, we examine related work and their contributions to address these challenges, and conclude with an explanation of how our contributions compare to the related work.

2.1 Tiled SpMM with Cyclic Row-wise Partition

To manage large matrices, a tiled algorithm is often used. This approach facilitates efficient handling by breaking the matrix into manageable sub-matrices (i.e., tiles). Algorithm 1 describes a tiled and parallelized implementation of SpMM. It computes the result of the generalized SpMM expression: $\mathbf{C} \leftarrow \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}$, where $\mathbf{A} \in \mathbb{R}^{M \times K}$ is a sparse matrix, $\mathbf{B} \in \mathbb{R}^{K \times N}$ is a dense matrix, $\mathbf{C} \in \mathbb{R}^{M \times N}$ is a dense matrix (both input and output), and α and β are scalar coefficients.

As shown in Figure 1, the input sparse matrix $\mathbf{A}$ is partitioned into tiles of size $M_0 \times K_0$, shown in red. The dense matrix $\mathbf{B}$ is partitioned into vertical stripes of size $K_0 \times N_0$, shown in blue. Each tile of $\mathbf{A}$ is multiplied with the corresponding stripe of $\mathbf{B}$ to produce a tile of the output matrix $\mathbf{C}$ of size $M_0 \times N_0$, shown in purple. The dashed arrows represent the computation flow direction: $\mathbf{A}$ tiles are streamed row-wise, while $\mathbf{B}$ stripes are streamed column-wise.

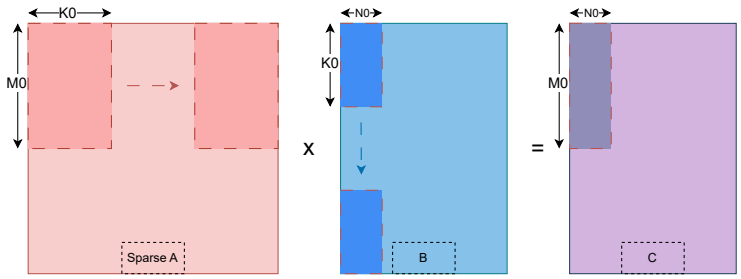


Fig. 1. Matrix partitioning and computation flow in Sparse Matrix–Dense Matrix Multiplication (SpMM).

For each tile in the output matrix **C**, a temporary buffer **Y** of size $M_0 \times N_0$ is initialized to accumulate the partial products of $\mathbf{A} \cdot \mathbf{B}$. The accumulation is performed using cyclic row-wise partitioning across P processing elements (PEs), where each PE is responsible for computing a subset of rows.

As stated in Algorithm 1 (Lines 5 to 14), within each PE, nonzero elements from the corresponding rows of **A** are streamed, and the matching tile from **B** is loaded into an on-chip buffer. For every nonzero a_{mk} in **A**, the algorithm performs a vector multiply-add with the corresponding row of **B**, updating the temporary buffer **Y**. After all partial products are accumulated into **Y**, the final update to the output matrix **C** is performed on Line 17.

The parallelization presented in Line 5 of Algorithm 1 incurs a significant challenge, namely achieving a balanced workload between Lines 8 and 12. If the workload is imbalanced, those PEs assigned with fewer a_{mk} must remain idle until the PE handling the heaviest workload completes its computation. Line 10 also requires an accumulation phase. Due to the latency associated with addition operations on the DSPs, this results in a Read-After-Write hazard in Line 10. Tackling this challenge requires scheduling the a_{mk} values so that any two with the same row index m are issued with sufficient temporal distance to allow the DSP pipeline to complete the first addition operation before the second begins.

Algorithm 1: Tiled SpMM with cyclic row-wise partition: $\mathbf{C} \leftarrow \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}$

Input: Sparse matrix $\mathbf{A} \in \mathbb{R}^{M \times K}$, dense matrix $\mathbf{B} \in \mathbb{R}^{K \times N}$, scalars α, β
1 **Input/Output:** Matrix $\mathbf{C} \in \mathbb{R}^{M \times N}$
Output: Updated matrix **C**
2 **for** $m \leftarrow 0$ **to** M **by** M_0 **do**
3 **for** $n \leftarrow 0$ **to** N **by** N_0 **do**
4 Initialize temp buffer $Y[M_0][N_0] \leftarrow 0$;
5 **for** $p \leftarrow 0$ **to** $P-1$ **do** `// Parallel across PEs`
6 **for** $k \leftarrow 0$ **to** K **by** K_0 **do**
7 Load $B\_buf[K_0][N_0] \leftarrow B[k:k+K_0][n:n+N_0]$;
8 **foreach** $m \bmod P = p$ *and* $a_{mk} \in A$ *and* $a_{mk} \neq 0$ **do**
9 **for** $j \leftarrow 0$ **to** $N_0 - 1$ **do**
10 $Y[m \bmod M_0][j]$ += $a_{mk} \cdot B\_buf[k \bmod K_0][j]$;
11 **end**
12 **end**
13 **end**
14 **end**
15 **for** $i \leftarrow 0$ **to** $M_0 - 1$ **do**
16 **for** $j \leftarrow 0$ **to** $N_0 - 1$ **do**
17 $C[m+i][n+j] \leftarrow \alpha \cdot Y[i][j] + \beta \cdot C[m+i][n+j]$;
18 **end**
19 **end**
20 **end**
21 **end**

2.2 Related Work

Workload Imbalance. A central challenge in SpMM acceleration is achieving workload balance across processing elements (PEs). Sextans [15] addresses this issue through a PE-aware, out-of-order scheduling mechanism that enables a pipeline initiation interval

(II) of one cycle. While effective under certain sparsity conditions, its approach shows performance degradation when dense rows are present, resulting in idling PEs due to scheduling bubbles. Leda [20] approaches workload imbalance with out-of-order scheduling after a column-wise reordering in the granularity of 16×16 tiles. However, this technique becomes ineffective in the presence of highly dense rows, where the imbalance persists despite reordering. In addition, unlike Sextans and our design, Leda does not perform coefficient scaling and only performs $C = A \times B$.

DySpMM [17] proposes an element-wise dynamic allocation scheme that improves PE utilization by distributing nonzero elements on the fabric. DySpMM balances workload within each PE via an Allocator that distributes sparse elements into the PE's PU (Processing Units inside each PE) queues in a round-robin fashion. This mechanism balances load neither across different Allocators nor across PEs. The sparse matrix is read from memory in a packed form and then unpacked into parallel streams, each scheduled by an Allocator. Each Allocator maintains four queues mapped one-to-one to the four Processing Units (PUs) within a PE. Sparse elements are enqueued to these queues in a round-robin fashion to equalize work across PUs - an approach DySpMM calls element-wise allocation. Inter-PE distribution is instead set by the lightweight packing (e.g., fixed row-to-stream partitioning) together with the Sparse Distributor's static mapping patterns, which are aimed at parallel access rather than per-batch load equalization. Consequently, unless additional balancing were introduced at packing time, which would amount to host-side preprocessing that DySpMM explicitly avoids, inter-PE imbalance remains. To our knowledge, DySpMM therefore provides intra-PE but not inter-PE load balancing. Although it results in higher throughput, much of the reported performance improvement stems from end-to-end evaluation, including host-side pre-processing when comparing to other work. In contrast, our work introduces a dense-row handling mechanism that specifically targets workload imbalance caused by *dense rows* - rows that contain disproportionately more nonzero elements than others. Distributing these dense rows across all PEs (instead of mapping them onto a single PE) improves PE utilization across a diverse set of matrices. This case is often overlooked in prior SpMM designs.

Tight Coupling between Computation Units and HBM Channels. A limitation in existing designs is their tight coupling between computation units and HBM channels, particularly for loading and storing the output matrix **C**. In Sextans, Leda, and DySpMM, this architectural constraint limits scalability, as additional PEs necessitate more HBM channels. For example, Sextans requires an additional HBM channel to broadcast a pointer list of workloads among PEs, and its architecture is tightly coupled to HBM channels, limiting its scalability. Like Sextans, Leda and DySpMM are constrained by a tight coupling between its computational cores and the HBM channels assigned to the dense output matrix, restricting scalability. To address this, we propose an architecture in which PE scaling is decoupled from HBM channels for dense matrix C. This allows our architecture to scale up to 80 PEs on Alveo U280 FPGA without being limited by the number of HBM channels, thereby overcoming a major bottleneck in previous designs.

RAW Hazards in Accumulation. Another challenge is the management of RAW hazards in the accumulation phase of SpMM, particularly when pipelined floating-point adders are used. Sextans mitigates this issue by reordering nonzero elements to resolve dependency conflicts, but its solution introduces complexity due to cross-PE pointer management. Leda applies a minimum similarity approach in granular tiles that avoids stalling, assuming consistent latency in the pipeline. Although this approach minimizes RAW hazards, it is not effective when dealing with granular tiles containing highly dense rows. DySpMM employs a lightweight reorder unit that dynamically interleaves tasks to eliminate pipeline stalls at runtime. While this approach is efficient in avoiding RAW hazards, it is not clearly shown that this method can scale effectively with dense rows. Our proposed design effectively addresses RAW hazards by implementing fixed distance scheduling, thereby ensuring that such hazards are prevented from occurring.

Design Space Exploration and Automation. Lastly, current state-of-the-art designs lack design space exploration (DSE) that adapts to varying matrix characteristics or user-defined hardware constraints. In contrast, our work introduces a lightweight, automated DSE framework that tailors architectural parameters based on matrix property/structure and user constraints. The DSE is not matrix-dependent, i.e., not coupled to the raw matrix. It utilizes the structure from the matrix to find the best configuration. We show that the DSE results is reusable among various matrices. Also, we provide two generic designs that show superior performance to state-of-the-art studies.

Summary. Table 1 provides a comparative summary of key design attributes across several state-of-the-art SpMM accelerators—Sextans, Leda, and DySpMM—alongside our proposed design. The comparison focuses on several key dimensions: workload balancing strategy, computation and HBM channel coupling, RAW dependency handling, as well as maximum achieved number of processing elements (PEs) on the target FPGA platform (AMD/Xilinx Alveo U280 HBM-based FPGA in our paper). Sextans

and Leda both adopt row-level workload distribution with out-of-order (OoO) scheduling to balance computation across PEs. Their strategies for mitigating RAW hazards differ, with Sextans employing minimum distance scheduling and Leda applying a minimum similarity strategy. However, both are ultimately constrained by tight coupling between compute cores and HBM channels, limiting their scalability to 64 PEs in practical deployments. DySpMM takes a different approach by using on-chip data distribution and element-wise allocation for intra-PE balancing. It resolves RAW dependencies through an interleaved reordering mechanism. Nonetheless, it also suffers from scalability limitations due to fixed HBM coupling, capping its effective PE count at 64.

Table 1. Comparison of SpMM accelerator designs on Alveo U280 FPGA

Design	Workload Imbalance	PE-HBM Coupling	RAW Hazards in Accumulation	Max #PEs
Sextans [15]	OoO Scheduling	Tight Coupling	Min. Distance Scheduling	64
Leda [20]	OoO Scheduling	Tight Coupling	Min. Similarity Strategy	64
DySpMM [17]	On-Chip Data Distribution	Tight Coupling	Interleaved Reordering	64
HiSpMM (Ours)	Dense Row Sharing	Decoupled	Fixed Distance Scheduling	80

In contrast, our proposed design HiSpMM contains a dense-row sharing scheme during workload balancing, explicitly targeting the imbalance caused by the presence of rows with significantly more nonzero elements than others. By identifying and distributing such rows across multiple PEs, our design ensures balanced execution and higher overall utilization. Our design uses a deterministic reordering strategy to handle RAW dependencies through fixed-distance scheduling, simplifying control logic while preserving pipeline efficiency. Moreover, our architecture decouples compute units from HBM channels. Therefore, it allows great flexibility to scale the number of PEs or HBM channels. The only limitations to the scalability of the design are the on-chip resources and/or the routing congestion of the design. Finally, our design also supports the matrix-property driven design space exploration and automation that are not explored in prior studies.

3 HiSpMM Architecture Design and Implementation

This section begins with an introduction to the general overview of our design, which is subsequently followed by an in-depth explanation of each individual component in detail. We leave the details of our load balancing strategy to Section 4.

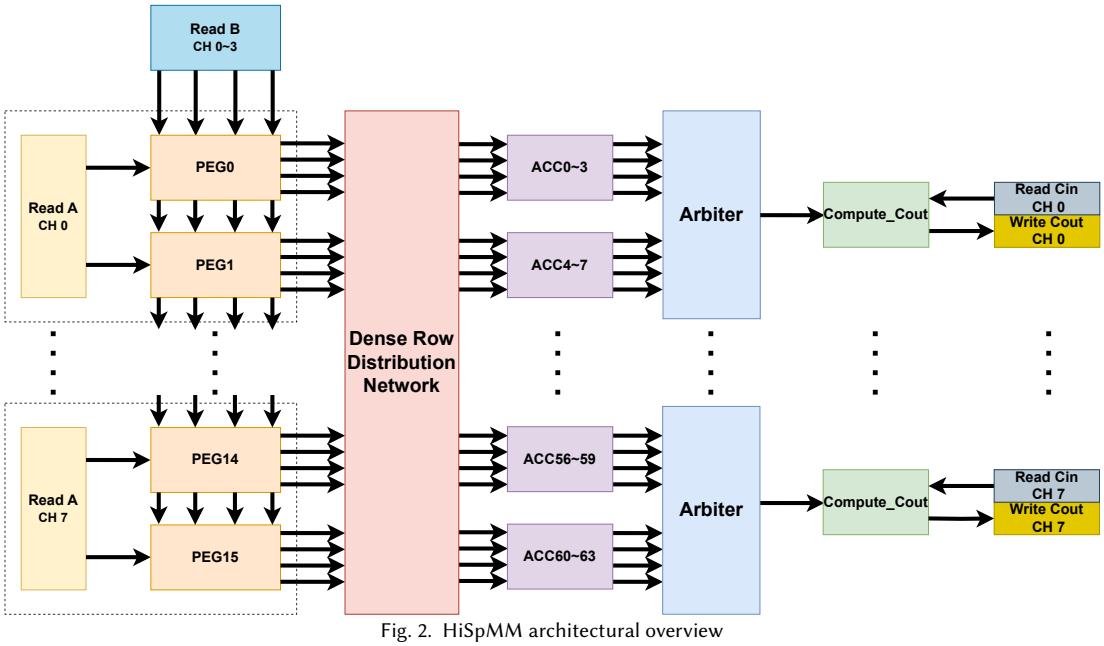
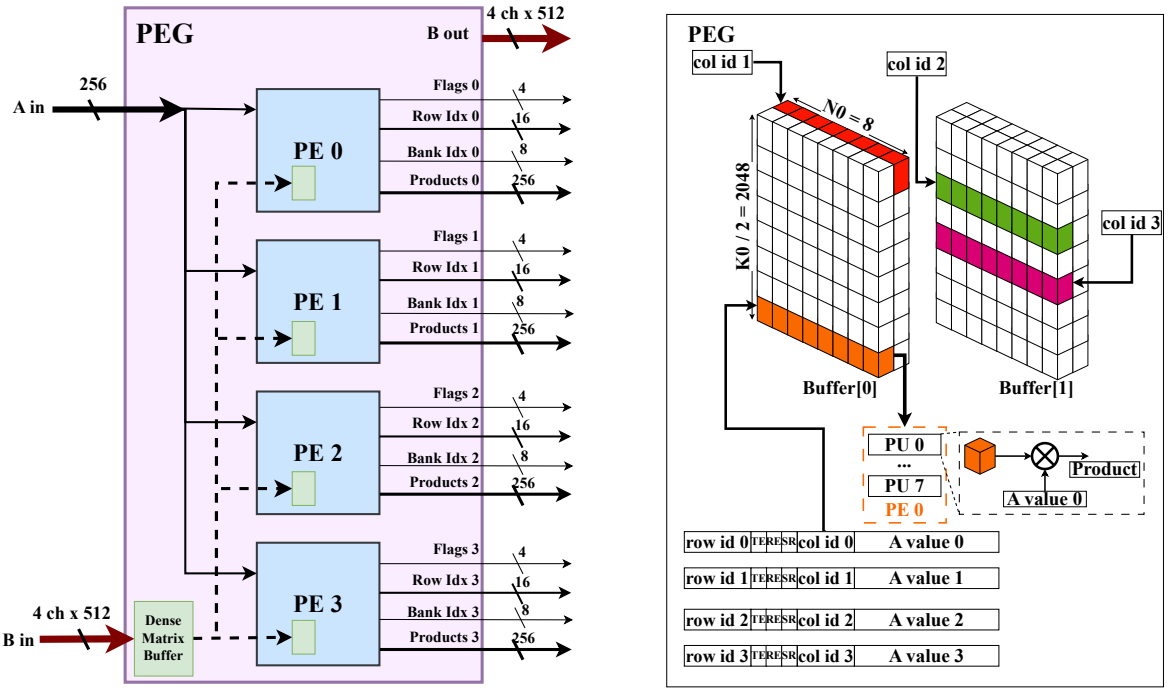


Fig. 2. HiSpMM architectural overview

3.1 Architectural Overview

In Figure 2, the architecture is depicted with a sample configuration of 16 processing element groups (PEGs), each comprising 4 PEs, which sums up to 64 PEs in total. Multiple channels stream the sparse matrix **A**, with each channel linked to 2 PEGs accounting

for 8 PEs. This configuration optimizes the usage of HBM channels with a port width of 512 bits [11]. Initially, PEG0 buffers the dense matrix **B** before transferring it to the next PEG. Matrix **A** is streamed with a 512-bits width and each element is 64-bit encoded entry. Therefore, two PEGs can produce 8 results of the **A**×**B** operation. Similar to Sextans [15], the dense matrix **B** is buffered via 4 HBM channels, given the dual-port BRAMs available on the FPGA. Following the PEGs, the Dense Row Distribution Network (DRDN) manages the sharing of dense rows within the sparse matrix **A**'s scheduled sequence. Using an identifier flag, the DRDN assesses if an entry is from inter-row or intra-row distribution. For inter-row entries, it acts as a direct connection to the output, while for intra-row entries, it performs addition through adder-tree structure and routes through switches to the appropriate Accumulator Unit in the subsequent stage. The Accumulator stage (**ACC**) aggregates the input data, storing the results on dual-port URAMs. Following accumulation, each unit's buffer empties into the Arbiter stage at a bitwidth of 256. The Arbiter combines 2 inputs into 512 bits and directs them to the unit in the next stage. To maximize HBM bandwidth usage, the dense **C** matrix is read and written with a bitwidth of 512. The **Compute_C** stage streams the initial dense **C** matrix, applies the final computations, and outputs the final dense **C** matrix.



(a) PE organization inside PEG (b) PEG data organization

Fig. 3. PEG architectural overview

3.2 PEG and PE Architecture

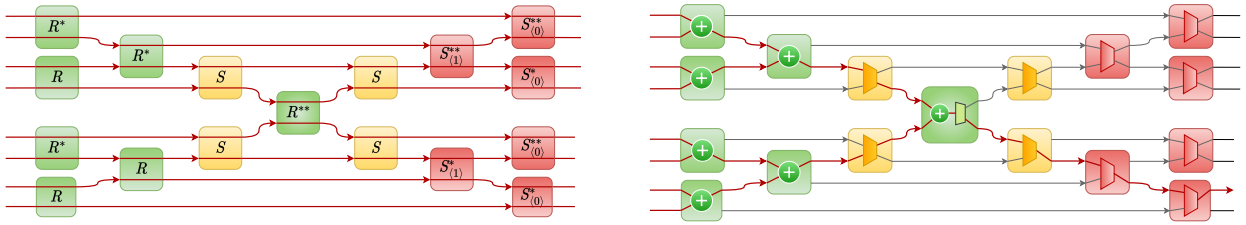
PEGs serve as a fundamental computational unit in our architecture, consisting multiplication and additional operations essential to the processing pipeline. As illustrated in Figure 3a, each PEG is composed of four processing elements (PEs). The sparse matrix **A** is streamed into each PEG as 256-bit words, where each word contains four 64-bit encoded entries. Simultaneously, each PEG buffers segments of the dense matrix **B** using a 512-bit-wide interface. This configuration enables each PE to independently perform a multiplication between a single nonzero entry from **A** and the corresponding segment of the buffered **B** matrix, ensuring high parallelism and throughput within the PEG.

As illustrated in Figure 3b, each nonzero entry is based on an encoded COO (COOrdinate) format which consists of 32-bit value, column index, row index, and three flag bits. Flag bits are `TileEnd (TE)`, `RowEnd (RE)`, and `SharedRow (SR)`. `TE` flag controls the termination of the multiplication and accumulation of a tile. `RE` and entry row index will construct the row index for the next stage. `SR` is being set in the scheduling phase and will be used in Dense Row Distribution Network. Figure 3b also illustrates how the dense matrix **B** is buffered within the PEG. For each nonzero element of the sparse matrix **A**, its column index is interpreted as the

row index used to access the buffered **B** matrix. Each PE contains eight processing units (PUs), and PUs perform a multiplication between the `a_value` and a corresponding row of buffered **B** values.

Apart from multiplication, PEG prepares the crucial information needed for the next stages of the design. PEGs construct the `Row index`, `flags`, and `Bank index`. The `Bank index` is determined via a standard way to map rows to banks/PEs in a round-robin fashion.

3.3 Dense Row Distribution Network (DRDN)



(a) Inter-row mode. Eight packets from eight distinct rows are propagated to the downstream blocks.

(b) Intra-row mode. Eight packets from one row are reduced and routed to the corresponding accumulator.

Fig. 4. An example 8 × 8 dense row distribution network (DRDN). R, R^* are pairwise vector addition blocks; S, $S^*_{\langle k \rangle}$, $S^{**}_{\langle k \rangle}$ are packet switch modules. R^{**} is a hybrid addition-switch block. The DRDN is between the upstream PEGs and downstream Accumulators.

The Dense Row Distribution Network (DRDN), as shown in Figure 4, addresses the complexities introduced by dense row sharing during the scheduling phase inspired from [14]. The inputs to the DRDN consists of all output streams from all the PEs. Internally, the DRDN is composed of two main components: an adder-tree stage and a routing stage. Its behavior is governed by two control fields embedded in each entry: the `SharedRow (SR)` flag and the `Bank Index`.

When the `SR` flag is set, indicating that the corresponding row is shared across multiple PEs, the entry is directed through the adder-tree stage, where it is combined with other partial sums belonging to the same row. The aggregated result is then routed to the appropriate accumulator based on its `Bank Index`. On the other hand, if the `SR` flag is not set, the entry bypasses the adder-tree and is routed directly to its designated accumulator without modification. This mechanism enables the architecture to handle both shared and unshared rows efficiently, maintaining correctness.

DRDN is a deeply pipelined, hierarchical task network that performs distributed accumulation and dynamic routing of partial results across multiple stages. Data enters the network from multiple parallel upstream producers, i.e. PEGs, and flows toward a set of accumulator modules. Each stage of the network is designed to either aggregate, re-route, or restructure the data based on contextual metadata, i.e. bank index, and flags.

Pairwise Vector Reduction (PVR). Three functionally distinct types of arithmetic modules are deployed throughout the pipeline to perform pairwise vector summation of data packets. Figure 5 depicts the internal schematic of the PVR.

- *The first type of vector addition module (R)* performs a straightforward element-wise addition of two input vectors. When a control flag depending on `SharedRow` flag of first input ($input_0.sharedrow$) and non-dummy status of both inputs is active, only first output ($output_0$) carries the result of the addition while the other output transmits a zero vector with a dummy flag ($output_1.dummy = true$). Otherwise, both inputs are passed through transparently without any addition.
- *The second type of vector addition module (R^*)* mirrors the logic of R, but swaps the roles of the output streams. While functionally similar, this type provides complementary packet routing behavior to enable symmetry and balanced latency in the reduction tree.
- *A third, hybrid addition-switch module (R^{**})* performs element-wise addition of two input vectors, but a dynamic output selection based on a different control flag depending on the $log_2(num_{pes} - 1)$-th bit of bank index of first input, `SharedRow` flag of first input, and non-dummy status of both inputs. This logic allows the reduction tree to forward the summed value to different outputs. For each reduced value produced and routed by R^{**}, its other output carries a zero vector with a dummy flag.

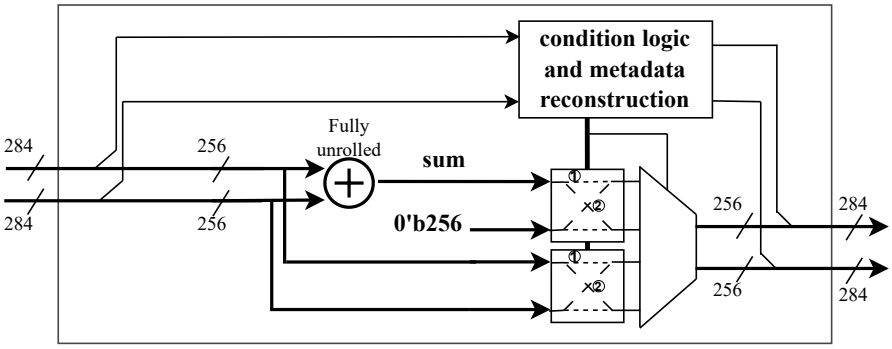


Fig. 5. Pairwise Vector Reduction units, type ① (R) and ② (R^*).

Stream-Switching Modules (SSM). Between pairwise vector addition modules (R, R^*, R^{**}) and after hybrid addition-switch (R^{**}), a network of switch-like tasks determines how packets are routed downstream. These modules operate entirely on packet metadata. Two categories are present:

- *Simple shared-row-aware switches (S)* inspect the `SharedRow` flag of first input packet. Based on this, both packets are either forwarded directly or switched for the target outputs.
- *Bank-index-aware switches* ($S^*_{\langle k \rangle}$ and $S^{**}_{\langle k \rangle}$)): $S^*_{\langle k \rangle}$ inspects the `SharedRow` flag and the k-th bit of the `BankIndex` of the first input packet. The switching is based on the logical AND of these two bits. $S^{**}_{\langle k \rangle}$ inspects the `SharedRow` flag and the k-th of the `BankIndex` of the second input packet. The switching is based on the logical NOR of these two bits. In summary, there are $(2 \times log_2(num_{pes}) + 1)$ stages in the DRDN pipeline. Switches with a higher value of k where $k \in \{0, log_2(num_{pes}) - 2\}$ are instantiated earlier in the pipeline (i.e., $S^*_{\langle 3 \rangle}$, then $S^*_{\langle 2 \rangle}$, then $S^*_{\langle 1 \rangle}$, then $S^*_{\langle 0 \rangle}$). By cascading such switches across multiple levels, each focusing on a different bit position, the design effectively organizes the data into binary subgroups at each stage. Such a fabric of switches ensures data correctness by routing partial sums to their corresponding downstream accumulators.

Final Routing and Output Alignment. At the terminal stage of the network, $S^*_{\langle 0 \rangle}$ and $S^{**}_{\langle 0 \rangle}$ also convert the routed packet to a format compatible with the downstream accumulators. The routing logic ensures that each pair of partial results reaches the correct accumulation unit. The DRDN design is fully pipelined. So, it contributes only a one-time latency (i.e., pipeline depth) to the total execution latency, which is 66 cycles for the 48-PE design and 72 cycles for the 64-PE design.

3.4 Decoupling Computation from HBM Channels

Efficient utilization of HBM channels is critical for achieving high throughput in FPGA-based SpMM designs. The number of HBM channels available for streaming the sparse matrix A is determined by the allocation of PEs per channel. For instance, to preserve efficient memory access with a width of 512 bits, it is standard practice for each HBM channel to be utilized by 8 PEs since each encoded A element is 64 bits. As a result, it needs to access 8 columns of dense matrix B every cycle, which can be buffered in 4 dual-port BRAM blocks. This results in a typical 4 HBM channels dedicated for dense matrix B. Prior architectures often tie each computational core (i.e., a pair of two PEGs in our case) directly to dedicated HBM channels for reading and writing the output matrix C_{out} as well, with each core using one channel for reading and one channel for writing. This configuration typically supports up to 8 cores on platforms like AMD/Xilinx Alveo U280, consuming 16 out of 32 available HBM channels solely for output matrix C. Scaling up such architectures further limits the remaining available channels for matrix **A**.

To overcome these limitations, we decouple the handling of C_{out} from the computational cores (i.e., PEGs and Accumulators) via a dedicated arbitration stage. This allows matrix **A** to leverage a greater number of HBM channels while maintaining a smaller number of units for computing $\beta \cdot C_{\text{out}}$, each using 2 channels. This reallocation improves overall memory bandwidth available for computing $A \cdot B$ while marginally sacrificing throughput in the final output write-back stage. Moreover, it also allows to increase the number of HBM channels for dense matrix C if needed. The arbitration stage provides architectural flexibility by disassociating the number of PEs from number of C channels. This way, the design scalability is only limited by on-chip resource utilization and routing congestion in practice.

We implement an arbiter between the accumulation stage and the compute units responsible for applying $\beta \times C_{\text{out}}$. On the input side of the arbiter stage, there are multiple 256-bit streams—one from each Accumulator. The arbiter aggregates these streams and produces 512-bit output streams, corresponding to the number of HBM channels allocated for the dense matrix C. Two variants of the arbiter are considered: 1) Monolithic Arbiter: A centralized unit connecting all accumulation outputs to compute units. While simple, it introduces routing congestion, especially in large designs (e.g., 64 PEs × 256-bit outputs = 16,384 wires), and can exceed SLR (Super Logic Region) boundary limits in the U280 fabric. 2) Non-monolithic Arbiter: A modular, distributed arbitration design that reduces congestion by enabling better floorplanning. This version is preferred for large-scale configurations. For a 64-PE design with 8 C channels, we use eight 8-to-1 arbiter modules; and for a 64-PE design with 4 C channels, we use four 16-to-1 arbiter modules, keeping the resource utilization for the interconnection network of both designs quite similar.

The use of any types of arbiter requires its own C_{in} pattern with respect to the number of PEs in the design and number of HBM channels for dense **C** matrix. The selection of arbiter type and its requirement are included in our automation tool as well.

4 Load Balancing Strategy for Sparse Matrix Multiplication

In this section, we present a formulation and offer a thorough rationale for the proposed balancing strategy. Subsequently, we provide detailed explanation of the specifics and underlying principles of the balancing strategy.

4.1 Problem Formulation and Motivation

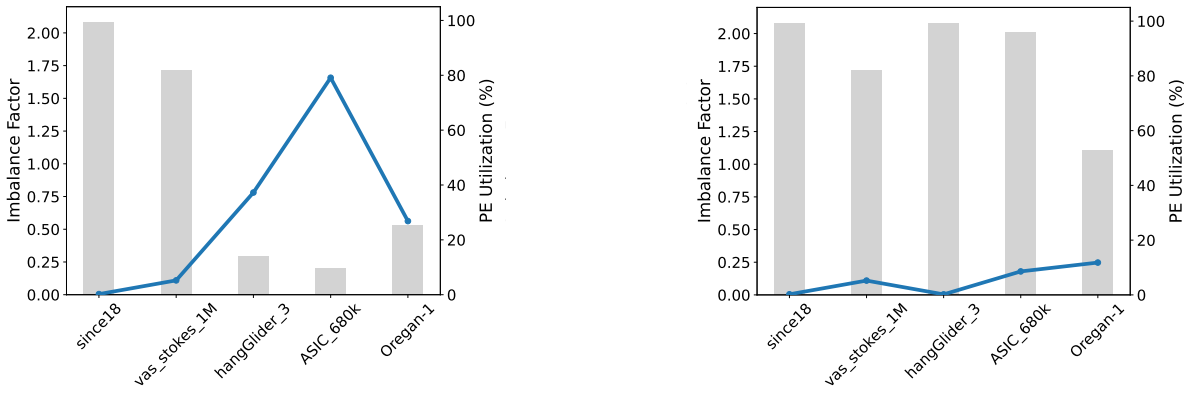
A primary challenge in scheduling the nonzero elements of a sparse matrix is achieving balanced workload distribution across all processing elements (PEs). If the workload is unevenly distributed, PE utilization can degrade significantly. In such scenarios, the most heavily loaded PE determines the overall execution time for a given phase, as all other PEs—despite completing their smaller workloads earlier—must be idle until the busiest PE finishes before progressing to the next task set. This leads to inefficient resource utilization and under-performance of the parallel architecture. This imbalance is a direct consequence of the inherently irregular memory access patterns associated with sparse matrices. Since nonzero elements are distributed non-uniformly, it is common for some rows or blocks of the matrix to contain significantly more nonzero entries than others. Without careful scheduling and load balancing strategies, this skew can cause substantial variation in computational demand across PEs, making workload imbalance an observable and recurring issue in sparse matrix processing.

Equation (1) shows the number of cycles for the computation of SpMM, where the multiplication of each tile of $A(i,k)$ and $B(k,j)$ is bounded by the maximum PE workloads $\max(\text{PE loads})_{ik}$ that is determined by the tile of sparse matrix $A(i,k)$ in our streaming design. The cycle count for computation can be approximately re-formulated as the RHS (right hand side) of the Equation (1) further, where, NNZ is the number of nonzeros in sparse matrix A, N and $N0$ are the number of columns of dense matrix and the number of columns in a tile of the dense matrix B, respectively, σ_{load} is the standard deviation of the loads assigned to the PEs, μ_{load} is the average load across all PEs.

$$\text{Cycle}_{\text{comp}} = \sum_{i=1}^{\lceil \frac{M}{M_0} \rceil} \sum_{j=1}^{\lceil \frac{N}{N_0} \rceil} \sum_{k=1}^{\lceil \frac{K}{K_0} \rceil} Cycle\_tile(i,j,k) = \left(\sum_{i=1}^{\lceil \frac{M}{M0} \rceil} \sum_{k=1}^{\lceil \frac{K}{K0} \rceil} \max(\text{PE loads})_{ik} \right) \cdot \left(\frac{N}{N0} \right) \approx \frac{NNZ}{\#PEs} \cdot \left(1 + \frac{\sigma_{\text{load}}}{\mu_{\text{load}}} \right) \cdot \left(\frac{N}{N0} \right) \quad (1)$$

To address this primary challenge, our solution focuses on minimizing the imbalance factor, defined as the ratio of the standard deviation of the workload across PEs to the mean workload, i.e., $\delta = \frac{\sigma_{\text{load}}}{\mu_{\text{load}}}$. This metric quantifies the degree of load imbalance (δ), with lower values indicating a more uniform distribution of workloads.

In an ideal scenario, the imbalance factor would approach zero, meaning all PEs are assigned exactly equal workloads, resulting in optimal resource utilization and maximum parallel efficiency. However, achieving such uniformity is particularly difficult in the context of sparse matrices. These irregularities often lead to significant variance in PE workloads, making load balancing a nontrivial task. Therefore, our approach aims to bring the imbalance factor as close to zero as possible through strategic workload partitioning and scheduling techniques tailored to the matrix's sparsity pattern. Figure 6a illustrates the role of imbalance ratio δ in PE utilization. For matrices with significant workload imbalance, the PE utilization is extremely low. For instance, the imbalance ratio for ASIC_680k matrix is 1.75 and leads to less than 10% percent PE utilization.



(a) Before applying Dense Row Sharing.

(b) After applying Dense Row Sharing.

Fig. 6. Effectiveness of the Dense Row Sharing method. The line plot shows the imbalance factor. The bar plot shows PE utilization.

Algorithm 2: Shared row selection for load balancing

Require: $tiledMatrices[i][j]$, $M0$, $\#PEs$
Ensure: $sharedRows[i][j]$, $numSharedRows[i][j]$

$nnzCounts[M0][2] \leftarrow \emptyset$ {Row index and nonzero count}
$peWorkloads[\#PEs][2] \leftarrow \emptyset$ {PE index and workload}
$CandidateRows \leftarrow \emptyset$
for $r = 0$ **to** $M0 - 1$ **do**
  $nnzCounts[r][0] \leftarrow r$
  $nnzCounts[r][1] \leftarrow$ count nonzeros in row r
  $pe \leftarrow r \ \% \ \#PEs$
  $peWorkloads[pe][0] \leftarrow pe$
  $peWorkloads[pe][1] \leftarrow peWorkloads[pe][1] + nnzCounts[r][1]$
end for
Sort $nnzCounts$ by $nnzCounts[r][1]$ in descending order
Compute initial δ_init
for $r = 0$ **to** $(M0 \ / \ 2) - 1$ **do**
  $newTotal \leftarrow total - nnzCounts[r][1]$
  $newPeWorkloads \leftarrow peWorkloads$
  $targetPE \leftarrow nnzCounts[r][0] \ \% \ \#PEs$
  $newPeWorkloads[targetPE][1] \leftarrow newPeWorkloads[targetPE][1] - nnzCounts[r][1]$
  Compute new δ_new
  if $(\delta\_new < \delta\_init)$ **then**
    $total \leftarrow newTotal$
    $peWorkloads \leftarrow newPeWorkloads$
    $CandidateRows.append(nnzCounts[r][0])$
  end if
  $\delta\_init \leftarrow \delta\_new$
end for
Sort $CandidateRows$ in ascending order
for $k = 0$ **to** $|CandidateRows| - 1$ **do**
  $sharedRows[i][j][k] \leftarrow CandidateRows[k]$
end for
$numSharedRows[i][j] \leftarrow |CandidateRows|$

4.2 Dense Row Sharing Strategy

To achieve a more balanced workload distribution among the PEs, we introduce a strategy called Dense Row Sharing in the pre-processing stage. This approach targets dense rows, which could lead to increased load imbalance if assigned to a single PE.

Dense Row Sharing works by distributing computationally expensive rows' nonzero elements across multiple PEs. By doing so, the excessive workload associated with a dense row is no longer concentrated on a single PE but is instead shared among all PEs, thereby reducing the variance in workload across the design. This distribution enhances PE utilization and minimizes idle time, ultimately leading to improved overall performance of the SpMM kernel.

Algorithm 2 outlines the procedure for selecting shared rows within each matrix tile to improve workload balance across PEs. The algorithm takes as input a tiled sparse matrix, the number of rows per tile (M_0), and the total number of PEs ($\#PEs$). The output is a set of rows designated for sharing ($sharedRows[i][j]$) along with the total number of shared rows per tile ($numSharedRows[i][j]$).

The process begins by computing the number of nonzeros in each row of the tile. For each row, the algorithm records both the row index and its number of nonzeros in Lines 5 and 6. Simultaneously, the initial workload assigned to each PE is accumulated in the `peWorkloads` array, using a modulo-based mapping. Once workloads are collected, the algorithm sorts the rows in descending order based on their number of nonzeros, prioritizing denser rows for possible sharing. The initial load imbalance factor (δ_{init}) is then computed based on the current distribution across PEs. The main loop iterates over the top half of sorted rows (those most likely to cause imbalance), evaluating each as a candidate for sharing. For each candidate row, the algorithm simulates removing its workload from its initially assigned PE and computes a new imbalance factor (δ_{new}). If this new imbalance is lower than the previous, the row is accepted as a shared row and the updated workload and imbalance values are stored. After evaluating all candidate rows, the final list of shared rows is sorted in ascending row index order for scheduling. The total number of selected shared rows is recorded for downstream processing stages. This adaptive and incremental approach ensures that only rows contributing to significant imbalance are selected for sharing, minimizing overhead while maximizing PE workload uniformity.

Algorithm 3: Scheduling mechanism

Require: Tiled matrices, *sharedRows[i][j]*, tile sizes
Ensure: Encoded matrix `fpgaAmtx`

1: **for** each tile (i, j) in the matrix **do**
2: Initialize PE load matrix: `Loads[PE][DIST]` ← 0
3: Split rows into:
- *sharedRows[i][j]*: rows to be distributed across PEs
- Remaining rows: assigned to fixed PEs using modulo mapping (e.g., PE ← row_id % #PEs)

4: /* **Schedule shared rows first** */
5: Sort shared rows by descending nonzero count
6: **for** each shared row **do**
7: Slice row across all PEs in a round-robin fashion
8: **for** each PE **do**
9: Select least-loaded DIST slot from `Loads[PE]`
10: Encode and store row slice to `fpgaAmtx`
11: Increment `Loads[PE][DIST]`
12: **end for**
13: **end for**
14: /* **Schedule remaining rows** */
15: Sort remaining rows by size
16: **for** each remaining row **do**
17: Assign to PE ← row_id % #PEs
18: Select least-loaded DIST slot for the assigned PE
19: Encode row to `fpgaAmtx` and update `Loads`
20: **end for**
21: **end for**
22: **return** `fpgaAmtx`

4.3 Scheduling Scheme

Algorithm 3 describes the scheduling mechanism for efficient FPGA execution. The input consists of tiled sparse sub-matrices, tile size information and the set of identified shared rows for each tile (*sharedRows[i][j]*) from Algorithm 2. The output is the encoded sparse matrix (`fpgaAmtx`), which contains the scheduled and formatted data ready for streaming into the accelerator.

For each tile (i, j), the algorithm initializes a load tracking matrix (*Loads[PE][DIST]*) that keeps track of the number of nonzeros assigned to each distance (DIST) slot per PE. This helps maintain a controlled dependency distance between consecutive accumulation operations, removing pipeline stalls due to RAW hazards.

The scheduling process consists of two main phases. First, the algorithm schedules the shared rows identified during the load balancing phase. These rows typically contain a high number of nonzero elements and are sliced across all PEs in a round-robin fashion to prevent load concentration. For each slice of a shared row, the algorithm selects the DIST slot with the minimum current

load for the target PE to ensure even distribution of work and to respect RAW dependency constraint. Second, the remaining non-shared rows are scheduled. These rows are assigned to specific PEs using a simple modulo-based mapping. For each assigned row, the algorithm again chooses the least-loaded DIST slot within the target PE to maintain a balanced workload. This two-phase scheduling approach ensures that load imbalance is minimized. The final output, `fpgaAmtx`, contains the fully encoded and dependency-aware sparse matrix representation suitable for high-throughput and hazard-free FPGA execution.

Through this adaptive and load-aware scheduling strategy, the algorithm significantly improves the uniformity of workload distribution across PEs, reduces idle time, and increases the overall computational throughput of the SpMM kernel on FPGA platforms. Figure 6b illustrates the effectiveness of the Dense Row Sharing method. For matrices where the workload distribution is already balanced, the PE utilization is already high. Moreover, for matrices with significant workload imbalance, the method maintains high PE utilization by substantially reducing the imbalance factor. Our lightweight one-time preprocessing can be well applied into batch processing scenarios. Moreover, many applications call the SpMM kernel iteratively, often without changing the sparse matrix properties, which also makes our one-time preprocessing overhead well amortized and negligible.

5 Automation Tool

Figure 7 illustrates the end-to-end automation flow of our HiSpMM design framework. The automation tool, highlighted within the red boundary, serves as the central component that bridges user input, design space exploration, and automatic code generation for both host and FPGA kernel codes. The flow begins with User Input, where users specify two main categories: FPGA hardware details (including the number of BRAMs, URAMs, DSPs, LUTs, FFs, and HBM channels) and the input sparse matrix. This information is forwarded to the Design Space Exploration (DSE) module.

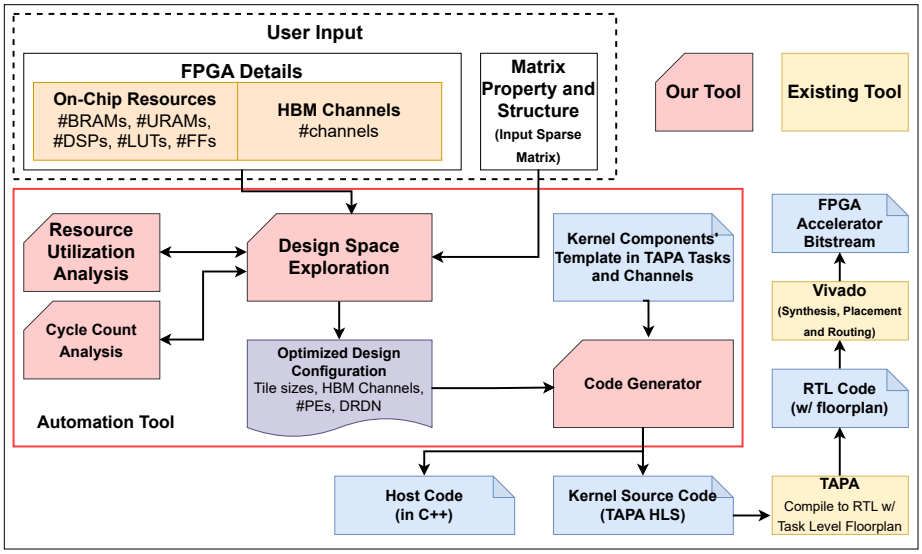


Fig. 7. HiSpMM design exploration and automation flow

Our DSE is *matrix structure/property driven*: i.e., which applies to a group of matrices that have the same structure or properties, instead of specific to each individual matrix. Guided by Equation (4) and Algorithm 4, detailed in Section 5.1 and Section 5.3, our DSE operates on a lightweight matrix profile—including matrix dimensions M, K, N, the sparsity ρ and imbalance ratio δ of matrix A, together with hardware constraints, including number of HBM channels, DSP, LUT, FF, BRAM, and URAM budgets. Then our DSE tool decides the number of HBM channels allocated to each matrix A, B, and C, and sets the number of PEs as $\#PEs = A\_CH \times \#PE_{A\_CH}$, where A_CH is the number of HBM channels allocated to matrix A and $\#PE_{A\_CH}$ is the number of PEs driven per A channel. It also chooses proper tiling sizes M_0^{eff} and K_0^{eff} and decides whether to enable DRDN (if it improves the imbalance ratio: $\delta_{\text{improv}} < 0.75\,\delta_{\text{orig}}$). The goal is to minimize the cycle count predicted by Eq. (4). Note the search is matrix property-driven, and it does not depend on the input matrix values. In practice, a chosen best configuration often serves many matrices without per-matrix reconfiguration.

Following the DSE phase, the Code Generator module uses this optimized configuration to generate two key outputs: (1) Host Code written in C++ and (2) Kernel Source Code written in TAPA HLS [3], which enables task-parallel high-level synthesis (HLS) and explores coarse-grained floorplanning to improve the timing closure. The code generator uses our pre-defined template of kernel components structured as TAPA tasks and channels, ensuring modularity across different configurations. The generated kernel code is then processed using the TAPA framework, which calls Vitis HLS to compile the code into RTL code with coarse-grained floorplanning constraints provided by TAPA. The resulting RTL design undergoes synthesis, placement, and routing using Vivado to produce the final FPGA bitstream.

In summary, the automation tool significantly reduces the manual design effort by automating the matrix property-specific exploration, workload-aware kernel customization, and host-kernel interface generation.

5.1 Cycle Count Analysis

In our automation tool, DSE has two objectives: 1) find the best generic design configurations for balanced and imbalanced matrices under user constraints, and 2) find the best matrix property-based design configuration under user constraints.

In a tiled SpMM with cyclic row-wise partitioning, the total cycle count consists of three principal components. As shown in Equation (2), these correspond to the buffering (i.e., loading) of the dense matrix **B**, the compute determined by max(PE_Load), and the streaming (i.e., storing) of the result dense matrix **C**.

$$Cycle\ Count = ([(Cycle\_Stream\_BTile + \max(PE\_Load)) \cdot KTiles + Cycle\_Stream\_CTile] \cdot NTiles) \cdot MTiles \tag{2}$$

To fully utilize the HBM bandwidth, we employ 512-bit port width for each channel. That is, each cycle it reads/writes 16×32-bit elements. Based on the tile dimensions illustrated in Figure 1, the cycle count expression can be expanded to Equation (3). To account for the imbalance in the distribution of sparse entries across processing elements, the compute term is further generalized, leading to the final expression in Equation (4). In this final form, multiple parameters are influenced by the number of channels allocated to streaming the sparse **A** matrix (denoted as A_CH). These include the tile height M_0, the load imbalance ratio $\delta = \frac{\sigma_{\text{load}}}{\mu_{\text{load}}}$, and the total number of PEs. Note the number of HBM channels allocated for buffering the dense **B** matrix is 4, similar to Sextans [15]. The DSE will find the minimum cycle count in Equation (4).

$$Cycle\ Count = \left(\frac{K0 \cdot N}{B\_CH \cdot 16} \cdot \left\lceil \frac{K}{K0} \right\rceil \cdot \left\lceil \frac{M}{M0} \right\rceil\right) + \left(\left(\sum_{i=1}^{\lceil \frac{M}{M0} \rceil} \sum_{k=1}^{\lceil \frac{K}{K0} \rceil} \max(\text{PE\_loads})_{ik}\right) \cdot \left(\frac{N}{N0}\right)\right) + \left(\frac{M0 \cdot N}{C\_CH \cdot 16} \cdot \left\lceil \frac{M}{M0} \right\rceil\right) \tag{3}$$

$$Cycle\ Count = \left(\frac{K_0 \cdot N}{B\_CH \cdot 16} \cdot \left\lceil \frac{K}{K_0} \right\rceil \cdot \left\lceil \frac{M}{M_0} \right\rceil\right) + \left(\frac{M \cdot K \cdot \rho}{A\_CH \cdot \#\text{PEs}_{\text{A\_CH}}} \cdot \left(\frac{N}{N_0}\right) \cdot \left(1 + \frac{\sigma_{\text{load}}}{\mu_{\text{load}}}\right)\right) + \left(\frac{M_0 \cdot N}{C\_CH \cdot 16} \cdot \left\lceil \frac{M}{M_0} \right\rceil\right) \tag{4}$$

5.2 User-Defined Resource Constraints

The total count of PEs is calculated by the product of A_CH and PEs_{A_CH}. Each PE carries out $N_0 = 8$ floating-point multiplications, and as each multiplication requires 3 DSPs, the DSP demand per PE amounts to $3 \cdot N_0$, or equivalently, 24. A buffer accommodating dense matrix B is available to each PE and utilizes 8×BRAM18K blocks per B channel. A Processing Element Group (PEG), made up of 4 PEs, thereby necessitates 32 BRAM18K blocks for each B channel. In addition, every PE needs an accumulator, which buffers partial results from matrix **C**. These partial outcomes are stored in URAMs, with each accumulator needing 8 URAM blocks.

Table 2 provides a detailed breakdown of how resources are consumed by each task in our design. Additionally, it displays the count of individual tasks in a configuration featuring 64 PEs, requiring 8 A channels ($A\_CH = 8$), along with 4 B channels ($B\_CH = 4$), and 8 C channels ($C\_CH = 8$). In Equation (5), $\mathcal{H}$ is the hardware configuration model. It considers the on-chip resource utilization ($\mathcal{U}$) and number of HBM channels ($\mathcal{N}_{\mathcal{HBM}}$). In total, the BRAM18K usage amounts to $64 \cdot A\_CH \cdot B\_CH$, while the URAM usage is $64 \cdot A\_CH$, and the DSP consumption estimates to $448 \cdot A\_CH + 128 \cdot C\_CH$. The user available resources in the dynamic region of the Alveo U280 FPGA are 3,504 BRAM18K blocks, 8,496 DSPs, 2.33 million FFs, 1.16 million LUTs, and 960 URAM blocks.

$$\mathcal{H} = \Phi(\mathcal{U},\ N_{\text{HBM}}), \quad \mathcal{U} = (u_{\text{BRAM}},\ u_{\text{URAM}},\ u_{\text{DSP}},\ u_{\text{LUT}},\ u_{\text{FF}}), \quad N_{\text{HBM}} = A\_\text{CH} + B\_\text{CH} + 2 \cdot C\_\text{CH}. \tag{5}$$

Our design space exploration (DSE) framework accepts user-defined constraints such as limits on BRAM, URAM, DSP, LUT, and FF utilization, or the number of available HBM channels. These constraints are incorporated into the hardware configuration

Table 2. Resource utilization per task in HiSpMM

Task	BRAM18K	DSP	FF (k)	LUT (k)	URAM	# Task (in 64-PE)
Stream_A	0	1	7.0	6.8	0	8
Load_B	0	2	7.5	7.0	0	4
Stream_Cin	0	2	7.5	7.0	0	8
Stream_Cout	0	2	7.5	7.6	0	8
Accumulator	0	16	3.0	3.0	8	64
Compute_C	0	128	9.5	7.8	0	8
PEG	128	96	5.3	8.3	0	16
SSM_simple	0	0	0.6	1.21	0	60
SSM_par	0	0	0.6	1.5	0	62
PVR	0	16	3.1	3.4	0	63
Arbiter	0	0	1.65	3.36	0	8

model, which determines the valid set of (A_CH, B_CH, C_CH) values used on Lines 6 and 13 in Algorithm 4. For example, if the BRAM and URAM usage limits are set to 20%, the resulting configuration space $\mathcal{H}$ may include combinations such as $(A\_CH, B\_CH) = \{(3,3),\ (2,4)\}$, with corresponding $C\_CH = \{12,\ 8\}$. Our DSE then selects the matrix property specific optimal configuration based on the estimated cycle count.

Algorithm 4: Design space exploration in HiSpMM

Require: Matrix set $\mathcal{M}$, hardware configurations $\mathcal{H}(A\_CH, B\_CH, C\_CH)$
Ensure: Optimal configuration set $\mathcal{R}$
1: Initialize result set $\mathcal{R} \leftarrow \emptyset$
2: Set constants: $K_0 \leftarrow 4096$, $N_0 \leftarrow 8$, $B\_CH \leftarrow 4$, $PEs_{A\_CH} \leftarrow 8$
3: **for** each matrix $M_i \in \mathcal{M}$ **do**
4: Extract (M, K), compute $\rho = \frac{NNZ}{M \cdot K}$
5: Initialize candidate set $C_i \leftarrow \emptyset$
6: **for** each $A\_CH \in \mathcal{H}$ **do**
7: $\#PEs \leftarrow A\_CH \cdot PEs_{A\_CH}$
8: $M_0 \leftarrow \#PEs \cdot 8192$
9: $M_0^{\text{eff}} \leftarrow \min(M, M_0), \quad K_0^{\text{eff}} \leftarrow \min(K, K_0)$
10: $(\delta_{\text{orig}}, \delta_{\text{impr}}) \leftarrow \text{GetImbalanceRatios}(M_i, A\_CH)$
11: improv. $\leftarrow \frac{(\delta_{\text{orig}} - \delta_{\text{impr}}) \cdot 100}{1 + \delta_{\text{orig}}}$
12: $\delta \leftarrow (\text{improv.} > 25\%)\ ?\ \delta_{\text{impr}} : \delta_{\text{orig}}$
13: **for** each $C\_CH \in \mathcal{H}$ **do**
14: $T_1 \leftarrow \frac{K_0^{\text{eff}} \cdot N}{B\_CH \cdot 16} \cdot \lceil K/K_0 \rceil \cdot \lceil M/M_0 \rceil$
15: $T_2 \leftarrow \frac{M \cdot K \cdot \rho}{\#PEs} \cdot \frac{N}{N_0} \cdot (1 + \delta)$
16: $T_3 \leftarrow \frac{M_0^{\text{eff}} \cdot N}{C\_CH \cdot 16} \cdot \lceil M/M_0 \rceil$
17: cycles $\leftarrow T_1 + T_2 + T_3$
18: Add $\{A\_CH, C\_CH, \#PEs, \text{cycles}, M_0^{\text{eff}}, K_0^{\text{eff}}, \delta, M_i\}$ to C_i
19: **end for**
20: **end for**
21: Add $\arg\min_{\text{config} \in C_i}$ config.cycles to $\mathcal{R}$
22: **end for**
23: **return** $\mathcal{R}$

5.3 SpMM Hardware Configuration via DSE

Algorithm 4 describes the design space exploration (DSE) process for selecting the optimal hardware configuration for accelerating sparse matrix multiplication. The algorithm takes as input a set of matrices $\mathcal{M}$ and a set of hardware configuration options $\mathcal{H}$, and outputs the optimal configuration for each matrix based on estimated cycle counts.

The process begins by initializing an empty result set $\mathcal{R}$. Fixed hardware parameters are defined, including the initial tile width K_0 and N_0, the number of B matrix channels B_CH, and the number of processing elements per channel PEs_{A_CH}. For each matrix M_i in the dataset, the algorithm reads its dimensions (M, K) and calculates the density ρ based on the number of nonzero elements.

For each possible value of *A_CH*, representing the number of channels for streaming the sparse matrix **A**, the total number of PEs and effective tile height M_0 are computed. We also calculate M_0^{eff} and K_0^{eff} in case the matrix size is smaller than the tile size. Then imbalance in the PE load distribution is calculated, including the original and improved imbalance factors, δ_{orig} and δ_{impr}. If the improvement exceeds a threshold (25%) then the improved δ_{impr} is used; otherwise, the original δ_{orig} is retained. Next, based on resource constraints, valid values of *C_CH* (number of output channels for matrix **C**) are fetched.

The total estimated cycle count is computed as the sum of T_1, T_2, and T_3. Each evaluated configuration, along with its parameters and metrics, is added to a set C_i specific to the matrix M_i. After exploring all configurations for a given matrix, the algorithm selects the one with the minimum estimated cycle count. This best configuration is added to the overall result set $\mathcal{R}$.

6 Experiments and Results

In this section, we first present the experimental setup in Section 6.1 and the design configurations and resource utilization of generic SpMM hardware accelerators, including two of our proposed generic designs (HiSpMM-balanced for balanced matrices and HiSpMM-imbalanced for imbalanced matrices) and three state-of-the-art (SoTA) baselines: Sextans (both TAPA [9] and HLS [15] versions), Leda [20], and DySpMM [17] in Section 6.2. Then we present their overall throughput and energy efficiency comparison summary in Section 6.3, detailed performance and energy efficiency comparison for imbalanced matrices and balanced matrices in Section 6.4 and Section 6.5 using the widely used SuiteSparse [8] benchmark suite. We also perform an ablation study using the SNAP dataset [10] in Section 6.6. Next, we explore the impact of our design space exploration to show further speedup against our two generic designs in section 6.7. Finally, we evaluate the performance impact and hardware cost of our two major optimizations, dense row sharing and HBM decoupling, in Section 6.8 and Section 6.9.

6.1 Experimental Setup

We use our automation tool to generate the HiSpMM designs for the AMD/Xilinx U280 HBM-based FPGAs, with TAPA HLS version 0.0.20221113.1. Vitis version 2023.2 and XRT version 2.14.354 are used to build and run the HiSpMM designs on the actual U280 FPGA board. We have tested a total of 866 sparse matrices obtained from SuiteSparse [8] and 50 matrices obtained from SNAP dataset (Stanford Large Network Dataset Collection) [10] with different numbers of dense matrix columns from 8 to 256. The size of sparse matrices ranges from 7 to 513,351 rows and columns with density from 5.75E-06 to 1. Among 866 SuiteSparse matrices, 92 matrices are imbalanced matrices and 774 matrices are balanced matrices. Our major results are reported for SuiteSparse matrices unless otherwise specified, and we use SNAP matrices for ablation study in Section 6.6.

We run each kernel 1000 times and get the mean time to calculate the throughput. For power data, we run each kernel to have at least 5 valid (non-idle) power samples and average them. The board power consumption is measured using vendor-provided XRT APIs during the kernel runtime. We have extensively compared our generic designs (HiSpMM-balanced and HiSpMM-imbalanced) to state-of-the-art (SoTA) SpMM designs, including Sextans (both TAPA [9] and HLS [15] versions), Leda [20], and DySpMM [17]. We use the open-source bitstream files of SoTA designs and only re-compile their host code with our power measurement API. In addition, we also compare the performance of our DSE-chosen best designs to our two generic designs for further improvement.

6.2 Comparison of Design Configurations and Resource Utilization

Before comparing the performance and energy efficiency, Table 3 first presents a comparative analysis of various hardware configurations for generic SpMM hardware accelerators, including two of our proposed generic designs (HiSpMM-balanced for balanced matrices and HiSpMM-imbalanced for imbalanced matrices) and three state-of-the-art baselines: Sextans (both TAPA [9] and HLS [15] versions), Leda [20], and DySpMM [17]. The table reports architectural parameters such as the number of processing elements (PEs), number of HBM channels assigned to the dense matrix **C** (*C_CH*), whether dense row distribution network (DRDN) is utilized for better balance in imbalanced matrices, and whether HBM channel decoupling is supported. The dense row sharing and HBM decoupling are unique architecture features supported by our HiSpMM designs. Note all prior work use 64 PEs with 8 C channels due to the HBM coupling design (i.e., number of PEs must be a multiple of 8 of number of C channels), while our design allows more flexibility to go as high as 80 PEs. The operating frequency and the post-place-and-route resource utilization results on the U280 FPGA are also included.

Table 3. Comparison of generic SpMM accelerators on Alveo U280 FPGA: architecture features, frequency, and resource utilization

Work	Architecture features				Freq.	Resource utilization (%)				
	#PEs	#C_CH	DRDN	HBM decouple	(MHz)	LUTs	FF	BRAM	DSP	URAM
Sextans-tapa [9]*	64	8	–	No	253	29%	26%	56%	36%	53%
Sextans-hls [15]**	64	8	–	No	197	29%	26%	56%	36%	80%
Leda [20]	64	8	–	No	235	38.3%	22%	42.9%	50.8%	53%
DySpMM [17]	64	8	–	No	180	73%	53%	74%	71%	80%
HiSpMM-imbalanced	64	4	Yes	Yes	225	55%	27%	56%	46%	53%
HiSpMM-balanced	80	4	No	Yes	216	37%	20%	70%	43%	66%

* Sextans ported to TAPA [9] available at https://github.com/linghaosong/Sextans/tree/tapa
** Sextans with HLS design [15] available at https://github.com/linghaosong/Sextans/tree/hls

6.3 Overall Throughput and Energy Efficiency Comparison Summary

Table 4 compares HiSpMM-balanced and HiSpMM-imbalanced against those state-of-the-art SpMM accelerator designs across key performance and energy efficiency metrics, including geomean throughput under both imbalanced and balanced matrices, peak throughput, geomean power consumption, and peak energy efficiency. Our proposed design outperforms prior works in multiple dimensions. First, under imbalanced workload conditions, while all using 64 PEs, our HiSpMM-imbalanced design achieves 36.86 GFLOPS, compared to just 6.36 GFLOPS, 6.04 GFLOPS, and 6.41 GFLOPS for Sextans-tapa, Sextans-hls, and Leda, respectively, demonstrating the effectiveness of our dense row sharing mechanism. Second, our HiSpMM-imbalanced design (64 PEs) achieves a geometric mean throughput of 48.78 GFLOPS and peak throughput of 200.1 GFLOPS, while our HiSpMM-balanced design (80 PEs) reaches 49.97 GFLOPS geometric mean and 244.6 GFLOPS peak throughput, exceeding the performance of Sextans, Leda, and DySpMM. In terms of energy efficiency, our HiSpMM-balanced design reaches 4.17 GFLOPS/W at 80 PEs, surpassing all baselines. Our HiSpMM-balanced design mainly benefits from the decoupling of HBM channels from PEs, which makes an 80-PE configuration feasible with 4 C channels. This enables a peak throughput of 244.6 GFLOPS, which exceeds SoTA SpMM designs by about 20%.

Table 4. Performance and energy efficiency comparison to SoTA generic SpMM accelerators

Design	**Geomean Throughput for Imbalanced Matrices (GFLOPS)**	**Geomean Throughput for All Matrices (GFLOPS)**	**Peak Throughput (GFLOPS)**	**Geomean Power (W)**	**Peak Energy Efficiency (GFLOPS/W)**
Sextans-tapa [9]	6.36	43.26	201.1	54.63	3.38
Sextans-hls [15]	6.04	41.10	181.1	52.00	3.50
Leda [20]	6.41	45.18	203.3	57.04	3.36
DySpMM [17]	–	–	180.0	61.00	2.95
HiSpMM-imbalanced	36.86	48.78	200.1	54.93	3.86
HiSpMM-balanced	–	49.97	244.6	52.38	4.17

6.4 Performance and Energy Efficiency Comparison to SoTA Accelerators for Imbalanced Matrices

Figure 8 plots the detailed comparison of our HiSpMM-imbalanced design against both Sextans (tapa version) and Leda in the presence of imbalanced matrices. Specifically, Figure 8a demonstrates a 5.81× geometric mean speedup and 5.75× geometric mean energy efficiency relative to Sextans. Figure 8b shows a 5.49× geometric mean speedup and 4.97× geometric mean energy efficiency gain over Leda. The span of speedup is from a few times for marginally imbalanced matrices up to 30 times for extremely imbalanced matrices. Many of speedup points are above geomean line. These results further reinforce that our HiSpMM-imbalanced design, thanks to the dense row sharing technique, effectively addresses imbalanced matrices—a limitation largely not addressed in prior work.

6.5 Performance and Energy Efficiency Comparison to SoTA Accelerators for Balanced Matrices

Figure 9 plots the detailed comparison of our HiSpMM-balanced design against both Sextans (tapa version) and Leda in the presence of balanced matrices. Even though the frequency of our HiSpMM-balanced design is slightly lower than that of Sextans and Leda, on average, Figure 9a demonstrates a 1.03× speedup and 1.07× improvement in energy efficiency relative to Sextans, while Figure 9b shows a 1.06× speedup and 1.14× energy efficiency gain over Leda. This is mainly because our HBM channel

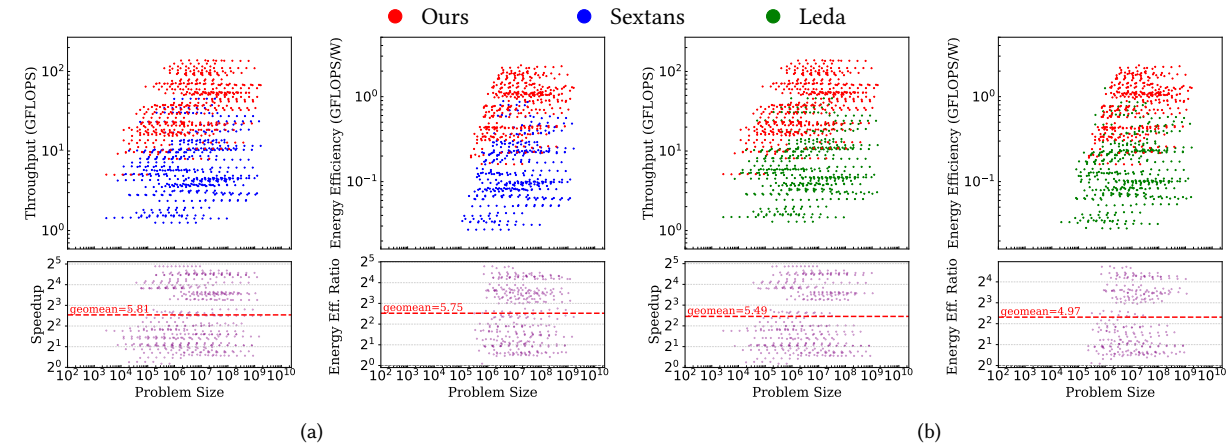


Fig. 8. Our HiSpMM-imbalanced design in comparison to Sextans and Leda for imbalanced matrices.

decoupling allows our HiSpMM-balanced design to have 80 PEs, more than 64 PEs that can be supported by Sextans and Leda. Moreover, we achieve a peak throughput of 244.6 GFLOPS, which exceeds the current SoTA by about 20%.

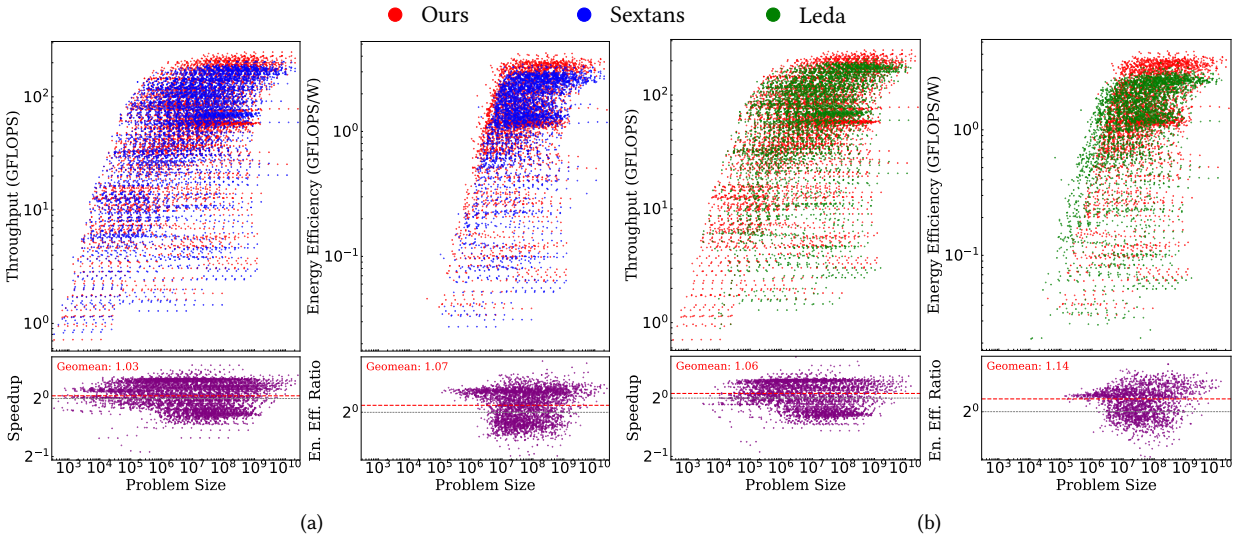


Fig. 9. Our HiSpMM-balanced design in comparison to Sextans and Leda on balanced matrices.

6.6 Ablation Study on SNAP Dataset

In this subsection, we performance an ablation study using the SNAP dataset to evaluate our HiSpMM-imbalanced and HiSpMM-balanced designs against Sextans and Leda. Among the 50 SNAP matrices, there are 8 imbalanced matrices and 42 balanced matrices. As shown in Figure 10, for the imbalanced matrices, our HiSpMM-imbalanced design achieves a geomean speedup of 2.42× and 2.39× over Sextans and Leda. While for the balanced matrices, as shown in Figure 11, our HiSpMM-balanced design achieves a geomean speedup of 1.13× and 1.12× over Sextans and Leda.

6.7 Impact of Our Design Space Exploration

Note our design space exploration (DSE) is matrix property-driven, and it does not depend on the input matrix values. In fact, a chosen best configuration often serves many matrices without per-matrix reconfiguration. As shown in Table 5, based on our DSE, many matrices from SuiteSparse and SNAP datasets (the number of matrices shown in the last column) converge to choose one of the following five best designs. Table 5 also lists the achieved frequency and post-place-and-route resource utilization of each DSE-chosen best design configuration. To demonstrate the effectiveness of our DSE, in Table 6, we also compare the throughput of our DSE-chosen best design configuration against our generic designs presented in Section 6.4 and Section 6.5.

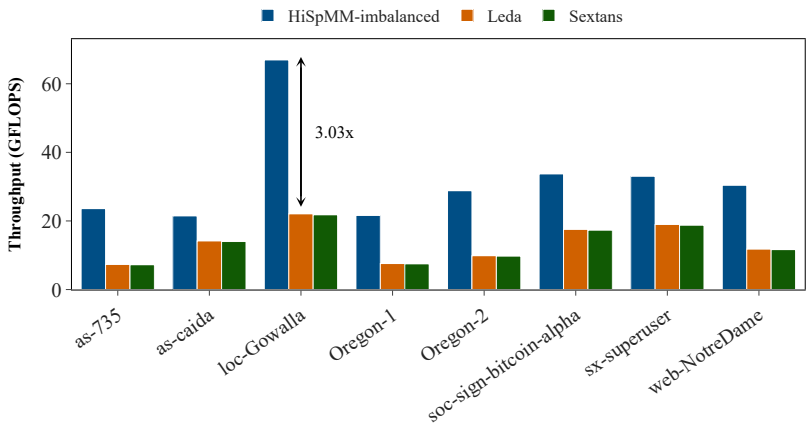


Fig. 10. Throughput comparison of HiSpMM-imbalanced, Leda and Sextans for imbalance matrices in studied SNAP dataset.

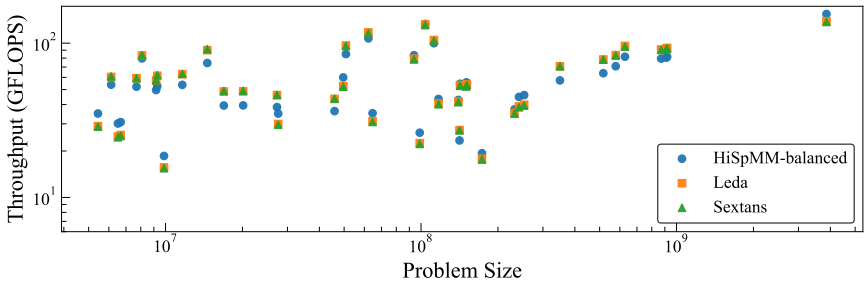


Fig. 11. Throughput comparison of HiSpMM-balanced, Leda and Sextans for balanced matrices in studied SNAP dataset.

- 260 matrices choose design configuration 1 (i.e., our generic HiSpMM-balanced design): 80 PEs, 4 C channels, and without DRDN. These are mostly for balanced matrices that are more matrix multiplication bound. For these matrices, our DSE-chosen best design is the same as our generic HiSpMM-balanced design.
- 216 matrices choose design configuration 2: 64 PEs, 8 C channels, and without DRDN. These are mostly for balanced matrices that are more bandwidth bound for dense matrix C. For these matrices, our DSE-chosen best design achieves a geomean speedup of 1.05× and up to 1.40× speedup compared to our generic HiSpMM-balanced design.
- 29 matrices choose design configuration 3: 48 PEs, 8 C channels, and without DRDN. These are mostly for small matrices that are relatively balanced and more bandwidth bound for dense matrix C. For these matrices, our DSE-chosen best design achieves a geomean speedup of 1.02× and up to 1.04× speedup compared to our generic HiSpMM-balanced design.
- 30 matrices choose design configuration 4 (i.e., our generic design): 64 PEs, 4 C channels, and with DRDN. These are mostly for imbalanced matrices that are more matrix multiplication bound. For these matrices, our DSE-chosen best design is the same as our generic HiSpMM-imbalanced design.
- 67 matrices choose design configuration 5: 48 PEs, 8 C channels, and with DRDN. These are mostly for imbalanced matrices that are bandwidth bound for dense matrix C. For these matrices, our DSE-chosen best design achieves a geomean speedup of 1.07× and up to 1.82× speedup compared to our generic HiSpMM-imbalanced design.

6.8 Impact of Dense Row Sharing and Its Hardware Cost

Performance Speedup. While section 6.4 already showed the great performance speedup enabled by our dense row sharing strategy using the Dense Row Distribution Network (DRDN), here we perform a more controlled experiment to demonstrate the exact speedup brought by DRDN for imbalanced matrices. As shown in Table 7, we measure two designs: 1) 48-PE design with 8 C channels, and 2) 64-PE design with 4 C channels. In each design, we measure the impact of DRDN by enabling and disabling only DRDN. As shown in the last column, for imbalanced matrices, enabling DRDN can bring a geomean speedup of 5.20× and 5.97× for each design.

Table 5. DSE-chosen best design configuration for each category of matrices.

Design #	#PE	#C_CH	DRDN	Freq. (MHz)	LUTs	FF	BRAM	DSP	URAM	# Matrices
Design 1*	80	4	No	216	37%	20%	70%	43%	66%	260
Design 2	64	8	No	204	34%	18%	56%	40%	53%	216
Design 3	48	8	No	225	29%	16%	42%	29%	40%	29
Design 4**	64	4	Yes	225	55%	27%	56%	46%	53%	30
Design 5	48	8	Yes	225	46%	25%	42%	42%	40%	67

* Generic HiSpMM-balanced design for balanced matrices
** Generic HiSpMM-imbalanced design for imbalanced matrices

Table 6. Effectiveness of our DSE: how many matrices converge and the speedup from our DSE

HiSpMM Designs	# Matrices	Geomean Throughput (GFLOPS)		DSE Speedup	
		Generic Design	DSE-Chosen Design	Geomean	Max
Design 1	260	112.09	112.09	1.00	1.00
Design 2	216	43.70	45.78	1.05	1.40
Design 3	29	4.90	5.02	1.02	1.04
Design 4	30	47.90	47.90	1.00	1.00
Design 5	67	29.06	31.10	1.07	1.82

Hardware Cost. Employing DRDN introduces a resource overhead, which is dominated by LUTs as shown in Table 7. Note, DRDN does not utilize any BRAM or URAM. For the 48-PE design, compared to the design without DRDN, the design with DRDN introduces 17% more LUT usage (of the total available LUTs on U280 FPGA). Similarly, for the 64-PE design, DRDN introduces 25% more LUT usage. Its performance speedup of 5.20× and 5.97× for the 48-PE and 64-PE designs on imbalanced matrices well justifies this hardware cost. In addition, we would like to point out that the designs without DRDN are mostly bound by the BRAM resources and LUTs are underutilized. With DRDN, the LUT utilization percentage comes to a similar level of BRAM percentage.

Table 7. Performance speedup using Dense Row Distribution Network (DRDN) in HiSpMM and its resource overhead

#PEs	#C_CH	Freq	LUTs (%)	FF (%)	BRAM (%)	DSP (%)	URAM (%)	DRDN	Speedup
48	8	225	29	16	42	34	40	No	1×
48	8	225	46 (+17)	25 (+9)	42	42 (+8)	40	Yes	5.20×
64	4	225	30	16	56	34	53	No	1×
64	4	225	55 (+25)	27 (+11)	56	46 (+12)	53	Yes	5.97×

6.9 Impact of HBM Decoupling and Its Hardware Cost

Performance Speedup. While section 6.5 already showed the flexibility and performance speedup enabled by computation and HBM decoupled design, here we perform a more controlled experiment to demonstrate the exact speedup brought by HBM decoupling. First of all, we note that in presence of DRDN and 8 C channels (8 for reading and another 8 for writing C matrix), a design with 64 PEs and coupled 8 C channels will require 86.9% LUTs by the synthesis report, which fails the implementation. To make such a 64-PE design with DRDN enabled, we need to decouple the number of PEs and the number of C channels, i.e., removing the requirement that the number of PEs must be a multiple of 8 of that of C channels. In fact, we can build the 64 PEs with 4 C channels and DRDN enabled, which is our generic HiSpMM-imbalanced design. This case shows the great flexibility brought by the decoupled design to enable more PEs. In addition, as shown in Table 8, we also compare two more designs: 1) 48-PE coupled design with 6 C channels, and 2) 64-PE decoupled design with 4 C channels (more PEs enabled by the HBM decoupling). Comparing these two, the decoupled design achieves a geomean speedup of 1.08× for imbalanced matrices and 1.07× for all matrices. Finally, comparing the second and third designs in Table 8, the DRDN gives an extra speedup of 5.97× for imbalanced matrices, which can be stacked with the speedup from HBM decoupling: the third design achieves a geomean speedup of 6.47× compared to the first design.

Hardware Cost. Regarding the hardware cost of the interconnection network between PEs and C channels to support their decoupling, we use a lightweight hierarchical arbiter to provide the connections. Each arbiter module connects multiple PEs to a C channel. The reason for having hierarchical arbiters is to avoid a hotspot on the fabric where all the PEs need to connect and may

Table 8. Performance impact of dense row sharing and HBM decoupling

#PEs	#C Channels	Coupling HBM Channels	DRDN Enabled	Geomean Throughput on Imbalance Matrices	Geomean Throughput for All Matrices
48	6	Coupled	No	5.70	40.83
64	4	Decoupled	No	6.17	43.52
64	4	Decoupled	Yes	36.86	48.74
64	8	Coupled	Yes	Fail	Fail

cause routing failure. Also, it shrinks the latency of arbitration (negligible) and the arbiter modules run in parallel. The arbiter does not contribute to interconnection density nor interconnection congestion on the fabric. For a 64-PE design with 8 C channels, we use eight 8-to-1 arbiter modules; each 8-to-1 arbiter module just consumes 3,360 LUTs and 1,650 FFs. For a 64-PE design with 4 C channels, we use four 16-to-1 arbiter modules; each 16-to-1 arbiter module consumes twice as much as an 8-to-1 arbiter module, keeping the resource utilization for the interconnection network of both designs quite similar.

7 Conclusion and Future Work

In this work, we identified the workload imbalance issue as the most significant challenge to accelerate sparse matrix - dense matrix multiplication (SpMM) on FPGAs. We provided a precise analysis and solution to balance the workload on identified imbalance matrices. Our experiments show this issue is well addressed by HiSpMM, resulting in a geomean of 5.81× speedup and 5.75× energy efficiency improvement for imbalanced matrices. We also enabled higher number of computation units for balanced matrices by decoupling the HBM channels from PEs. It exhibits better performance compared to SoTA designs. Furthermore, we provide a matrix-property driven design exploration and automation tool (DSE) which can find the best configuration specific to matrix property and user's resource constraints. The DSE demonstrates that various matrices can benefit from the flexibility and scalability of our design, particularly with respect to specific characteristics such as matrix sparsity, imbalance ratio, and matrix dimensions.

For future work, first, we plan to open source our HiSpMM design and toolflow in near future at https://github.com/SFU-HiAccel/HiSpMM. Second, our current design has a one-time preprocessing overhead, which is well amortized and negligible for many applications that call the SpMM kernel iteratively. In future work, we plan to explore dynamically accelerating the preprocessing step as well for non-iterative SpMM applications. Lastly, HiSpMM only supports single precision floating-point data type at the moment, while quantization is widely used in neural networks that use SpMM kernels. In future work, we plan to support different data types for HiSpMM.

Acknowledgments

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

References

[1] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2023. Fast Sparse GPU Kernels for Accelerated Training of Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 501–511. doi:10.1109/IPDPS54959.2023.00057

[2] Hayato Goto, Kotaro Endo, Masaru Suzuki, Yoshisato Sakai, Taro Kanao, Yohei Hamakawa, Ryo Hidaka, Masaya Yamasaki, and Kosuke Tatsumura. 2021. High-performance combinatorial optimization based on classical mechanics. *Science Advances* 7, 6 (2021), eabe7953. doi:10.1126/sciadv.abe7953 arXiv:https://www.science.org/doi/pdf/10.1126/sciadv.abe7953

[3] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[5] Václav Hapla, David Horák, and Michal Merta. 2012. Use of direct solvers in TFETI massively parallel implementation. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing* (Helsinki, Finland) *(PARA'12)*. Springer-Verlag, Berlin, Heidelberg, 192–205. doi:10.1007/978-3-642-36803-5_14

[6] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A flexible and efficient backend for graph neural network systems. (2020). https://www.amazon.science/publications/featgraph-a-flexible-and-efficient-backend-for-graph-neural-network-systems

[7] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS.

In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC.2016.7761646

[8] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244. doi:10.21105/joss.01244

[9] UCLA VAST Lab. [n. d.]. Sextans: FPGA-Accelerated Sparse Matrix Multiplication. https://github.com/UCLA-VAST/Sextans. Accessed: 2025-06-25.

[10] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[11] Alec Lu, Zhenman Fang, and Lesley Shannon. 2022. Demystifying the Soft and Hardened Memory Systems of Modern FPGAs for Software Programmers through Microbenchmarking. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 43 (June 2022), 33 pages. doi:10.1145/3517131

[12] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. 2013. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference, HPEC 2013 (2013 IEEE High Performance Extreme Computing Conference, HPEC 2013)*. IEEE Computer Society. doi:10.1109/HPEC.2013.6670338 2013 IEEE High Performance Extreme Computing Conference, HPEC 2013 ; Conference date: 10-09-2013 Through 12-09-2013.

[13] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 27–40. doi:10.1145/3140659.3080254

[14] Manoj B. Rajashekar, Xingyu Tian, and Zhenman Fang. 2024. HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs. *FPGA 2024 - Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (4 2024), 154–164. doi:10.1145/3626202.3637557

[15] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young Kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *FPGA 2022 - Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, Inc, 65–77. doi:10.1145/3490422.3502357

[16] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. doi:10.14778/2809974.2809983

[17] Hongyi Wang, Kai Zhong, Haoyu Zhang, Shulin Zeng, Zhenhua Zhu, Xinhao Yang, Shuang Wang, Guohao Dai, Shanghai Jiao, Huazhong Yang, and Yu Wang. 2024. DySpMM: From Fix to Dynamic for Sparse Matrix-Matrix Multiplication Accelerators. *Proceedings of the 61st ACM/IEEE Design Automation Conference* (6 2024), 1–6. doi:10.1145/3649329.3657362

[18] Ichitaro Yamazaki and Xiaoye S. Li. 2011. On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver. In *High Performance Computing for Computational Science – VECPAR 2010*, José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 421–434.

[19] Serif Yesil, José E. Moreira, and Josep Torrellas. 2022. Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques. In *Proceedings of the 36th ACM International Conference on Supercomputing, ICS 2022 (Proceedings of the International Conference on Supercomputing)*. Association for Computing Machinery, United States. doi:10.1145/3524059.3532369 This work was supported in part by the IBM-Illinois Discovery Accelerator Institute, and by NSF under grants CNS-1763658, CNS-1956007, CCF-2028861, and CCF-2107470.; 36th ACM International Conference on Supercomputing, ICS 2022 ; Conference date: 27-06-2022 Through 30-06-2022.

[20] Enxin Yi, Jiarui Bai, Yijie Nie, Dan Niu, Zhou Jin, and Weifeng Liu. 2025. Leda: Leveraging Tiling Dataflow to Accelerate SpMM on HBM-Equipped FPGAs for GNNs. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* (4 2025). doi:10.1145/3676536.3676773