Misam: Machine Learning Assisted Dataflow Selection in Accelerators for Sparse Matrix Multiplication

Sanjali Yadav
University of Maryland
College Park, USA
sanjali7@umd.edu

Amirmahdi Namjoo
University of Maryland
College Park, USA
namjoo@umd.edu

Bahar Asgari
University of Maryland
College Park, USA
bahar@umd.edu

Abstract

The performance of Sparse Matrix-Matrix Multiplication (SpGEMM), a foundational operation in scientific computing and machine learning, is highly sensitive to the diverse and dynamic sparsity patterns of its input matrices. While specialized hardware accelerators improve efficiency, their reliance on fixed dataflows, each optimized for a narrow sparsity regime, results in suboptimal performance on real-world workloads. Even recent flexible accelerators that support multiple dataflows face two critical limitations: (1) the lack of a fast and principled mechanism for runtime dataflow selection, and (2) the area overhead and hardware underutilization incurred to provide that flexibility. We present Misam, a machine learning framework that addresses these challenges to enable adaptive and hardware-efficient SpGEMM acceleration. Misam employs a lightweight decision tree to dynamically predict the optimal hardware configuration from matrix features. To overcome hardware underutilization, Misam leverages FPGA reconfigurability to deploy specialized, resource-efficient bitstreams on demand. This process is governed by an intelligent reconfiguration engine that evaluates whether the anticipated performance gain justifies the overhead of switching hardware configurations. Misam's dynamic approach yields up to a 10.76× speedup by judiciously reconfiguring. Misam demonstrates that a synergistic combination of machine learning-based prediction and judicious hardware reconfiguration can achieve high performance across a wide spectrum of sparsity patterns, bridging the gap between specialized efficiency and general-purpose adaptability.

Keywords

Flexible SpGEMM accelerators, FPGA Reconfiguration, Decision Tree.

ACM Reference Format:
Sanjali Yadav, Amirmahdi Namjoo, and Bahar Asgari. 2025. Misam: Machine Learning Assisted Dataflow Selection in Accelerators for Sparse Matrix Multiplication. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756126

1 Introduction

Sparse matrix-matrix multiplication (SpGEMM) serves as a crucial kernel for intricate operations across various fields such as scientific computing [19, 29, 92], graph analytics [3, 9, 65], and machine learning [10, 21, 56]. SpGEMM computations capitalize on efficiently discarding ineffectual zero elements, substantially reducing storage demands and computational load, especially in scenarios characterized by large matrices with high levels of sparsity.

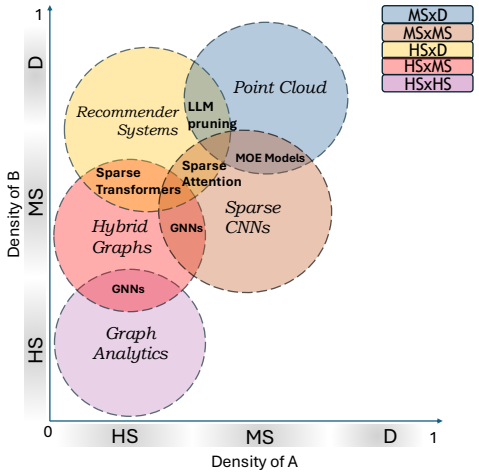

Figure 1: Various sparse applications across distinct regions of the sparsity space. *Sparsity of A* × *Sparsity of B* is color coded. HS: Highly Sparse, MS: Mildly Sparse, and D: Dense.

The diverse sparsity patterns observed in traditional scientific computing, coupled with the growing adoption of sparsity in emerging domains such as deep neural networks [18, 30, 64], large language models [16, 21, 104], and recommendation systems [11, 45, 55, 110], have significantly increased the demand for high-performance and efficient SpGEMM. Figure 1 illustrates how various sparse applications cluster in distinct regions of the sparsity space. However, these regions are not mutually exclusive, and a single application may traverse multiple sparsity regimes during execution. A **sparsity regime** refers to a combination of the sparsity level of the matrix and structural characteristics, such as the distribution of nonzeros, regularity, and symmetry. For instance, neural networks may initially exhibit moderate sparsity in their input matrices, but techniques such as pruning [28, 83] or sparsity-inducing regularization [59, 81] can significantly increase sparsity in specific layers. As these patterns evolve dynamically, both software and hardware systems must adapt to maintain performance and efficiency.

As SpGEMM gains traction across various domains, the need for hardware accelerators that can handle varying sparsity levels and structural irregularities, has grown significantly. Early accelerators such as OuterSPACE [71], MatRaptor [88], SpArch [108], GAMMA [107], and InnerSP [4] were designed for highly sparse

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756126

inputs, while others like SIGMA [77], DSTC [106], and HighLight [96] target moderately sparse matrices. However, these designs typically support a narrow range of sparsity regimes and are optimized around fixed dataflow strategies, limiting their adaptability to real-world, evolving workloads.

To overcome these limitations, more recent designs aim to build versatile accelerators capable of adapting to a wide range of sparsity regimes. For instance, Trapezoid [102] introduces novel dataflow schemes to support a broad range of sparsity levels, while Flexagon [66] employs a reconfigurable architecture to handle diverse sparse workloads. Although these architectures represent an important step toward general-purpose sparse acceleration, they still lack key capabilities. In particular, neither provides a robust mechanism for selecting the optimal dataflow based on the input matrices. Flexagon relies on a simple offline profiling method, deferring the challenge of comprehensive dataflow selection to future work. Trapezoid, while supporting multiple dataflows, offers no dynamic strategy for selecting among them at runtime.

A second challenge faced by these versatile accelerators is the cost of flexibility. Supporting both dense and sparse workloads often requires incorporating heterogeneous hardware components—compute-dense units for dense regimes, and sparsity-aware units for sparse workloads. This increases area overhead and can lead to significant underutilization. For example, when executing a dense workload, the sparse-specific units often remain idle. Although Trapezoid attempts to reuse some of these units, many components such as local buffers, sparse schedulers, and distribution networks necessary for sparse execution are underutilized during dense phases. As a result, these accelerators suffer from inefficiencies stemming from hardware underutilization and increased resource costs associated with generality.

In this work, we aim to address two key challenges: (1) enabling flexible and dynamic selection of optimal dataflow schemes based on input sparsity characteristics, and (2) minimizing hardware underutilization and area overhead in versatile accelerator designs.

To tackle the first challenge, we propose a machine learning (ML)-driven approach for dynamic dataflow selection. We broaden the definition of a dataflow scheme to not only encompass the scheduling of data elements to processing elements (PEs), but also the configuration of supporting hardware units that facilitate data distribution and computation. The task of selecting an appropriate dataflow scheme naturally aligns with ML-based classification: by extracting features from input matrices, we classify them corresponding to specific dataflow implementations optimized for different sparsity regimes.

To address the second challenge of reducing hardware overhead and avoiding underutilization, we target an FPGA-based implementation. Specifically, we leverage the reconfigurability of FPGAs to dynamically switch between bitstreams corresponding to different dataflow schemes. Each bitstream is tailored to utilize only the hardware resources required for its targeted subset of sparsity regimes, thereby achieving generality without compromising efficiency. Additionally, a reconfiguration engine is introduced, which plays a key role in determining when switching designs is beneficial—an often non-trivial decision due to runtime overheads.

Building on these two insights, we introduce Misam [1], an ML-based framework for dynamic dataflow scheme selection and intelligent hardware reconfiguration. Misam prototypes sparse matrix-matrix multiplication on the Xilinx Alveo U55C FPGA, equipped with high-bandwidth memory (HBM). The host CPU predicts the optimal dataflow scheme for a given input and loads the corresponding bitstream onto the FPGA. A key feature of Misam is its intelligent reconfiguration strategy. Selecting the best-performing dataflow scheme alone is insufficient; the system must also determine whether the reconfiguration overhead is justified by the expected performance gain. Misam models this trade-off by evaluating both the predicted performance of each dataflow scheme and the overhead associated with bitstream reconfiguration. It then makes a holistic decision on whether reconfiguration is justified.

To evaluate our models, we compiled a dataset of 6,219 matrices with sparsity levels ranging from 1% to 99%, combining highly sparse matrices from SuiteSparse [12] with moderately sparse and dense matrices from deep learning workloads. This diversity enables robust and generalizable dataflow selection. We compared our dataflow scheme predictions against widely used libraries such as Intel MKL and cuSPARSE, both optimized for sparse matrix multiplication, and achieved speedups of 15.33× over MKL, 4.48× over cuSPARSE, and 3.23× over Trapezoid's fixed dataflows. Finally, we integrated Misam with Trapezoid's dataflows, demonstrating its compatibility and practical applicability in existing accelerator architectures, with a selection accuracy of 90%.

In summary, Misam makes the following key contributions:

- We formulate dataflow scheme selection as an ML classification problem and use lightweight decision trees to select optimal schemes. This model applies to any application leveraging multiple dataflow options.
- We develop a reconfiguration engine that evaluates whether switching dataflow schemes is beneficial based on runtime metrics. The engine is compatible with any architecture supporting reconfiguration at various granularities.
- We introduce a set of FPGA-based dataflow scheme implementations that demonstrate the functionality of Misam's dataflow scheme selection and reconfiguration engine.

2 Background & Targeted Challenges

Sparse matrix-matrix multiplication (SpGEMM) plays a pivotal role in numerous domains such as scientific computing, graph analytics, and modern machine learning. Its computational advantage stems from exploiting the sparsity in input matrices—ignoring zero elements to reduce both memory and compute overhead. This makes SpGEMM especially effective for large-scale systems where data is often inherently sparse.

2.1 Dataflows for Sparse Matrix Multiplication

Various dataflow strategies can be employed to multiply input matrices $A_{M\times K}$ and $B_{K\times N}$ to produce $C_{M\times N}$. The three common ones are presented in Figure 2.

Inner Product (Figure 2, left): Dataflow multiplies a row of A with a column of B, requiring A to be in compressed sparse row (CSR) and B to be in compressed sparse column (CSC) format to mitigate

[1] **Misam** is a star in the constellation Perseus.

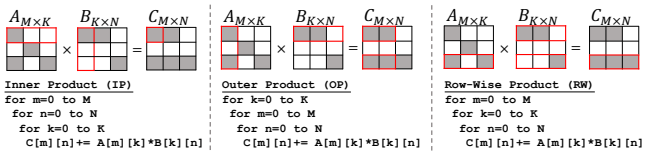

Figure 2: Three SpGEMM Dataflow Approaches– Inner Product (IP), Outer Product (OP), and Row-wise Product (RW).

irregular memory access. While it enables direct accumulation into C, it suffers from redundant fetching of B's columns—once per row of A. Enhancements such as better caching [4], PE utilization [77], and intersection detection [36] improve its efficiency.

Outer Product (Figure 2, middle): Dataflow pairs columns of A with rows of B, maximizing input reuse by avoiding index matching. However, the resulting partial matrices of C can exceed on-chip memory limits, leading to high off-chip traffic. Prior work mitigates this by decoupling accumulation [71], improving locality [108], optimizing off-chip memory [37], and pattern-aware scheduling [86].

Row-wise Product (Figure 2, right): Dataflow multiplies all nonzero elements of a row in matrix A with corresponding nonzero elements in matrix B, ensuring uniform input formats and avoiding index matching. It supports effective output reuse and minimizes zero outputs. However, irregular access to B's rows due to sparse and unstructured nonzero distribution in A reduces reuse efficiency. Improvements by recent works include hardware-friendly sparse formats [88] and custom FPGA accelerators [2].

Due to fundamental architectural differences, each dataflow exhibits distinct performance characteristics depending on the sparsity pattern of the input. These accelerators are typically optimized for specific sparsity structures that best align with their underlying hardware. They adopt a fixed dataflow that favors either input or output reuse—often at the expense of the other. Consequently, performance degrades when the workload's sparsity pattern does not match the assumptions baked into the accelerator's design.

Recent efforts have sought to address this limitation by proposing more flexible SpGEMM architectures capable of handling a wider variety of input patterns. Notable examples include Flexagon[66] and Trapezoid[102], which represent early steps toward general-purpose sparse matrix multiplication accelerators. Flexagon employs a reconfigurable architecture that adapts to varying sparsity regimes, while Trapezoid uses a modular design to support multiple dataflows. However, both rely on static configuration or offline profiling to choose an execution strategy, limiting their adaptability to dynamic sparsity patterns at runtime.

This leads to our first motivation: **the lack of fast and accurate dataflow selection**. Given the high variability in sparsity across matrices and applications, heuristic-based selection is often insufficient. Manual heuristics are often brittle, require domain expertise to update, and lack quantified accuracy. In contrast, machine learning provides a strong, data-driven alternative that is both low-overhead and highly accurate. We therefore frame dataflow selection as a classification problem, which creates an adaptable model that can be easily retrained as workloads evolve. This approach replaces fixed, manual rules with an automated, learning-based optimization that remains effective across diverse and changing sparsity patterns.

2.2 FPGA Implementation of Dataflows

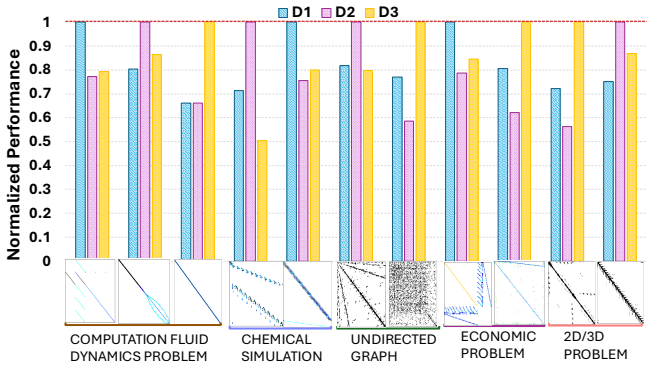

Figure 3: Performance comparison of Misam's design suite (D1, D2, D3) across workloads from diverse applications (normalized to the best design)

Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling platform for accelerating sparse matrix computations. Their architecture offers massive fine-grained parallelism, and their reconfigurable fabric enables the design of deeply pipelined, custom data paths and memory systems. This combination is particularly effective for handling the irregular memory access patterns and low arithmetic intensity characteristic of sparse workloads, often yielding significant performance and energy efficiency gains over traditional CPUs and GPUs. Prior works have proposed numerous specialized accelerators for sparse linear algebra kernels [8, 17, 35, 39, 79, 85, 86].

Sextans[86] and FSpGEMM[8] are two prominent works that implement outer-product-like dataflows for sparse matrix-dense matrix multiplication (SpMM) and row-wise-like dataflows for sparse matrix-matrix multiplication (SpGEMM), respectively. While both achieve high performance within their targeted kernels, they are specialized for either SpMM or SpGEMM and do not fully leverage the reconfigurability of FPGAs to adapt to a broader range of workloads. This brings us to our second motivation: **to harness the reconfigurable nature of FPGAs to build a more generalizable hardware architecture**—one that can dynamically select and switch between dataflows based on the workload characteristics. Unlike Sextans and FSpGEMM, which rely on static configurations or offline profiling, our approach enables runtime dataflow prediction and reconfiguration, eliminating the need for extensive pre-analysis and allowing the hardware to respond to diverse and changing sparsity patterns on the fly.

Figure 3 presents the performance comparison across workloads for three of the Misam designs, where the performance is normalized to the best design. The figure includes a matrix footprint of each workload to illustrate the variety in sparsity patterns. This highlights a key motivation for our work: no single design consistently outperforms others across all sparse workloads. Even within the same application domain, such as computational fluid dynamics, different sparsity regimes lead to different designs yielding better performance. By supporting multiple dataflows and runtime reconfigurability, our architecture adapts to matrices from diverse

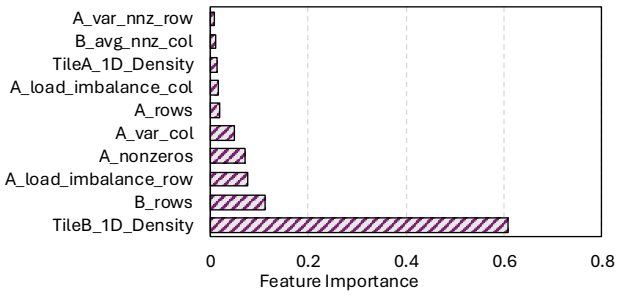

Figure 4: Feature Selection – Analysis of the feature importance in the decision tree.

application domains, including those within the same domain but with varying sparsity regimes, achieving performance portability without sacrificing efficiency.

3 Misam

This section details the core components of Misam. We first describe the ML-based predictor and the hardware it controls. Section 3.1 covers the ML-based dataflow predictor, a lightweight decision tree that classifies matrix features to select an optimal dataflow with high accuracy and minimal overhead. Section 3.2 then introduces the suite of specialized FPGA designs that the model chooses from, each architecturally tuned for different sparsity regimes to provide a range of efficient hardware options.

Building on these components, Section 3.3 explains the reconfiguration engine that governs the system's dynamic behavior. This engine intelligently decides whether to trigger a hardware switch by using a secondary model to weigh the predicted performance gain against the runtime overhead of reconfiguration. This ensures that hardware adaptivity is only employed when it is truly beneficial.

3.1 ML-Based Dataflow Predictor

A decision tree is only as effective as the feature set used to describe the data, and capturing the characteristics of sparse matrices is a particularly challenging task due to the multifaceted nature of sparsity. This includes not just tracking the number of nonzeros, but also understanding matrix patterns, symmetry, regularity, and structural properties, and encoding them in a way that is meaningful for learning. To address this, we began with a comprehensive list of candidate features designed to represent various aspects of sparsity.

This initial feature set included: the sparsity of matrices A and B, the average and variance of nonzeros per row and column in both matrices, tile density under both 1D and architecture-aware 2D tiling schemes, the total number of 1D and 2D tiles, and inthe ratio of the longest row/column to the average row/column length (as a measure of potential imbalance). These features are efficiently derived from the CSR and CSC formats using row and column pointer offsets.

Using this full set, we trained an initial decision tree and then analyzed feature importance to identify the most influential predictors in selecting the optimal design, as shown in Figure 4. Among the selected features, $Tile\_1D\_Density$, which measures the average density of tiles in matrix B when it is partitioned in one dimension, and row_B (the number of rows in matrix B) emerged as the most impactful. The model associates specific matrix dimensions and corresponding tile densities with typical workload characteristics. For instance, in machine learning workloads, matrix B commonly has dimensions such as 256, 512, 1024, or 2048, and is often dense or moderately sparse due to pruning. In contrast, scientific workloads typically involve much larger matrices with low tile density, which strongly suggests that B is highly sparse.

Other key features include $A\_load\_imbalance\_row$ and A_rows, which capture the ratio of the longest row in matrix A to the average row length, and the total number of rows in A, respectively. These features help the model detect potential load imbalances when distributing rows of A across PEs, allowing it to select designs that better accommodate such irregularities. The remaining features shown in the figure further enrich the representation of matrix structure, enabling the model to learn more nuanced sparsity patterns and their associated optimal execution strategies. Features not included in the figure were pruned from the model, as their removal had no measurable impact on accuracy and contributed to a more lightweight and efficient decision tree.

To train our decision tree, we split the dataset into 70% training and 30% validation sets. The dataset contains comprehensive information about running each design across all workloads, including key performance metrics such as latency, throughput, PE utilization, and energy consumption. The decision tree learns to associate these features with the observed performance of each design to accurately predict the optimal configuration.

During training, we observed class imbalance in the dataset, which could bias the model toward the majority class. To mitigate this, we applied a class weighting strategy, assigning weights inversely proportional to class frequency. This approach improves the model's sensitivity to underrepresented classes and results in a more balanced and generalizable predictor. The final decision tree is highly compact, requiring only 6 KB of storage, making it suitable for deployment in memory-constrained environments. Through 10-fold cross-validation, the model achieved a prediction accuracy of 90%, demonstrating both its effectiveness and efficiency.

Misam employs a decision tree model due to its lightweight footprint and low-latency inference, making it well-suited for systems with limited storage and computational resources. Currently, the model resides on the host, where it analyzes matrix features and selects the optimal design configuration, requiring just 6KB of storage. In future iterations, if inference is migrated to the FPGA to enable on-device reconfiguration decisions, the model's efficiency and small memory footprint become even more critical.

Furthermore, Misam allows users to prioritize performance metrics based on their application requirements. For example, a user may choose to optimize exclusively for performance, prioritize energy efficiency, or apply a weighted combination of multiple objectives. This tunable decision-making capability makes Misam adaptable to a wide range of use cases. While adding more objectives increases the decision tree's complexity and, consequently, inference time, the overhead remains modest. For instance, when optimizing solely for latency, the inference cost is less than 0.1% (see Section 5.5). As additional objectives are introduced, the tree becomes deeper and more complex, but given the model's inherently efficient structure, supporting two or three objectives is unlikely to impose significant performance penalties.

3.2 Misam Designs

Table 1: Parameter Configurations for Different Designs (Uncomp: Uncompressed, Comp: compressed).

Parameter	ID	Design 1	Design 2	Design 3	Design 4
ch_A	A	8	12	12	8
ch_B	B	4	4	4	8
ch_C	C	8	12	12	4
PEG	N	16	24	24	16
ACCG	M	16	24	24	16
Scheduler A	SA	Col	Col	Row	Col
Format B	CB	Uncomp.	Uncomp.	Uncomp	Comp.

This section introduces four designs, each tailored to different sparse workloads to showcase the functionality of Misam. These designs highlight that varying sparsity regimes benefit from specialized architectures optimized for their requirements. Later, in Section 3.3, we discuss how the inherent reconfigurability of FPGAs enables seamless switching between such architectures, and how Misam's portable prediction model supports an efficient pipeline for decision-making and reconfiguration. The framework is also flexible: users can integrate their own custom designs, and Misam can readily adapt to these use cases. We elaborate on Misam's adaptability in Section 6.3.

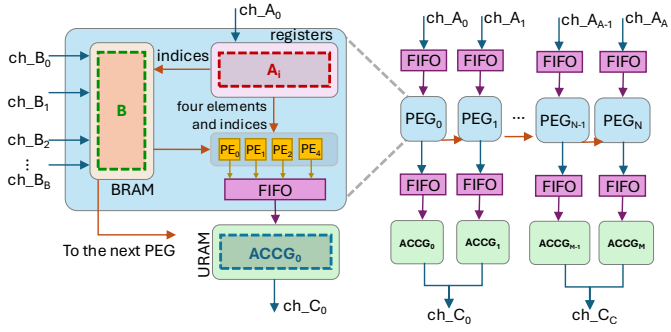

Figure 5: The microarchitecture of Misam

3.2.1 ***Design 1:*** The design is built like the Sextans accelerator [86] but adapted for a more resource-constrained Alveo U55C FPGA. The configuration of this design is summarized in Table 1. The architecture illustrated in figure 5 comprises N Processing Element Groups (PEGs), each with 4 PEs. It utilizes separate HBM channels for reading matrices A and B (ch_A and ch_B), and writing the output matrix C (ch_C). Sparse matrix A is partitioned and distributed across ch_A, where nonzero elements are streamed into PEG FIFOs. In contrast, dense matrix B is row-tiled and interleaved across ch_B channels. Each PEG receives A values via its FIFO, while the first PEG also reads B rows, which are forwarded downstream to subsequent PEGs through a broadcasting network, ensuring synchronized row processing across PEGs.

The architecture follows a row-wise product model: the column index of each nonzero A element identifies the corresponding row of B to be multiplied. Scheduling information is pre-generated on the host and includes a pointer list for each PEG, specifying how many A elements to consume per iteration. Each PE multiplies an element of A with a matching row element of B, accumulating partial results into eight-element vectors that are streamed to dedicated accumulator groups (ACCGs). Each ACCG consists of four accumulators (ACCs), which sum the results and store them in URAMs. B rows are temporarily buffered in BRAM for rapid access, while partial results from A are stored in URAMs due to BRAM capacity. Once a tile's computation is complete, the accumulated result is written to ch_C.

To optimize HBM bandwidth, 8 elements of A are coalesced into a 64-bit word containing row index, column index, and value. Similarly, 16 FP32 values from B are packed per read. Matrix A is column-tiled on the host, while matrix B is tiled in both dimensions: row tiling is based on BRAM capacity (4096 entries), and column tiling is limited by the number of PEGs. This strategy suits workloads like those in machine learning, which often feature tall-and-skinny matrices where smaller B tiles are acceptable.

Figure 6 presents a toy example highlighting the differences among the three SpMM kernel designs. In Design 1, the number of PEGs is reduced to one, with two PEs per PEG. Designs 2 and 3 illustrate increased parallelism by using two PEGs. Each design traverses matrix A in a column or row-wise fashion under a load/store dependency constraint of 2 cycles and assigns work to each PE in a round-robin manner. The figure includes input matrices with varying sparsity patterns to demonstrate how each design handles computation internally.

To estimate the total number of cycles, we account for the time required to read matrix B, set to 3 cycles in this example, as all designs access the same B matrix. Besides, we include a placeholder for the time needed to broadcast B. Once a PEG receives its segment of B, it begins computation in parallel while forwarding B to the next PEG. Since all PEGs operate concurrently, the overall computation time is determined by the PEG that completes its task last.

3.2.2 ***Design 2:*** The design builds upon Design 1 but introduces key changes to memory bandwidth allocation and resource scaling. As seen in Table 1, the primary distinction lies in the use of additional HBM channels for matrix A and C (i.e., increased ch_A and ch_C), and an expanded number of PEs. This configuration is selected by our decision tree for large and denser matrices with relatively consistent per-row sparsity. In such workloads, the higher degree of memory parallelism ensures more efficient processing. Design 2 fully utilizes available channels to accelerate large-scale sparse computations compared to design 1. The rationale for including Design 1, in addition to Designs 2 and 3, is discussed in detail in Section 6.2.

In Figure 6 (a), we observe that Design 1 is more load-balanced and efficient than Design 2 due to its better scheduling across fewer PEs when operating on highly sparse matrices. While this example is presented on a small scale, the performance gap between Designs 1 and 2 would become even more pronounced for larger matrices with very few nonzero elements. The key issue lies in load/store dependency stalls (or "bubbles") during scheduling. Since elements are assigned to PEs in a round-robin fashion and must respect load/store dependency cycles within each row, Design 1 produces a more compact schedule. By assigning more rows per PE, it can interleave elements from different rows to fill bubbles effectively. In contrast, Designs 2 and 3 lack sufficient elements to fill these bubbles, forcing them to pad with inefficient zeros and thereby reducing performance.

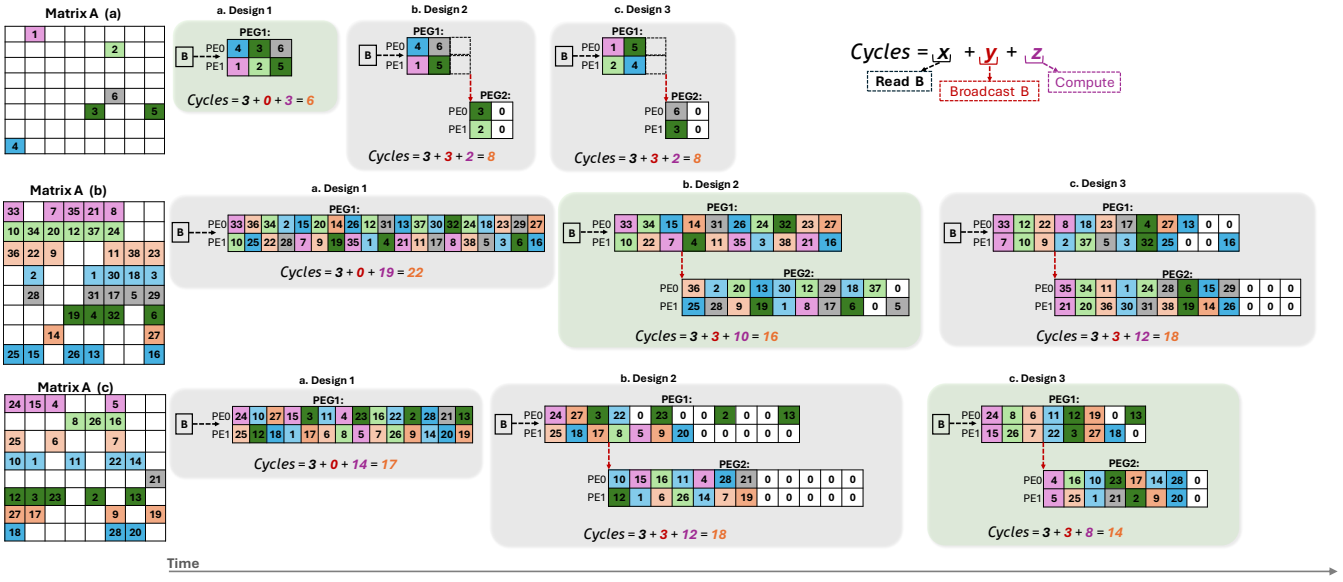

Figure 6: Timeline of applying three Misam designs to three sparse matrices with varying sparsity levels, illustrating that different designs can perform best depending on sparsity. The green shading highlights the fastest design for each matrix.

Conversely, for larger and denser matrices, each PE is assigned a greater number of rows containing more nonzero elements. This increases the likelihood that the scheduler can select independent nonzeros from different rows to mask sparsity-induced stalls (or bubbles) caused by load/store dependencies. For example, if a nonzero at time step t must be delayed until $t + 2$ due to row-level dependencies, the scheduler can fill time step $t + 1$ with a nonzero from another row mapped to the same PE. Such interleaving enables continuous execution across rows without violating dependency constraints. As illustrated in Figure 6(b), under these conditions, Design 2 produces a more efficient schedule than Design 1.

Ultimately, the choice between Design 1 and Design 2 depends on the matrix characteristics—particularly size, sparsity level, and the distribution of nonzeros. As matrix size increases, the set of candidate rows per PE grows, giving Design 2's scheduler greater flexibility to resolve scheduling gaps. This leads to more optimal execution with reduced idle time, higher PE utilization, and improved throughput. In contrast, for matrices with uniformly low nonzero density, where each row provides insufficient work for a large PE set, Design 1 is more efficient. These features serve as key decision factors in our model (Figure 4), guiding the selection of the design that delivers optimal performance for a given input.

3.2.3 ***Design 3:*** It shares its hardware configuration with Design 2 but alters the traversal and scheduling. Instead of processing matrix A in a column-wise manner, it adopts a row-wise traversal. As a result, elements are assigned to PEs based on the column index modulo the PE count (column_num%PE), shifting the load distribution logic to better accommodate irregular sparsity patterns.

This change is especially advantageous in scenarios where matrix A exhibits high load imbalance across rows. As shown in Figure 6, row-wise traversal can lead to a more compact schedule with fewer idle cycles ("bubbles") caused by load/store dependencies. Scheduling still respects a 2-cycle dependency constraint, but because elements from the same row are processed in order, and sparsity is irregular, the schedule interleaves more efficiently across PEs.

Design 3 is frequently selected by our decision tree when the A_nonzeroes and A_load_imbalance_row features signal high variation across rows. In such cases, traditional column-wise processing (Designs 1 and 2) often struggles with PE underutilization, whereas Design 3 maintains higher throughput by exploiting row-level irregularities. We illustrate this scenario in Figure 6(c), where Design 3 produces the most optimal schedule compared to the other designs. This example also highlights the necessity of Design 3, as the difference in scheduling quality between Designs 2 and 3 is substantial: Design 2 requires significant padding with inefficient zeros, whereas Design 3 generates a compact and performant execution schedule. As with all designs, effectiveness depends on matrix structure and sparsity profile. The decision tree encodes these dependencies and uses them to predict the most appropriate scheduling approach per workload.

3.2.4 ***Design 4:*** The design extends the baseline architecture to support SpGEMM, where both input matrices A and B are sparse. While the core architecture remains largely unchanged, several modifications are introduced to handle the sparsity of matrix B and enable compressed storage and computation. As with matrix A, the nonzero elements of matrix B are coalesced and encoded in a 64-bit format that includes the row index, column index, and value. The FIFO-based dataflow used between PEGs is retained, enabling pipelined and synchronized processing. An addition in this design is the introduction of a pointer list for matrix B, which tracks the number of nonzero elements to be read per tile, mirroring the mechanism used for scheduling matrix A.

Design 4 introduces a two-dimensional tiling strategy for matrix B, tailored to its sparse structure. Specifically, matrix B is tiled both row-wise and column-wise, with the tiling dimensions guided by its sparsity pattern and the available BRAM capacity. Since BRAM

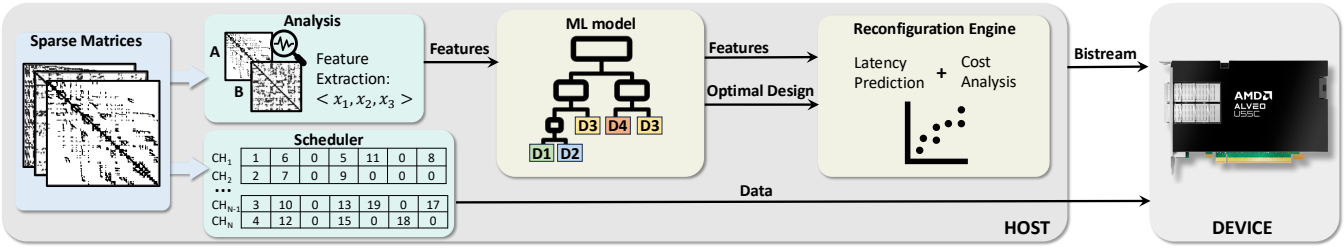

Figure 7: High-Level Overview of Misam

must now store coalesced sparse rows instead of dense vectors, the primary objective is to maximize the number of nonzero elements per tile while minimizing wasted space. To achieve this, the host performs a sparsity-aware packing analysis to determine an optimal row-wise tile size for B that fits efficiently within the BRAM limits. Furthermore, each BRAM row may pack multiple sparse rows of B, since each row contains relatively few nonzero elements. To track the start and end positions of each logical row of B within BRAM, metadata is stored in the PEG-local URAMs. At runtime, the column index of a nonzero in matrix A is used to index into this URAM mapping, retrieving the corresponding BRAM range where the matching row of B is stored.

After matrix B is tiled, matrix A is tiled column-wise, with each tile of A aligned to the corresponding rows of B required for computation. This tiling order is critical: the row-wise tiling of B constrains the column-wise tiling of A, since each tile of A must access a specific set of B rows that are already resident in BRAM. This dependency ensures high data locality and allows each PEG to operate on co-located tiles of A and B without incurring redundant BRAM accesses or overlapping data loads.

Analysis of the trained model reveals that Design 4 is selected for workloads where matrix B is highly sparse. In SpMM, HBM reads could fetch 16 FP32 values per access. In SpGEMM, however, B is stored in a 64-bit COO format, reducing each HBM read to 8 coalesced nonzero entries (each including row, column, and value). This effectively halves the read bandwidth for B, making compression worthwhile only when B's sparsity is high. For denser matrices, storing B in an uncompressed dense format offers better bandwidth utilization and lower latency. Our model is able to analyze this trade-off and make selections accordingly. Additional designs can be added for SpGEMM that have fine-grained support for highly sparse matrix patterns. Our framework can support as many designs as the user requires for their use case.

3.3 Reconfiguration Optimizations

While we have developed multiple architectural designs and a predictive model to select the most suitable one for a given input, an important question remains: how frequently should the FPGA be reconfigured to switch between designs? Figure 7 illustrates our proposed setup. On the host side, we first extract relevant features from the input matrix, which are then passed to a trained model that predicts the optimal design. Both the features and the selected design are then provided as inputs to the reconfiguration engine.

This engine includes a secondary model that estimates the expected latency for the predicted design, based on the matrix features and the current FPGA configuration. The engine also checks whether the required bitstream for the new design is already loaded. If not, it adds the bitstream reconfiguration time, which we extract empirically by analyzing timeline traces of bitstream loading on the FPGA, to the predicted latency. If the total estimated runtime (latency + reconfiguration time) is below a predefined reconfiguration threshold, then reconfiguration is deemed beneficial and is triggered. This threshold is user-defined and can be tuned to balance performance gains against reconfiguration overheads, depending on the deployment context. In our experiments, we set the threshold to 20%, meaning reconfiguration is triggered only when its overhead is less than 20% of the expected gain. Increasing the threshold allows more frequent reconfiguration by relaxing this constraint, while decreasing it results in stricter decisions, triggering reconfiguration only when substantial gains are expected.

Our system operates on a streaming execution model, where large matrices are divided into smaller tiles of varying sizes, typically ranging from 10k to 50k. These tiles are then streamed into the host incrementally. To prevent matrix dimension bias in our model, the tile sizes are selected randomly from within this range. The matrix A is partitioned in a tile-wise manner, while matrix B is partitioned row-wise, ensuring that the tiles are independent of each other. As a result, no partial reduction is needed within the tiles, eliminating the overhead of post-processing on the host side. Once the tile computation is completed on the FPGA, the results are streamed back to the host for storage.

As a result, reconfiguration granularity is defined at the tile level. If performance projections indicate that switching designs between tiles of the same matrix will yield a net latency benefit, reconfiguration is initiated by sending the new bitstream to the device. Otherwise, the host only sends the scheduled data. We will discuss in Section 6.1, the Xilinx U55C incurs significant overhead during reconfiguration, making tile-level reconfiguration suboptimal. In other scenarios where reconfiguration overhead is minimal, Misam provides the flexibility to perform reconfiguration at the tile level.

4 Experimental Setup

System: We prototype our proposed design for Xilinx Alveo U55C FPGA and optimized the design using Tapa [26] and RapidStream [27]. The estimated resource usage is summarized in Table 2. Notably, Designs 2 and 3 share the same bitstream, differing only in how the host system schedules access to HBM channels. To train our machine learning model using a dataset comprising thousands of matrices and corresponding performance metrics for each design, we developed a simulator for each design. The simulator is built using detailed profiling runs and HLS synthesis reports that capture hardware behavior like the number of cycles required to read data

Table 2: Resource estimation for Xilinx U55C.

Design Name	Resource Utilization					Freq (MHz)
	LUT	FF	BRAM	URAM	DSP	
Design 1	33.20%	23.61%	60.71%	26.67%	29.00%	284.02
Design 2 & 3	43.03%	30.35%	48.02%	40.00%	30.68%	290.3
Design 4	30.53%	21.15%	24.21%	30.00%	20.49%	287.4

from HBM, time spent in compute units, per-stage pipeline latency, and the time to write outputs to memory.

Baselines: We evaluate Misam against multiple baselines to assess its performance across a wide range of sparsity regimes. First, we compare against Trapezoid, using their cycle-accurate simulator across all of their proposed dataflow architectures. In addition, we benchmark Misam against two widely-used software libraries: cuSPARSE [67], executed on an NVIDIA RTX A6000 GPU, and Intel oneAPI Math Kernel Library (MKL) [47], run on an Intel Core i9-11980HK CPU with 32 GB of memory. RTX A6000 is a server-class GPU that features 84 streaming multiprocessors (SMs), each with 128 KB of L1 cache, and is equipped with 48 GB of GDDR6 memory (384 bits wide) that offers a bandwidth of 768 GB/s. These software libraries are evaluated on the same workload as Misam.

Energy: We estimate the energy consumption of Misam by profiling the generated bitstream and using Xilinx's xbutil tool to monitor power consumption. The measured power values are then combined with the kernel execution time to compute an estimate of the total energy consumed. For CPU-based computations on the Intel Core i9, we obtain power and energy usage programmatically via the PowerCap interface and Intel RAPL. For GPU-based execution on the NVIDIA RTX A6000, we use NVIDIA Management Library (NVML), a C-based API for monitoring NVIDIA GPU power consumption.

Table 3: Highly Sparse (HS) Matrices Used in Evaluation

Name	ID	Dens.	Rows	NNZ
p2p-Gnutella24	p2p	9.3e-5	26518	65369
sx-mathoverflow	sx	3.9e-4	24818	239978
ca-CondMat	cond	3.5e-4	23133	186936
Oregon-2	ore	3.5e-4	11806	65460
email-Enron	em	2.7e-4	36692	367662
opt1	opt	8.1e-3	15449	1930655
scircuit	sc	3.3e-5	170998	958936
gupta2	gup	1.1e-3	62064	4248286
sme3Db	sme	2.5e-3	29067	2081063
poisson3Da	poi	1.9e-3	13514	352762
wiki-RfA	wiki	1.5e-3	11380	188077
ca-AstroPh	astro	1.1e-3	18772	396160
msc10848	ms	1.0e-2	10848	1229776
ramage02	ram	1.0e-2	16830	2866352
cage12	cage	1.2e-4	130228	2032536
goodwin	good	6.0e-3	7320	324772

Workloads: To enable a fair comparison, we adopt a workload selection strategy similar to Trapezoid. Specifically, we evaluate 116 standalone matrix multiplication workloads, categorized as follows: 15 MS×D, 38 MS×MS, 12 HS×D, 36 HS×MS, and 12 HS×HS. Here, D refers to dense matrices, MS to moderately sparse matrices, and HS to highly sparse matrices.

For the D and MS categories, we use DNN-derived workloads from ResNet-50 and VGG-16 models. In the MS×D setting, the dense matrices have a fixed sequence length of 512, and the moderately sparse matrices are pruned ResNet-50 models. We apply structured pruning using STR [49], following the same methodology as Trapezoid, targeting weight densities of 0.1 and 0.2. Similarly, VGG-16 layers are pruned to the same density levels for the MS×MS combinations.

For combinations involving highly sparse (HS) inputs, we use the same matrices as Trapezoid from the SuiteSparse collection [12] shown in table 3. In the HS×D category, we evaluate the same 12 diverse matrices used in Trapezoid, multiplying each with a randomly generated dense matrix B having 512 columns—representative of applications such as solvers with multiple right-hand sides. In the HS×MS category, each of the 12 HS matrices is multiplied by three randomly generated sparse matrices (B) of 512 columns, with sparsity levels of 0.2, 0.4, and 0.6. Finally, for the HS×HS category, we evaluate A×A for each of the 12 matrices to model self-multiplication scenarios commonly seen in graph analytics and numerical solvers.

Datasets: To train and evaluate our decision tree model, we curated a dataset of 6,219 matrices covering a wide range of sparsity levels for both input matrices A and B, from 1% to 99%. To train and evaluate our latency predictor model, we curated a larger dataset comprising 19,000 matrices, which also includes the dataset used for training the decision model. Both datasets capture diverse sparsity patterns, enabling the model to learn meaningful correlations between matrix features and optimal dataflow designs. Highly sparse (HS) matrices were sourced from the SuiteSparse repository [12], while moderately sparse (MS) and dense (D) matrices were derived from deep learning workloads such as VGG, ResNet, MobileNet, and ImageNet. To generate MS matrices, we applied structured pruning techniques to intermediate layers of these models. The dataset included sufficient representation across all sparsity regimes to support the learning of thresholds for each architectural design. For training, we used a 70/30 train-evaluation split and performed k-fold cross-validation to assess accuracy and guard against overfitting.

5 Evaluations & Key Findings

This section presents the results of our experiments. We first evaluate the accuracy of the ML-based model selection and reconfiguration engine and their impact on Misam's performance. We then compare the performance of Misam designs against state-of-the-art sparse accelerators, followed by a detailed breakdown of Misam's components and their associated performance overheads.

5.1 ML Model Selection

Table 4: Geometric mean speedup of the optimal design over other designs for all workloads.

Speedup	Design 1	Design 2	Design 3
Design 1	1.00	1.35	1.35
Design 2	1.29	1.00	1.50
Design 3	1.28	1.81	1.00

The selection of the optimal design is highly dependent on the structure of the matrix. To quantify this observation, we calculate the speedup of the selected best-performing design over the other two for every matrix in our dataset. These results are summarized

in Table 4, which presents the geometric mean speedups. Design 4 is excluded from this analysis because its usage is explicitly determined by a clear decision in the model: it is mostly selected for highly sparse workloads, where no other design can compete. Conversely, for all other workloads, design 4 consistently underperforms.

Table 5: Confusion Matrix for the ML model

Predicted/Actual	Design 1	Design 2	Design 3	Design 4
Design 1	130	6	2	0
Design 2	4	257	28	0
Design 3	5	59	135	0
Design 4	0	1	0	617

Our ML model, which is responsible for optimal design selection, achieves 90% accuracy, translating to a geometric mean speedup of 1.31x for accurate predictions and a slight slowdown of 1.06× for mispredictions. The confusion matrix in table 5 highlights the model's ability to correctly classify most instances, with a high number of true positives along the diagonal. Misclassifications, indicated by off-diagonal elements, occur but are relatively few, reflecting the model's 90% accuracy. The predictions from the model are then routed to the reconfiguration engine, which is trained with a larger dataset and achieves an even higher accuracy of 97%. This secondary model acts as an additional layer of validation, ensuring that only predictions which result in performance improvements are accepted. In the following section, we quantify the end-to-end slowdown that occurs in the event of a failure in this secondary validation process. Overall, the lightweight inference of the model and the preprocessing stage yield substantial performance gains, while the optimal model size strikes a balance between accuracy and minimal inference time overhead.

5.2 Reconfiguration Engine Prediction

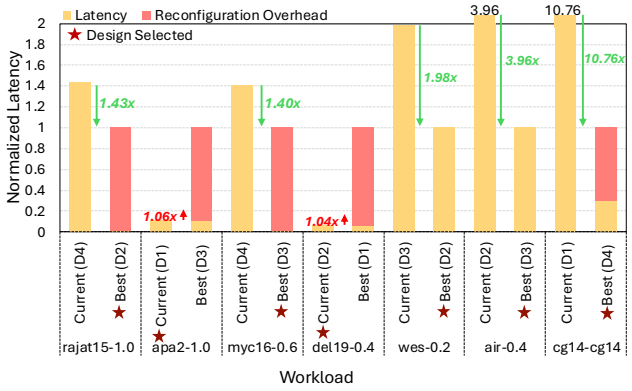

Figure 8: Reconfiguration overhead analysis on Xilinx U55C (lower is better).

We evaluate the performance of our reconfiguration engine using a diverse set of workloads from our dataset. Figure 8 compares executing each workload with the currently loaded bitstream ("current" bar) against the best-performing design for that workload. The design selected by the engine is marked with a star. We decompose the execution time into the design's intrinsic latency and the overhead incurred during design switching. Notably, transitions between design 2 and design 3 do not incur reconfiguration overhead, as they share the same bitstream. The Xilinx U55C shows significant reconfiguration overhead, discussed in more detail in Section 6.1.

Our analysis shows that reconfiguration often provides substantial speedups, especially for large matrices. For example, the cg15 workload (1.5M by 1.5M) achieves up to a 10.76× speedup, as the reconfiguration cost is amortized over tiled processing. In contrast, workloads like apa2 or del19 show minimal performance gain from switching designs, with the reconfiguration overhead outweighing the benefit. In these situations, the engine opts to retain the current design, resulting in a slight slowdown compared to the theoretical best. The engine achieves a geometric mean speedup of 2.74× where reconfiguration occurs and a slowdown of 1.02× when overhead outweighs performance gain.

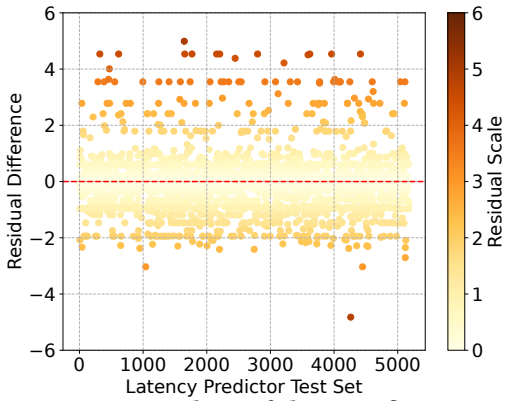

Figure 9: Accuracy analysis of the reconfiguration-engine predictor.

While the observed reconfiguration overhead stems from hardware limitations, not deficiencies in the engine, future FPGA platforms with reduced reconfiguration times could enable the engine to more aggressively select optimal designs. Additionally, users can configure reconfiguration times for designs to zero, allowing the engine to focus on design efficiency and always switch to the optimal bitstream. To further assess the accuracy of the engine's predictions, we examine residuals between predicted and actual latencies, shown in Figure 9. Our latency predictor achieves a mean absolute error (MAE) of 0.344 and an R^2 value of 0.978. The R^2 metric measures how well the predicted values explain the variance in the actual latencies, with a value close to 1 indicating high prediction accuracy. These results confirm the reliability of our model and support the engine's effectiveness.

5.3 Performance Gain

Figure 10 shows that Misam achieves average performance gains of 3.23×, 1.01×, and 5.84× over Trapezoid for HS×MS, MS×MS, and HS×D workloads, respectively. The most significant gains are observed in the HS×MS and HS×D workloads. Furthermore, Misam achieves average performance gains of 5.50×, 15.33×, and 20.27× over the CPU, and 1.37×, 4.48×, and 11.26× over the GPU for the HS×HS, HS×MS, and MS×MS workloads, respectively. As expected, GPUs excel in dense matrix multiplications due to their

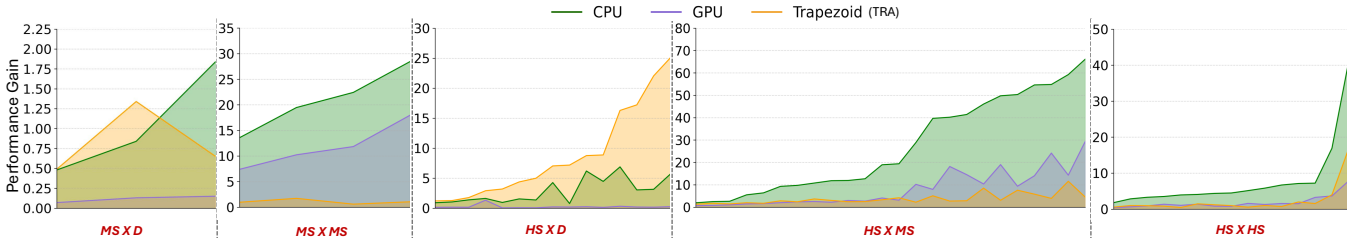

Figure 10: Performance Gain of Misam over CPU, GPU and Trapezoid on extended workload.

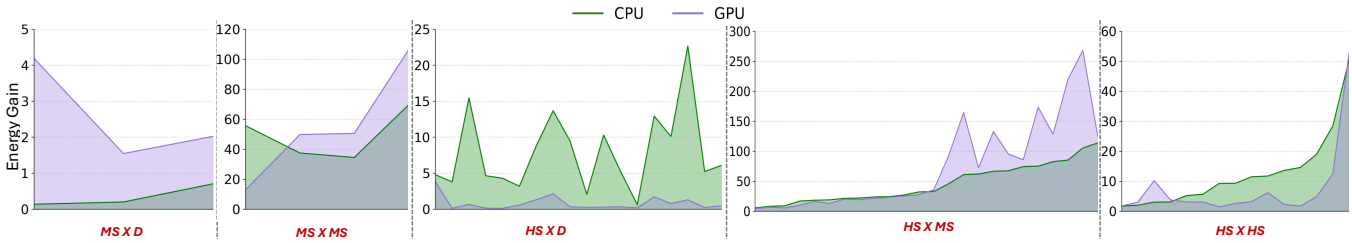

Figure 11: Energy Efficiency Gain of Misam over CPU and GPU on extended workload.

high-throughput architecture optimized for such workloads. Interestingly, for MS workloads, we observe a performance degradation, which could be due to pruning the layers such that it introduces a non-optimal sparsity structure for tensor cores. Both Trapezoid and Misam outperform the GPU on these workloads.

These performance improvements on highly sparse workloads stem from Misam's efficient computation pipelining and its adaptive tiling strategy for matrix B. In the case of HS × MS, some workloads require the second operand to be treated as highly sparse, while others benefit from treating it as dense. Similar pattern with MS × MS workloads. Through its model selection mechanism, Misam dynamically chooses between sparse and dense designs and benefits from improved performance. In contrast, other implementations lack such a runtime selection strategy and thus cannot exploit this advantage. We discuss how the model selection strategy can be adapted to CPU and GPU libraries in Section 6.3.

5.4 Energy Gain

We compare Misam's energy efficiency against both CPU and GPU baselines across different matrix categories. On average, Misam achieves significantly better energy efficiency over the CPU, with improvements of 14.94× (HS×HS), 47.24× (MS×MS), 33.96× (HS×MS), 6.08× (HS×D), and 5.51× (MS×D). Compared to the GPU, Misam still maintains substantial energy advantages in most cases—8.21×, 43.07×, and 39.86× for HS×HS, MS×MS, and HS×MS, respectively. However, for workloads involving dense matrices, such as HS×D and MS×D, the GPU's optimized dense operations lead to lower energy consumption than Misam, where we observe 0.47× and 0.27× efficiency ratios, respectively. The trapezoid simulator does not provide energy consumption metrics, so we were unable to include it in this comparison.

Figure 11 shows the energy efficiency of Misam compared to CPU and GPU implementations across various workloads. For highly sparse inputs, Misam demonstrates significantly higher energy gains over all three baselines. This can be attributed to two factors: (1) its substantial performance advantage, which reduces execution time, and (2) its implementation on an FPGA platform, which is inherently more power and energy-efficient than general-purpose processors and GPUs. These characteristics make Misam particularly well-suited for workloads dominated by extreme sparsity.

As workloads become denser, the energy advantage of Misam over GPUs diminishes. This is expected, as GPUs are high-throughput architectures optimized for dense computations. While GPUs may not match the energy efficiency of FPGAs, their ability to complete dense workloads rapidly offsets this difference. Nevertheless, the key takeaway is Misam's versatility: although it may not always outperform the best specialized hardware for a given workload, it consistently delivers competitive results across a wide spectrum of sparsity patterns, offering a balanced and adaptable solution.

5.5 Misam's Performance Breakdown

Figure 12 presents a normalized performance comparison of Misam, CPU, GPU, and Trapezoid across a representative set of matrices. Each subplot is normalized to the best-performing accelerator in its respective category. For Misam, we further break down its performance into three components: preprocessing latency, which is the time required to extract features from the input matrices; inference latency, which includes both the inference time of the machine learning model and the reconfiguration engine; and finally, the hardware execution latency of Misam.

As shown in the plots, both preprocessing and inference latencies make up a very small portion of the total execution time. Specifically, the ML model inference takes only 0.002 ms, and the reconfiguration engine adds just 0.005 ms, together accounting for about 0.1% of Misam's total execution time, making the inference latency line barely visible in most plots. This efficiency stems from our lightweight 6 KB model, which is pruned and uses only the top four features. Moreover, instead of using a Python inference library, which introduced a massive slowdown due to its lack of optimization for small models, we implemented a custom inference function by unrolling the decision logic. Inference time was derived by averaging results over 1,800 test cases.

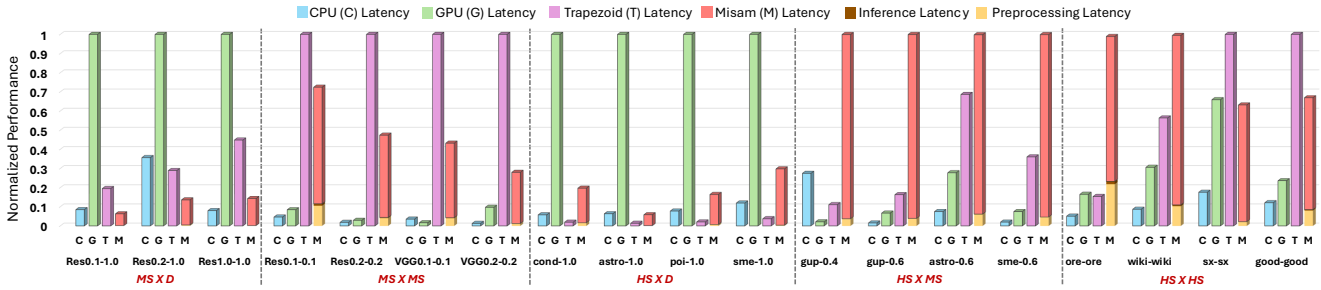

Figure 12: End-to-End performance comparison on representative workloads (normalized to the best accelerator).

In contrast, preprocessing latency is slightly higher, accounting for approximately 2% of Misam's overall performance, and it varies depending on matrix size. For smaller matrices, such as those in MS×D and HS×D, the overhead from preprocessing is negligible. This is because matrix B, although dense, is small in size compared to the very large HS matrices. Importantly, our most critical feature is derived from matrix B, and among the top four features, this one is the most time-consuming to extract.

6 Discussions

This section explores how Misam leverages hardware reconfigurability and ML-based decision-making to adapt to diverse workloads. We first analyze the overheads of full and partial FPGA reconfiguration, then demonstrate how resource-efficient bitstreams enable multi-tenant execution and improved hardware utilization. Finally, we show how Misam's low-overhead, high-accuracy components generalize across architectures and sparsity regimes.

6.1 Reconfiguration Time

Figure 8 illustrates that full bitstream reconfiguration on the Xilinx U55C FPGA typically takes 3–4 seconds , with bitstream sizes of 50–80 MB transferred over a PCIe Gen4 x8 interface at 6.4 GB/s. Profiling indicates that the primary contributor to this overhead is the FPGA fabric programming phase, which dominates the total reconfiguration time. To better understand this bottleneck, we evaluated multiple reconfiguration methods, including the Vivado GUI [98], OpenCL API [46], and XRT command-line interface [99]. All approaches yielded similar reconfiguration times, suggesting that the limiting factor is the bitstream transfer and device programming process itself, rather than the specific software stack used.

To address this challenge, we explored partial reconfiguration, where only a dynamic region of the FPGA is updated while the static region remains in place. For small dynamic regions, this approach reduced reconfiguration time to several hundred milliseconds. However, as the size of the dynamic region increases, the time savings diminish and the overhead approaches that of a full reconfiguration. In our case, the architecture did not naturally support very small dynamic regions, so we chose not to pursue partial reconfiguration further in this project. Nonetheless, our experiments with minimal dynamic regions demonstrated substantial performance improvements, highlighting that careful partitioning could make this a promising research direction for future work.

Looking ahead, designing architectures with small, modular dynamic regions could further exploit the benefits of partial reconfiguration. However, for highly latency-sensitive applications, even these reduced reconfiguration times may still be a limiting factor for end-to-end performance. An alternative direction is the use of Coarse-Grained Reconfigurable Architectures (CGRAs), which can achieve reconfiguration times in the microsecond to millisecond range, though at the cost of more complex and time-intensive compilation [58]. The Misam reconfiguration engine could be adapted for CGRA platforms, where intelligent runtime decisions are needed to balance reconfiguration overhead and performance gains. Given the low inference latency of our ML-based engine, further reducing reconfiguration time in such architectures could unlock additional performance benefits, especially in dynamic and adaptive computing scenarios.

6.2 Efficient Hardware Utilization

While ASIC accelerators like Trapezoid offer high-performance execution for sparse matrix workloads through customized dataflows, they incur significant area overhead and hardware underutilization due to their fixed-function nature. To support a wide range of dataflows optimized for different sparsity regimes, Trapezoid must integrate hardware blocks for all supported configurations. This results in area costs of $69.7mm^2$, $57.6mm^2$, and $51.2mm^2$ for different configurations. When switching to a dataflow that only requires the smaller configurations, up to 26.5% of the chip area becomes idle, yet still contributes to silicon cost and power leakage.

FPGAs address this inefficiency through reconfigurability and dynamic resource management. Our FPGA-based designs show diverse and compact footprints: for example, Design 1 uses only 33.2% of LUTs, 29.0% of DSPs, and 60.7% of BRAMs, while Design 4 uses just 24.21% of BRAMs. This enables a unique advantage: multi-tenant execution, where multiple independent bitstreams run concurrently on different regions of the FPGA. Based on resource usage profiles, we estimate that the FPGA can accommodate 1 instance of Design 1, 2 instances of Design 2 or 3, and up to 2 instances of Design 4 in parallel—each executing a different workload. Moreover, once a design is placed, any remaining FPGA capacity can be used to co-locate additional workloads, including other bitstreams with compatible resource footprints, provided their cumulative resource usage stays within the device's limits. This dynamic partitioning allows for full exploitation of LUTs, BRAMs, URAMs, and DSPs, dramatically improving effective hardware utilization. In contrast, ASICs like Trapezoid are over-provisioned for generality, paying the cost in underutilized silicon when running narrower workloads.

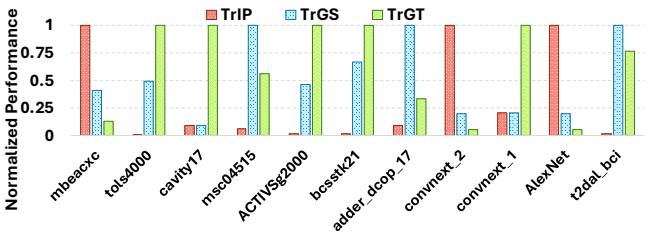

Figure 13: Performance comparison of Trapezoid's dataflows (normalized to the best dataflow).

Thus, FPGAs not only reduce area waste through specialization but also unlock higher throughput per chip through spatial multi-tenancy, making them especially advantageous in workload-diverse or multi-user environments.

6.3 Adaptability of Misam Components

Misam is built around two core components: a machine learning model that predicts the optimal dataflow scheme, and a reconfiguration engine that determines whether switching to a different dataflow is beneficial. One of the primary objectives of Misam was to automate dataflow selection—an issue that prior architectures, such as Trapezoid, leave unaddressed. While Trapezoid supports three specialized dataflows for SpGEMM and SpMM workloads, it does not provide a mechanism for selecting among them. Figure 13 illustrates these dataflows and highlights that their optimality varies across workloads, with no clear method to guide the choice. For instance, different layers of ConvNeXt benefit from different dataflows. This underscores the need for a systematic dataflow selection mechanism.

By training Misam's ML model on Trapezoid's dataflows, we achieved 92% prediction accuracy and observed up to a 15.8× speedup when the optimal dataflow was chosen. Notably, the ML inference overhead is just 0.1% of total execution time (geometric mean), ensuring minimal performance impact. Misam is also extensible to heterogeneous environments involving CPUs, GPUs, FPGAs, and ASICs. Based on performance trends across different sparsity regimes, the model can route workloads to the most suitable device; for instance, it correctly routes workloads to the GPU when it consistently offers better performance.

Beyond ASIC accelerators such as Trapezoid, Misam can be generally extended to scenarios where the optimal configuration depends on the matrix's sparsity pattern. Acamar [5], for instance, is a reconfigurable scientific computing accelerator that currently uses heuristics to predict which solver to use based on sparsity. Acamar's reconfiguration unit relies on heuristics for reconfiguration decisions, but lacks quantified accuracy or the overhead of this approach. The combination of our low-overhead and high-accuracy components makes it a strong candidate for deployment in such domains, offering a data-driven alternative to manual heuristics.

A core objective of Misam is to design ML-based components that remain adaptable across diverse scenarios. Its low preprocessing and inference overhead, combined with high prediction accuracy, make it a strong alternative to heuristic methods, which often require domain expertise and manual updates. Unlike fixed heuristics, Misam can be retrained as workloads evolve, often within minutes for reasonably sized datasets. Besides, insights from trained models can inform the design of new heuristics, bridging the gap between manual rule design and adaptive learning-based optimization.

7 Related Work

The related studies discussed throughout the paper are just a few among various recent works focused on sparse problems [1, 6, 7, 14, 15, 17, 20, 22, 23, 25, 33, 34, 38, 40, 41, 43, 44, 48, 50, 51, 53, 57, 60–62, 66, 70, 76, 78, 80, 82, 84, 85, 87, 90, 93–95, 97, 100, 101, 103, 105]. More specifically, there have been numerous sparse accelerators proposed that optimize for fixed dataflows [52, 68, 69, 72, 73, 89, 109]. Recent designs aim for flexibility by avoiding static dataflows [31, 54, 63, 66, 91, 102], but they typically depend on ad hoc heuristics, lacking a general, portable mechanism for dataflow selection. Misam aims to bridge this gap by offering a general, portable framework for dataflow selection that supports both existing and future sparse matrix multiplication accelerators.

While numerous works have focused on accelerating sparse linear algebra kernels on FPGAs [8, 13, 17, 24, 32, 39, 41, 42, 79, 86] and others have leveraged FPGA reconfigurability for various applications [5, 74, 75], Misam introduces a novel synthesis of these two areas. To our knowledge, it is the first framework to feature an intelligent reconfiguration engine that analyzes an input matrix's sparsity pattern to perform a cost-benefit analysis, determining at runtime whether the performance gain from switching configurations justifies the overhead.

8 Conclusions

Misam advances sparse matrix–matrix multiplication by unifying learned dataflow selection with targeted reconfigurability. It formulates dataflow choice as an ML classification problem over matrix features, using a lightweight decision tree to pick among distinct dataflow/hardware configurations rather than relying on ad-hoc rules or offline profiling. This selector is paired with an intelligent reconfiguration engine that performs an explicit cost–benefit analysis, considering both predicted latency and bitstream-switch overhead, to decide when a design switch is worthwhile. Together with a suite of specialized FPGA designs that expose complementary strengths across sparsity regimes, the framework provides a principled, runtime path to adaptability without overprovisioned hardware. The framework also exposes practical knobs to adapt decisions to deployment goals, while leaving room for future directions such as on-device inference or finer-grained reconfiguration. Its selector and reconfiguration logic are deployment-agnostic and can be integrated with CPUs, GPUs, FPGAs, or CGRAS. In short, Misam offers a general, portable mechanism for dataflow selection and runtime reconfiguration that can slot into existing or future sparse acceleration stacks.

Acknowledgments

Sanjali Yadav is supported by Bahar Asgari's US Department of Energy (DoE), Office of Science under the ASCR ECRP, Award DE-SC0024079. Amirmahdi Namjoo is supported by Bahar Asgari's NSF PPoSS program award, under Award Number 2316177.

References

[1] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Sung-Kyu Lim, Hyesoon Kim, et al. 2021. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *HPCA*. 908–920.

[2] M. R. Ashuthosh, Santosh Krishna, Vishvas Sudarshan, Srinivasan Subramaniyan, and Madhura Purnaprajna. 2022. MAPPARAT: A Resource Constrained FPGA-Based Accelerator for Sparse-Dense Matrix Multiplication. In *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*. 102–107. https://doi.org/10.1109/VLSID2022.2022.00031

[3] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[4] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-Aware Inner Product Processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 116–128. https://doi.org/10.1109/PACT52795.2021.00016

[5] Ubaid Bakhtiar, Helya Hosseini, and Bahar Asgari. 2024. Acamar: A Dynamically Reconfigurable Scientific Computing Accelerator for Robust Convergence and Minimal Resource Underutilization. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1601–1616. https://doi.org/10.1109/MICRO61859.2024.00117

[6] Ubaid Bakhtiar, Helya Hosseini, and Bahar Asgari. 2024. Acamar: A dynamically reconfigurable scientific computing accelerator for robust convergence and minimal resource underutilization. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1601–1616.

[7] Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Pipirima: Predicting Patterns in Sparsity to Accelerate Matrix Algebra. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference (DAC)*.

[8] Erfan Bank Tavakoli, Michael Riera, Masudul Hassan Quraishi, and Fengbo Ren. 2024. FSpGEMM: A Framework for Accelerating Sparse General Matrix–Matrix Multiplication Using Gustavson's Algorithm on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 4 (2024), 633–644. https://doi.org/10.1109/TVLSI.2024.3355499

[9] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: design, implementation, and applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[10] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[12] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[13] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2020. FBLAS: Streaming linear algebra on FPGA. In *SC20: International conference for high performance computing, networking, storage and analysis*. IEEE, 1–13.

[14] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1110–1123.

[15] Matthew Denton and Herman Schmit. 2022. Direct Spatial Implementation of Sparse Matrix Multipliers for Reservoir Computing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–11.

[16] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P Bosma, Zongwei Zhou, Tao Wang, Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2022. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 5547–5569. https://proceedings.mlr.press/v162/du22c.html

[17] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 54–64.

[18] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the Lottery: Making All Tickets Winners. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 2943–2952. https://proceedings.mlr.press/v119/evci20a.html

[19] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 278–291. https://doi.org/10.1145/3437801.3441599

[20] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. MeNDA: a near-memory multi-way merge solution for sparse transposition and dataflows. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 245–258.

[21] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 10323–10337. https://proceedings.mlr.press/v202/frantar23a.html

[22] Armin Gerami and Bahar Asgari. 2024. Gust: Graph edge-coloring utilization for accelerating sparse matrix vector multiplication. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 127–141.

[23] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[24] Juan Gonzalez and Rafael C Núñez. 2009. LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012042.

[25] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. 2022. CANDLES: Channel-Aware Novel Dataflow-Microarchitecture Co-Design for Low Energy Sparse Neural Network Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 876–891.

[26] 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. https://doi.org/10.1145/3609335

[27] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2023. RapidStream 2.0: Automated Parallel Implementation of Latency–Insensitive FPGA Designs Through Partial Reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 59 (Sept. 2023), 30 pages. https://doi.org/10.1145/3593025

[28] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic network surgery for efficient dnns. *Advances in neural information processing systems* 29.

[29] Kathleen E. Hamilton, Catherine D. Schuman, Steven R. Young, Ryan S. Bennink, Neena Imam, and Travis S. Humble. 2020. Accelerating Scientific Computing in the Post-Moore's Era. *ACM Trans. Parallel Comput.* 7, 1, Article 6 (March 2020), 31 pages. https://doi.org/10.1145/3380940

[30] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems* 28 (2015).

[31] Muhammad Abdullah Hanif, Rachmad Vidya Wicaksana Putra, Muhammad Tanvir, Rehan Hafiz, Semeen Rehman, and Muhammad Shafique. 2018. MPNA: A Massively-Parallel Neural Array Accelerator with Dataflow Optimization for Convolutional Neural Networks. arXiv:1810.12910 [cs.DC] https://arxiv.org/abs/1810.12910

[32] Xiaochen Hao, Mingzhe Zhang, Ce Sun, Zhuofu Tao, Hongbo Rong, Yu Zhang, Lei He, Eric Petit, Wenguang Chen, and Yun Liang. 2023. Lasa: Abstraction and specialization for productive and performant linear algebra on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 34–40.

[33] Xin He, Kuan-Yu Chen, Siying Feng, Hun-Seok Kim, David Blaauw, Ronald Dreslinski, and Trevor Mudge. 2022. Squaring the circle: Executing Sparse Matrix Computations on FlexTPU—A TPU-Like Processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 148–159.

[34] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[35] Zifan He, Linghao Song, Robert F Lucas, and Jason Cong. 2024. LevelST: Stream-based accelerator for sparse triangular solver. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 67–77.

[36] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[37] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 84–96. https://doi.org/10.1109/HPCA51647.2021.00017

[38] Helya Hosseini, Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Segin: Synergistically Enabling Fine-Grained Multi-Tenant and Resource Optimized SpMV. *IEEE Computer Architecture Letters* (2025).

[39] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[40] Chao-Tsung Huang. 2021. Ringcnn: Exploiting algebraically-sparse ring tensors for energy-efficient cnn-based computational imaging. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1096–1109.

[41] Abhishek Kumar Jain, Hossein Omidian, Henri Fraisse, Mansimran Benipal, Lisa Liu, and Dinesh Gaitonde. 2020. A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas. In *2020 30th International conference on field-programmable logic and applications (FPL)*. IEEE, 127–132.

[42] Hanchen Jin, Zichao Yue, Zhongyuan Zhao, Yixiao Du, Chenhui Deng, Nitish Srivastava, and Zhiru Zhang. 2024. Vesper: A Versatile Sparse Linear Algebra Accelerator With Configurable Compute Patterns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[43] Donghyeon Joo, Ramyad Hadidi, Soheil Feizi, and Bahar Asgari. 2024. Endor: Hardware-Friendly Sparse Format for Offloaded LLM Inference. *arXiv preprint arXiv:2406.11674* (2024).

[44] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *MICRO*. ACM, 600–614.

[45] Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H Chi. 2021. Learning to embed categorical features without embedding tables for recommendation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 840–850.

[46] Khronos Group. 2008. *OpenCL - The open standard for parallel programming of heterogeneous systems*. https://www.khronos.org/opencl/ Version 1.0 and later.

[47] Mariia Krainiuk, Mehdi Goli, and Vincent R Pascuzzi. 2021. oneAPI open-source math library interface. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 22–32.

[48] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.

[49] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. 2020. Soft Threshold Weight Reparameterization for Learnable Sparsity. arXiv:2002.03231 [cs.LG] https://arxiv.org/abs/2002.03231

[50] Gang Li, Weixiang Xu, Zhuoran Song, Naifeng Jing, Jian Cheng, and Xiaoyao Liang. 2022. Ristretto: An Atomized Processing Architecture for Sparsity-Condensed Stream Flow in CNN. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1434–1450.

[51] Shiyu Li, Edward Hanson, Xuehai Qian, Hai" Helen" Li, and Yiran Chen. 2021. ESCALATE: Boosting the efficiency of sparse CNN accelerator with kernel decomposition. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 992–1004.

[52] Shiqing Li, Shuo Huai, and Weichen Liu. 2023. An efficient gustavson-based sparse matrix–matrix multiplication accelerator on embedded FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4671–4680.

[53] Shiqing Li, Di Liu, and Weichen Liu. 2021. Optimized Data Reuse via Reordering for Sparse Matrix-Vector Multiplication on FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[54] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[55] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1754–1763.

[56] Ji Lin, Chuang Gan, and Song Han. 2019. TSM: Temporal Shift Module for Efficient Video Understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[57] Bowen Liu and Dajiang Liu. 2023. Towards High-Bandwidth-Utilization SpMV on FPGAs via Partial Vector Duplication. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 33–38.

[58] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* 52, 6, Article 118 (Oct. 2019), 39 pages. https://doi.org/10.1145/3357375

[59] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*. 2736–2744.

[60] Hang Lu, Liang Chang, Chenglong Li, Zixuan Zhu, Shengjian Lu, Yanhuan Liu, and Mingzhe Zhang. 2021. Distilling bit-level sparsity parallelism for general purpose deep learning acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 963–976.

[61] Kai Lu, Zhaoshi Li, Leibo Liu, Jiawei Wang, Shouyi Yin, and Shaojun Wei. 2019. Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[62] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[63] Shengbai Luo, Bo Wang, Yihao Shi, Xueyi Zhang, Qingshan Xue, and Sheng Ma. 2024. Sparm: A Sparse Matrix Multiplication Accelerator Supporting Multiple Dataflows. In *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 122–130. https://doi.org/10.1109/ASAP61560.2024.00034

[64] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. 2020. Proving the Lottery Ticket Hypothesis: Pruning is All You Need. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 6682–6691. https://proceedings.mlr.press/v119/malach20a.html

[65] Srđan Milaković, Oguz Selvitopi, Israt Nisa, Zoran Budimlić, and Aydin Buluc. 2023. Parallel Algorithms for Masked Sparse Matrix-Matrix Products. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3545008.3545048

[66] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (, Vancouver, BC, Canada,) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[67] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*, Vol. 12.

[68] 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 the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106.

[69] G Noble, S Nalesh, and S Kala. 2023. MOSCON: Modified Outer Product based Sparse Matrix-Matrix Multiplication Accelerator with Configurable Tiles. In *2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID)*. IEEE, 264–269.

[70] Nebil Ozer, Gregory Kollmer, Ramyad Hadidi, and Bahar Asgari. 2025. La Superba: Leveraging a Self-Comparison Method to Understand the Performance Benefits of Sparse Acceleration Optimizations. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–12.

[71] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[72] 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 *HPCA*. IEEE, 724–736.

[73] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. 2024. A row decomposition-based approach for sparse matrix multiplication on GPUs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 377–389.

[74] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. 2011. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4, 4 (2011), 1–24.

[75] Luca Pezzarossa, Andreas Toftegaard Kristensen, Martin Schoeberl, and Jens Sparsø. 2018. Using dynamic partial reconfiguration of FPGAs in real-Time systems. *Microprocessors and Microsystems* 61 (2018), 198–206.

[76] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E Moon, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Extending sparse tensor accelerators to support multiple compression formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1014–1024.

[77] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[78] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *HPCA*.

[79] Manoj B Rajashekar, Xingyu Tian, and Zhenman Fang. 2024. HiSpMV: Hybrid row distribution and vector buffering for imbalanced SpMV acceleration on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 154–164.

[80] Dheeraj Ramchandani, Bahar Asgari, and Hyesoon Kim. 2023. Spica: Exploring FPGA Optimizations to Enable an Efficient SpMV Implementation for Computations at Edge. In *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE, 36–42.

[81] Nikhil Rao, Robert Nowak, Christopher Cox, and Timothy Rogers. 2015. Classification with the sparse group lasso. *IEEE Transactions on Signal Processing* 64, 2 (2015), 448–463.

[82] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 347–358.

[83] Victor Sanh, Thomas Wolf, and Alexander Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in neural information processing systems* 33 (2020), 20378–20389.

[84] Björn Sigurbergsson, Tom Hogervorst, Tong Dong Qiu, and Razvan Nane. 2019. Sparstition: a partitioning scheme for large-scale sparse matrix vector multiplication on FPGA. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 51–58.

[85] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE design automation conference*. 211–216.

[86] 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 *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3490422.3502357

[87] 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 *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[88] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[89] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[90] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[91] Jianming Tong, Anirudh Itagi, Prasanth Chatarasi, and Tushar Krishna. 2024. FEATHER: A Reconfigurable Accelerator with Data Reordering Support for Low-Cost On-Chip Dataflow Switching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 198–214. https://doi.org/10.1109/ISCA59077.2024.00024

[92] John Towns, Tim Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science and Engineering* 16, 5 (Sept. 2014), 62–74. https://doi.org/10.1109/MCSE.2014.80

[93] Chaithanya Krishna Vadlamudi and Bahar Asgari. 2024. Electra: Eliminating the Ineffectual Computations on Bitmap Compressed Matrices. *IEEE Computer Architecture Letters* (2024).

[94] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[95] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[96] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. 2023. HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3613424.3623786

[97] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[98] Xilinx, Inc. 2021. *Vivado Design Suite User Guide: Using the Vivado IDE*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug893-vivado-ide.pdf UG893 (v2020.2).

[99] Xilinx, Inc. 2021. *Xilinx Runtime (XRT) Release Notes*. https://xilinx.github.io/XRT/master/html/index.html UG1451 (v2021.1).

[100] Sanjali Yadav and Bahar Asgari. 2025. DynaFlow: An ML Framework for Dynamic Dataflow Selection in SpGEMM accelerators. *IEEE Computer Architecture Letters* (2025).

[101] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2021. Spzip: architectural support for effective data compression in irregular applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1069–1082.

[102] Y. Yang, J. S. Emer, and D. Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 931–945. https://doi.org/10.1109/ISCA59077.2024.00072

[103] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 744–762.

[104] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[105] Shulin Zeng, Yujun Lin, Shuang Liang, Junlong Kang, Dongliang Xie, Yi Shan, Song Han, Yu Wang, and Huazhong Yang. 2019. A fine-grained sparse accelerator for multi-precision DNN. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 185–185.

[106] Chen Zhang, Yang Wang, Zhiqiang Xie, Cong Guo, Yunxin Liu, Jingwen Leng, Zhigang Ji, Yuan Xie, and Ru Huang. 2025. DSTC: Dual-Side Sparse Tensor Core for DNNs Acceleration on Modern GPU Architectures. *IEEE Trans. Comput.* 74, 2 (Feb. 2025), 341–355. https://doi.org/10.1109/TC.2024.3475814

[107] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[108] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 261–274. https://doi.org/10.1109/HPCA47549.2020.00030

[109] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[110] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.