# MAD-HiSpMV: Matrix Adaptive Design with Hybrid Row Distribution for Imbalanced SpMV Acceleration on FPGAs

MANOJ B. RAJASHEKAR, AKHIL R. BARANWAL, XINGYU TIAN, and ZHENMAN FANG, Simon Fraser University, Canada

Sparse matrix-vector multiplication (SpMV) is fundamental in numerous applications such as scientific computing, machine learning (ML), and graph analytics. While recent studies have made tremendous progress in accelerating SpMV on HBM-equipped FPGAs, there are still multiple remaining challenges to accelerate imbalanced SpMV where the distribution of nonzeros in the sparse matrix is imbalanced across different rows. These include (1) imbalanced workload distribution among the parallel processing elements (PEs), (2) long-distance dependency for floating-point accumulation on the output vector, (3) a new bottleneck due to the often-overlooked dense vectors' off-chip access after the SpMV acceleration, and (4) sub-optimal performance of generic accelerators for various types of sparse matrices. (5) Additionally, ML workloads often consist of both SpMV and general matrix-vector multiplication (GeMV), which suffer from kernel switching inefficiencies.

To address those challenges, we propose MAD-HiSpMV to accelerate imbalanced SpMV on HBM-equipped FPGAs with the following novel solutions: (1) a hybrid row distribution network to enable both inter-row and intra-row distribution for better balance, (2) a fully pipelined floating-point accumulation on the output vector using a combination of an adder chain and register-based circular buffer, (3) matrix adaptive design configurations generated by our automation framework via design space exploration (DSE) to maximize performance for the given matrix, and (4) a GeMV overlay built into the same kernel for efficient acceleration of mixed workloads. Experimental results demonstrate that the DSE-picked configuration of MAD-HiSpMV achieves a geomean speedup of 1.3x (up to 2.12x) for the SpMV benchmark matrices and achieves a geomean 1.15x (up to 1.54x) better performance per watt, when compared to state-of-the-art generic designs. For the SpMV benchmark matrices, compared to Intel MKL running on a 24-core Xeon Silver 4214 CPU, MAD-HiSpMV achieves a geomean speedup of 8.80x. Compared to cuSparse running on an Nvidia GTX 1080ti GPU, MAD-HiSpMV achieves a geomean of 2.57x better performance per watt. Additionally, a GeMV overlay built into MAD-HiSpMV achieves a peak throughput of 156.7 GFLOPS, which is 2.64x better than the Vitis L2 GeMV benchmark on U280, and performs 2.7x better for an end-to-end mixed workload, when compared to Intel MKL running on a 24-core Xeon Silver 4214 CPU. MAD-HiSpMV is available at https://github.com/SFU-HiAccel/HiSpMV.

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

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

**ACM Reference Format:**

Manoj B. Rajashekar, Akhil R. Baranwal, Xingyu Tian, and Zhenman Fang. 2025. MAD-HiSpMV: Matrix Adaptive Design with Hybrid Row Distribution for Imbalanced SpMV Acceleration on FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (October 2025), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Manoj B. Rajashekar, mba151@sfu.ca; Akhil R. Baranwal, akhil_baranwal@sfu.ca; Xingyu Tian, xingyu_tian@sfu.ca; Zhenman Fang, zhenman@sfu.ca, Simon Fraser University, Burnaby, BC, Canada.

## 1 Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental mathematical operation used in various fields, including scientific computing [9, 15], circuit simulation [12], machine learning [14, 18, 19, 39, 40], and graph analytics [4, 25]. It mainly involves multiplying a sparse matrix (with many zero entries) by a dense vector, resulting in a new dense vector. More specifically, the SpMV operation is described in Equation 1.

$$\vec{y} = \alpha.\mathbf{A} \times \vec{x} + \beta.\vec{y} \tag{1}$$

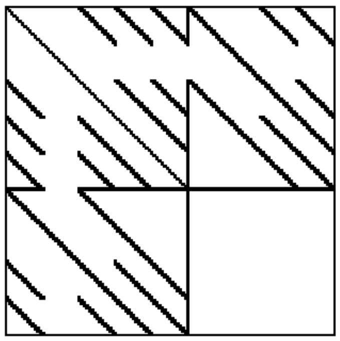
where $\mathbf{A}$ is a sparse matrix, $\vec{x}$ and $\vec{y}$ are dense vectors, $\alpha$ and $\beta$ are scalar constants. Since the sparse matrix has no data reuse and has irregular distribution of nonzero elements (i.e., random memory access), it poses great challenges in accelerating SpMV on FPGAs.

**Algorithm 1** Tiled SpMV acceleration w/ cyclic row-wise partition

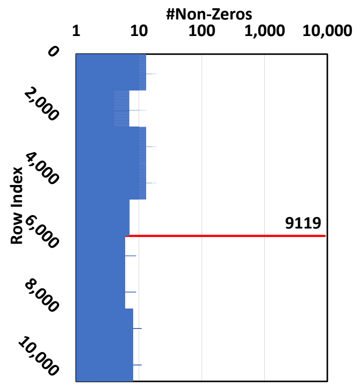
```
1: for ($r = 0; r < R; r = r + R_t$) do
2:     $y\_Ax[R_t] \leftarrow \vec{0}$                              ▷ buffer temporary output $\overrightarrow{y\_Ax}$
3:     for ($p = 0; p < P; p++$) do in parallel                  ▷ $P$ = #PEs
4:         for ($c = 0; c < C; c = c + C_t$) do
5:             $buf\_x[C_t] \leftarrow x[c : c + C_t)$                  ▷ load $\vec{x}$ buffer
6:             for all ($r\%P = p \mid a_{rc} \in \mathbf{A} \mid a_{rc} \neq 0$) do      ▷ stream $a_{rc}$
7:                 $y\_Ax[r\%R_t] += a_{rc} * buf\_x[c\%C_t]$           ▷ $\mathbf{A} \times \vec{x}$
8:     for ($i = r; i < r + R_t; i++$) do                          ▷ stream & compute $\overrightarrow{y\_out}$
9:         $y\_out[i] \leftarrow \alpha * y\_Ax[i\%R_t] + \beta * y\_in[i]$
```

Recent studies [11, 24, 34] have made great progress in harnessing HBM-based FPGAs to overcome those memory-bound challenges. A common processing pattern in these approaches involves three key techniques, as illustrated in Algorithm 1. Assume the sparse matrix $\mathbf{A}$ has $R$ rows and $C$ columns, and each time one tile of $\mathbf{A}$ is processed by $P$ number of processing elements (PEs): $R_t$ and $C_t$ are the tile sizes along rows and columns, respectively. First, nonzero elements within a tile of $\mathbf{A}$ are streamed into $P$ PEs from multiple HBM channels; line 6 in Algorithm 1 traverses all the $(r, c)$ indices that have a nonzero in the $\mathbf{A}$ tile. A distinct set of rows is usually cyclically assigned to each PE (and each HBM channel), with the aim of a more balanced workload partition. Second, the input vector $\vec{x}$ is tiled and buffered on-chip as $buf\_x[C_t]$ (size of $C_t$, line 5 in Algorithm 1), and the temporary output vector $y\_Ax[R_t]$ (size of $R_t$, line 2) for computing $\mathbf{A} \times \vec{x}$ (a tile) is processed on-chip (line 7), which limit the random accesses to on-chip memory only. The input vector buffer $buf\_x[C_t]$ is either duplicated (example in Algorithm 1) or dynamically shared by multiple PEs. Lastly, a tile of the output vector $\vec{y}$ (tile size of $R_t$) is streamed in ($\overrightarrow{y\_in}$) and out ($\overrightarrow{y\_out}$), and partitioned in the same cyclic fashion as matrix $\mathbf{A}$.

However, existing approaches [11, 24, 34] no longer work effectively when accelerating *imbalanced SpMV* where the distribution of nonzeros in the sparse matrix is highly imbalanced across different rows. For instance, Figure 1 illustrates the nonzero distribution of the *hangGlider*_3 matrix used in the optimal control solver [6], where one row contains nearly 1,000 times more nonzero elements compared to the average number of nonzeros in other rows. Such imbalanced matrices are commonly used in multiple application domains, such as optimal control solver [6], orbit transfer [5], circuit simulation [10], and natural language processing [10], which create multiple new challenges for efficient SpMV acceleration. Moreover, recent studies such as FlightVGM [29] and FlightLLM [43] have shown the potential of implementing the latest large language and video models with sparsification on FPGAs. In such AI models, the sparsity of the layers can vary, and in such cases, we need an accelerator that can process both sparse and dense matrices, which is also explored in one of the recent studies [36]. Such workloads with both SpMV and GeMV suffer from the overhead of switching between two different kernels, motivating us to design an overlay for both SpMV and GeMV. In summary, the following problems are addressed in this work:



(a) nonzero Structure



(b) Row Distribution

Fig. 1. Motivation: highly imbalanced nonzero distribution of the hangGlider_3 matrix in optimal control solver [6]

1). **Imbalanced Workload:** For the highly imbalanced matrices such as the example in Figure 1, the cyclic row assignment strategy used in existing SpMV accelerators [11, 24, 34] (illustrated in Algorithm 1) results in a severe workload imbalance among PEs, which can lead to up to 280x slowdown for the computation time of $\mathbf{A} \times \vec{x}$ in our evaluated imbalanced matrices.

2). **Long-Distance RAW Dependency:** Inside each PE, the floating-point accumulation on the temporary output vector $y\_Ax[r\%R_t]$ (line 7 in Algorithm 1) causes a read-after-write (RAW) dependency when processing multiple nonzero elements from the same row (i.e., same $r$, different $c$) consecutively. Due to the long latencies (10 clock cycles) involved in the floating-point addition and read/write operations on the large buffer $y\_Ax$, this RAW dependency inhibits effective pipeline (leading to pipeline stalls) to process nonzero elements from the same row.
Existing studies [11, 34] usually use re-ordering techniques to schedule the processing of nonzeros from other rows to fill in the pipeline gap (stalls) during the RAW dependency waiting cycles. However, this no longer works effectively for highly imbalanced matrices such as the example in Figure 1, as there are significantly fewer nonzeros in other rows available to fill in the pipeline stalls between the nonzeros in the densest row.

3). **Input/Output Dense Vector Off-Chip Access Latency:** Using a single design with a fixed number of HBM channels for sparse matrix and dense vectors is not optimal for all the matrices. In matrices with extremely low density (e.g., $10^{-4}$ or lower), the off-chip memory access for the input dense vector $\vec{x}$ and output vector $\overrightarrow{y\_in}/\overrightarrow{y\_out}$ could become a new performance bottleneck after the acceleration of SpMV, which is often overlooked in prior studies [11, 24, 34].

4). **Resource v/s Optimization Trade-offs:** Using a generic accelerator with a fixed set of optimizations is sub-optimal, as these optimizations consume more resources and limit the total number of PEs, which can be detrimental to the matrices that do not see substantial benefit from certain optimizations. For example, a well-balanced matrix performance is better with a 192 PE without the optimization for addressing imbalanced row distribution, compared to a 128 PE design with that optimization.

5). **Applications with Mixed Workload:** In certain applications like machine learning, models often have both dense and sparse linear layers, which necessitate the support for both SpMV and GeMV kernels on the hardware. Using different bitstreams for these kernels requires re-configuration of the FPGA every time we switch kernels, which has a big overhead, outweighing the speedup offered by the kernels.

In this work, we build MAD-HiSpMV to accelerate imbalanced SpMV and GeMV on HBM-equipped FPGAs, with the following novel features:

1). **Hybrid Row Distribution Network:** To achieve a more balanced workload distribution, we design a hybrid row distribution network such that the same set of PEs can either work on different rows (inter-row distribution) or collaboratively work on a single row (intra-row distribution).

2). **Fully Pipelined Floating-Point Accumulation:** To address the long-distance RAW dependency issue, we propose two techniques to achieve fully pipelined floating-point accumulation. First, to conceal the read/write latency to the large buffer $y\_Ax$, we design a small **register-based circular buffer** in each PE for fast access of the intermediate accumulation result. This could essentially reduce the dependency distance from 10 to 5. Second, to hide the latency of the single floating-point adder for accumulation, we implement a small **adder chain** (thanks to the reduced dependency distance) to independently pre-add results from the same row before accumulation, thus avoiding pipeline stalls without the need of scheduling nonzeros from other rows.

3). **Matrix Adaptive Design:** In MAD-HiSpMV, we build matrix adaptive designs, which are configured with an optimal number of HBM channels for the sparse matrix and dense vectors, along with enabling the right combination of the aforementioned optimizations that are optimal for the given input matrix. The automation tool is equipped with a **Design Space Exploration (DSE)** to perform this customization.

4). **Dense Overlay**: In MAD-HiSpMV, the automation tool provides an optional flag to implement PEs with a dense overlay, and allows the execution of both SpMV and GeMV on the same kernel, thus avoiding the re-configuration of different bitstreams while executing a mixed workload.

We previously built HiSpMV [33] (our conference version) on top of state-of-the-art open-source SpMV accelerator Serpens [34] due to its superior performance among existing studies, with the aforementioned new features to accelerate imbalanced matrices. In MAD-HiSpMV, we build matrix adaptive designs, unlike the prior generic design used for all the input matrices. We evaluate the performance of MAD-HiSpMV using a dataset comprising 10 balanced and 10 imbalanced matrices from the widely used SuiteSparse [10], with the balanced matrices chosen from prior SpMV studies [24, 34]. On AMD-Xilinx Alveo U280 FPGA, for balanced matrices, the MAD-HiSpMV DSE picked best configurations achieve geomean speedups of 1.34x, 1.31x, 1.06x, 1.7x, 2.03x and 1.22x over prior studies Serpens-24 [34], AMD design [24], HiHiSpMV [37], Cuper [42], CoSpMV+ [38] and HiSpMV-20 [33] respectively. For imbalanced matrices, MAD-HiSpMV configurations achieve geomean speedups of 23.0x, 5.2x, 26.79x, 3.43x and 1.51x over prior studies Serpens-24 [34], HiHiSpMV [37], Cuper [42], CoSpMV+ [38] and HiSpMV-16 [33], respectively.

In addition, MAD-HiSpMV achieves geomean speedups of 6.19x and 12.51x over Intel MKL running on a 24-core Xeon Silver 4214 CPU, for balanced and imbalanced matrices, respectively. It also achieves a geomean of 2.95x and 2.24x better performance per watt (GFLOPS/watt) compared to Nvidia cuSparse running on the GTX 1080ti GPU, for balanced and imbalanced matrices, respectively. Furthermore, the MAD-HiSpMV configuration built with dense overlay achieves a peak GFLOPS of 104.0 and 156.73 on AMD-Xilinx Alveo U50 and U280, respectively, for the matrix size of $8192 \times 8192$, which surpasses the performance of Vitis L2 GeMV benchmarks on both U50 [2] and U280 [13] by 2.6x. Finally, MAD-HiSpMV performs 2.7x better for an end-to-end neural network with mixed workload, when compared to Intel MKL running on a 24-core Xeon Silver 4214 CPU.

Table 1. Comparison of MAD-HiSpMV with prior SpMV accelerators on HBM-based FPGA. *The configuration of hybrid row distribution is optional, and it can either use an adder chain or reordering as conflict resolution based on the input matrix.

| **Accelerator** | **No. of HBM Channels Assigned for Sparse Matrix and Dense Vectors** | **Imbalanced Workload** | **RAW Dependency on Output Buffer** | | **Input Vector Buffer** | **Geomean GFLOPS** | |
|---|---|---|---|---|---|---|---|
| | | | **Distance Reduction** | **Resolution Technique** | **PE Access** | **Bala-nced** | **Imbal-anced** |
| Serpens-16 [34] | Fixed | X | None Distance: 10 | Re-Ordering | Private Copy | 33.23 | 1.22 |
| Serpens-24 [34] | Fixed | X | None Distance: 10 | Re-Ordering | Private Copy | 54.67 | 1.23 |
| HiSparse-PB [11] | Fixed | X | IFWQ Distance: 7 | Partial Sum Buffers | Dynamic Sharing | 9.86 | 1.12 |
| HiSparse-RI [11] | Fixed | X | IFWQ Distance: 7 | Re-Ordering | Dynamic Sharing | 9.78 | 0.85 |
| AMD design [24] | Fixed | X | N / A | Dynamic Stall | Dynamic Sharing | 43.48 | N / A |
| HiSpMV-16 [33] (our previous design) | Fixed | Hybrid Row Distribution | Circular Buffer Distance: 5 | Adder Chain | Hybrid Buffering | 39.14 | 18.72 |
| HiSpMV-20 [33] (our previous design) | Fixed | Hybrid Row Distribution | Circular Buffer Distance: 5 | Re-Ordering | Hybrid Buffering | 46.47 | 16.04 |
| Cuper [42] | Fixed | X | None Dsitance: 10 | Partial Sum Buffers | Private Copy | 33.3 | 1.05 |
| HiHiSpMV [37] | Fixed | X | BRAM Buffer Distance: 5 | Adder Chain | Private Copy | 53.4 | 5.43 |
| CoSpMV+ [38] | Fixed | Input Specific Design Parameters | Register Chain Distance: 5 | Adder Chain | Dynamic Sharing | 27.94 | 8.22 |
| **MAD-HiSpMV (this work)** | **Input Specific** | **Hybrid Row Distribution*** | **Circular Buffer Distance: 5** | **Adder Chain/ Re-Ordering*** | **Private Copy** | **56.8** | **28.23** |

## 2 Related Work

### 2.1 SpMV Accelerator Design

In Table 1, we present a detailed breakdown of the novel contributions that set our work apart from the other SpMV accelerators. We published our first work, HiSpMV [33], with two of the key contributions presented in this journal extension, which include a hybrid-row-distribution for addressing imbalanced row distribution, a register-based circular buffer for reduced dependency distance, and an adder-chain to resolve RAW conflicts of the long latency floating-point accumulation on the output buffer. However, in this work, we introduce several novel features, including matrix adaptive design to address inefficiencies of generic accelerators for various input matrices, and a dense overlay to support GeMV on the same kernel, thereby improving the efficiency of mixed workloads.

*2.1.1 Input Specific Design:* The recent work CoSpMV [38] attempts software and hardware co-design for SpMV acceleration, with their hardware primarily optimized for DDR-based FPGAs. They nonetheless managed to scale up the design for HBM-based FPGAs. However, there are limitations on the hardware configurations they can generate for different input matrices, and the lack of support for imbalanced row distribution results in inferior performance compared to our designs.

*2.1.2 Dependency Distance and RAW Conflict Resolution:* After HiSpMV [33], the works such as HiHiSpMV [37], and CoSpMV [38] were also able to achieve a reduced dependency distance (*dd*) of 5. While the solution in CoSpMV [38] is similar to our solution of using local register-based small buffers for accumulation instead of accumulating directly on

Table 2. GeMV performance comparison of MAD-HiSpMV with Vitis L2 GeMV benchmarks

| **Kernel (Device)** | Vitis L2 GeMV (U50) [2] | **MAD-HiSpMV (U50)** | Vitis L2 GeMV (U280) [2] | Vitis L2 GeMV (U280) [13] | **MAD-HiSpMV (U280)** |
|---|---|---|---|---|---|
| **Num. Of HBM Ch. (for streaming A)** | 16 | **16** | 16 | 32 | **24** |
| **Peak GFLOPS (8192 x 8192)** | 40.32 | **104.0** | 40.35 | 59.37 | **156.73** |

URAM buffers, HiHiSpMV [37] gets rid of the URAMs completely and uses BRAM buffers instead. However, this results in reduced output buffer size, and the design fails to run for the matrices with larger dimensions. Other studies, such as Cuper [42] and HiSparse-PB [11], use partial sum buffers, where the accumulation is done on multiple partial sum output buffers; however, this demands *dd* times more buffer storage to store the same size of the output vector.

*2.1.3 Input Buffer Access:* There are two primary methods by which the PEs can access the input vector. In the first approach, a private copy is used in the studies [34, 37, 42], where the input vector is replicated for each PE to access it without any dependencies with other PEs. The other approach, dynamic sharing, is used in the studies [11, 24, 38], which involves sharing a single partitioned copy of the input vector among multiple PEs. Although this method consumes fewer on-chip memory resources, it leads to conflicts when various PEs want to access the same partition. Based on our analysis (Table 1), the performance benefit of the private copy outweighs its increased on-chip resource consumption.

*2.1.4 Other Studies:* Prior to Serpens, several designs aimed to accelerate graph applications through SpMV. GraphLily [20] targets graph applications presented in SpMV, and designs a general SpMV accelerator with an overlay design, but it only reaches 165MHz with limited performance. HitGraph [44] proposes a graph processing acceleration framework that can perform SpMV. ThunderGP [7] is an HLS-based graph processing framework that can perform graph partition automatically for various graph algorithms. Both HitGraph and ThunderGP only utilize DDR memory, and none of the above designs is specialized for SpMV. Jain et al. [23] present an accelerator design based on the AMD/Xilinx GEMX SpMV engine, but the performance is much lower. Liu et al. [28] reorder the data to solve the input vector conflict and design an adder tree to resolve the write conflict when multiple PEs work on the same row. However, their design does not address the RAW dependency for accumulation and can hardly scale to larger FPGAs. Li et al. [27] propose a novel compressed format tackling inefficient memory access, though scalability remains unaddressed. Other SpMV-related accelerator designs, like SpaceA [41] and GraphR [35], are only evaluated in simulation.

## 2.2 GeMV and Overlay Designs

Table 2 presents a summary of the performance of our GeMV overlay designs compared to the Vitis L2 GeMV benchmarks [2, 13], highlighting the performance gains achieved by our designs. Recent studies [29, 36] further emphasize the growing role of FPGAs in accelerating mixed workloads involving both sparse and dense computations. In particular, architectures such as systolic sparse tensor slices [36] and FlightVGM [29] demonstrate significant improvements in performance and energy efficiency, reinforcing the potential of FPGAs for next-generation AI applications.

## 2.3 Timing Optimization for HLS Designs

HBM and multiple dies have been adopted into modern datacenter FPGAs such as AMD-Xilinx Alveo U280. Interconnections crossing the dies with long delays introduce more challenges to improving the quality of placement and routing. To tackle this problem, we utilize the recent PASTA [26] framework, which is built on top of TAPA/Autobridge [16, 17].

TAPA/Autobridge proposes a task-parallel HLS programming model with a coarse-grained floorplanning approach to improve the timing closure and clock frequency. Based on TAPA/Autobridge, PASTA further extends the task communication channel support from FIFOs to both FIFOs and buffers, greatly improving its programmability for a wider range of applications.

## 3 Motivation and High-Level Ideas

In this section, we present a comprehensive analysis of new challenges associated with accelerating imbalanced SpMV and mixed workloads, along with our high-level ideas to address these challenges.

### 3.1 Imbalanced Workload

*3.1.1 Analysis of the Problem:* First of all, we analyze how much impact the workload imbalance can have on the performance of existing SpMV accelerators [11, 24, 34] that cyclically assign rows onto $P$ number of PEs. Assume the input matrix has $R$ rows and $C$ columns, $NNZ$ is the number of nonzeros, and the density $\rho$ of the matrix is $\rho = NNZ/(R*C)$. Using the cyclic row assignment, each PE gets $R/P$ rows. To illustrate the inefficiency in prior SpMV accelerators, we define the matrix **imbalance ratio** $\delta$ as:

$$\delta = actual\_PE\_load \ / \ ideal\_PE\_load \tag{2}$$

The ideal PE workload is evenly divided among all the PEs:

$$ideal\_PE\_load = NNZ/P = R*C*\rho/P \tag{3}$$

In the worst case, all the nonzeros could be allocated to one PE. But since a PE only gets $R/P$ rows, it can only have a maximum of $R*C/P$ nonzeros. Hence, the upper limit for actual PE load is:

$$actual\_PE\_load \leq Min(NNZ, \ R*C/P) \tag{4}$$

If we substitute equation 3 and 4 in equation 2, we get:

$$\delta \leq Min(P, \ 1/\rho) \tag{5}$$

From equation 5, we observe that: 1) as the matrix density $\rho$ becomes lower, the imbalance ratio's upper bound becomes higher; 2) having more PEs also increases the imbalance ratio's upper bound. We have profiled a set of sparse matrices from the widely used SuiteSparse [10] with a different imbalance ratio $\delta$ (Table 4 in Section 7.1). For imbalanced matrices with $\delta \geq 2$, state-of-the-art SpMV accelerator Serpens-16 [34] only achieves a geomean of 1.22 GFLOPS, which is about 27.2x lower than that for balanced matrices.

*3.1.2 Proposed Solution:* To address this imbalanced workload issue, we propose a novel **hybrid row distribution network** to allow the same set of PEs to work in two different modes: 1) inter-row distribution where the PEs work on different rows assigned cyclically, and 2) intra-row distribution where all the PEs collectively work on the same row. Its detailed design and implementation will be presented in Section 4.2 and 4.3.

### 3.2 Long-Distance RAW Dependency

*3.2.1 Analysis of the Problem:* In line 7 of Algorithm 1, a floating-point (FP) accumulation on $y\_Ax[r\%R_t]$ occurs. In FPGAs lacking dedicated hardware for FP addition, soft IPs require multiple clock cycles for this addition. Moreover, read and write latencies are incurred for on-chip buffers like BRAM and URAM.

Consider a PE load scenario with $a_{00}, a_{01}, a_{02}, a_{09}, a_{12}, a_{23}$, where $a_{rc}$ represents the value in the $r^{th}$ row and $c^{th}$ column of the sparse matrix $A$ in Figure 2(a). In the second iteration, a read-after-write (RAW) dependency occurs on $y\_Ax[0]$, preventing the pipeline from achieving an initiation interval (II) of 1. The total latency, including reading,

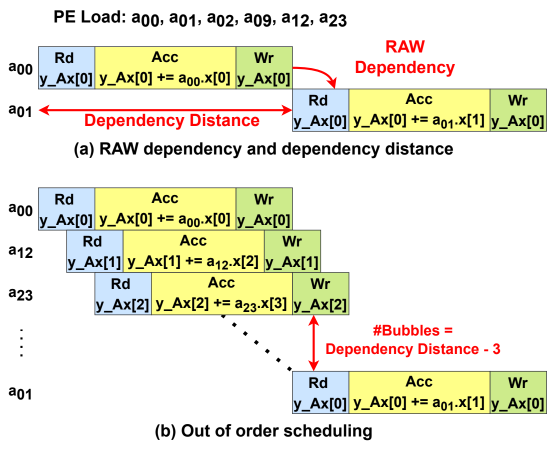


Fig. 2. RAW dependency for floating-point accumulation on $y\_Ax[r\%R_t]$ and out-of-order scheduling to avoid stalls

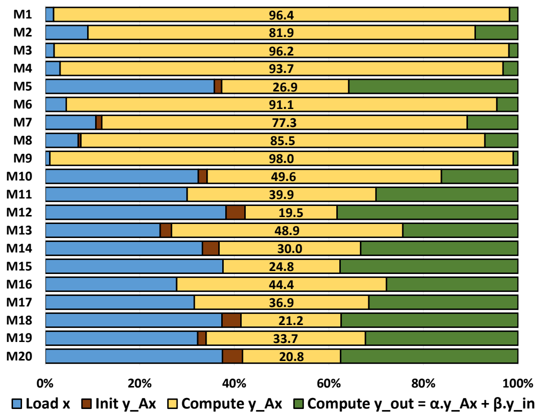


Fig. 3. Execution cycle breakdown for each component in the well-optimized SpMV accelerator

writing, and accumulation, defines the dependency distance ($dd$) across loop iterations. Maintaining element order would introduce $dd - 1$ pipeline bubbles between elements of the same row, severely impacting performance.

Prior studies such as Serpens [34] and HiSparse-RI [11] adopt an out-of-order scheduling, shown in Figure 2(b), to schedule nonzeros from other rows to fill pipeline gaps. While effective for balanced matrices, this method faces challenges with imbalanced matrices like hangGlider_3 in Figure 1: when dealing with a much denser row, there are not enough nonzeros available from other rows to fill this pipeline gap. On the other hand, HiSparse-PB [11] introduces additional $y\_Ax$ buffer copies to resolve this issue, which incurs more on-chip buffer overhead. The latest AMD design [24] dynamically stalls the pipeline using a hazard resolution back-pressure unit, which does not effectively avoid the stalls.

Another alternative is to reduce the dependency distance $dd$, which is crucial to minimize pipeline gaps. While FP addition latency is unavoidable in FPGAs without hardened FP IPs, read/write latency could be eliminated by storing a few recent $y\_Ax$ results in local registers. For example, HiSparse [11] uses an *in-flight-wait-queue (IFWQ)* to store results in local registers and reduces $dd$ from 10 (in Serpens [34]) to 7. However, to ensure correct values are used, it needs to check whether the required result is in the queue by comparing row indices with every single element in the queue. This leads to unnecessary resource overhead.

*3.2.2 Proposed Solution:* We propose two novel techniques to address this long-distance RAW dependency. First, we design a register-based **circular buffer** to reduce the dependency distance to 5. Compared to an IFWQ in HiSparse [11], it only requires a row index check in one location in the buffer to ascertain the presence of a $y\_Ax_i$ value. Second, we employ an **adder chain** to independently pre-add $\mathbf{A} \times \vec{x}$ product results during the last $dd$ iterations (including the current $i^{th}$ iteration) while waiting for the accumulation of $y\_Ax_{i-dd}$ to finish. In the next cycle, we can add $y\_Ax_{i-dd}$ (retrieved from the circular buffer) with the pre-added result to get the new accumulation result $y\_Ax_i$. Therefore, we can achieve full pipelining of the FP accumulation process without the need for out-of-order scheduling or bubbles in the pipeline. The detailed design and implementation will be presented in Section 4.4.

### 3.3 Bottlenecks on Dense Vectors and Resource v/s Optimization Trade-offs

*3.3.1 Analysis of the Problems:* After streaming the sparse matrix **A** and optimizing the computation of $\mathbf{A} \times \vec{x}$, we profile the execution cycle breakdown for each component of the SpMV accelerator described in Algorithm 1. As shown in Figure 3, an interesting observation was made: for very low-density matrices (e.g., $\rho \leq 10^{-4}$), two new bottlenecks arise in 1) loading the input dense vector $\vec{x}$ (line 5 in Algorithm 1), and 2) streaming and computing the output dense vector $\overrightarrow{y\_out}$ (line 9 in Algorithm 1). Unfortunately, these new bottlenecks are often overlooked in prior studies [11, 24, 34]. To address these bottlenecks, allocating more HBM ports to dense vectors might appear to be a straightforward solution; however, this approach will necessitate more on-chip resources and reduce the number of available ports for streaming sparse matrices, potentially degrading the performance in cases where computation is the primary bottleneck.

On the other hand, optimizations such as *hybrid row distribution* and *adder chain* increase resource consumption, meaning that a design incorporating both optimizations will be constrained to a lower number of PEs compared to a design with only one of the optimizations. Hence, having a generic accelerator for all the matrices is not optimal; for example, a well-balanced matrix like $M1$ from our benchmark matrices (Table 4) benefits more from having a higher number of PEs without *hybrid row distribution* (which is an optimization used to address imbalanced row distribution). Similarly, other matrices that benefit more from a higher number of PEs rather than having the *adder chain*, can rely on the re-ordering to avoid RAW conflicts from long latency floating-point accumulation on the output buffer.

*3.3.2 Proposed Solution:* Previously in HiSpMV [33], to tackle the bottleneck of loading the input dense vector $\vec{x}$, we used the hybrid-buffering feature from the PASTA framework [26]. To address the output dense vector $\overrightarrow{y\_out}$, we simply scaled the number of ports as much as possible. In this journal extension, we take advantage of the 2-port buffer channels to reduce the overall buffer utilization, thanks to the latest improvement in the PASTA framework [26]. Moreover, this new reduced utilization allows us to scale up the number of HBM ports used to load the input dense vector from off-chip. In MAD-HiSpMV, we build matrix adaptive accelerators which are configured with the appropriate optimizations, the optimum number of HBM channels for the matrix **A**, and input/output dense vectors $\vec{x}$, $\overrightarrow{y\_in}$, and $\overrightarrow{y\_out}$. This is accomplished via the automation tool with **Design Space Exploration (DSE)** and accurate analytical resource and performance estimation models. The detailed design and implementation will be presented in Section 6.

### 3.4 Applications with Mixed Workload

*3.4.1 Analysis of the Problem:* In recent years, machine learning and deep learning (ML/DL) models have grown significantly in size, making pruning essential for deploying these models on hardware with limited memory. Models with multiple layers, which are dense initially, will end up with layers of varying sparsity after pruning. This leads to a workload that has mixed GeMV and SpMV kernels, which are run sequentially. Switching between two different bitstreams each time we run a different kernel is highly inefficient, as the time required to load a new bitstream could be four orders of magnitude higher than the typical kernel runtime, as illustrated in Figure 4.

*3.4.2 Proposed Solution:* We observed that in our SpMV kernels, the matrix data has no reuse, and the number of computations we can do is limited to the number of elements in **A** that we can read in parallel. Even after scaling the design to use the maximum number of HBM channels to stream the **A** matrix, the DSPs on the FPGA are still underutilized. Accordingly, utilizing these available DSPs, we built a dense overlay within our PEs, which can perform GeMV on the same kernel. A global flag set by the host while invoking the kernel determines the operation mode.

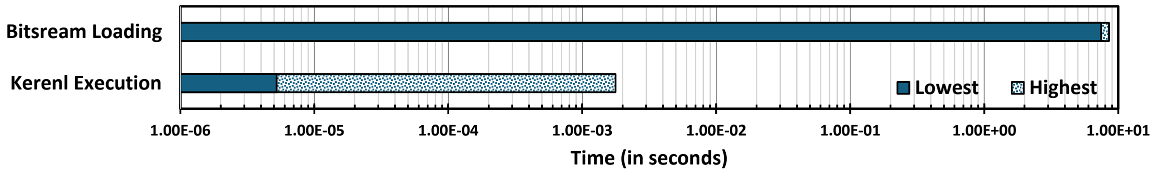


Fig. 4. Bitstream loading time compared to the kernel execution time of MAD-HiSpMV for benchmark matrices on U280

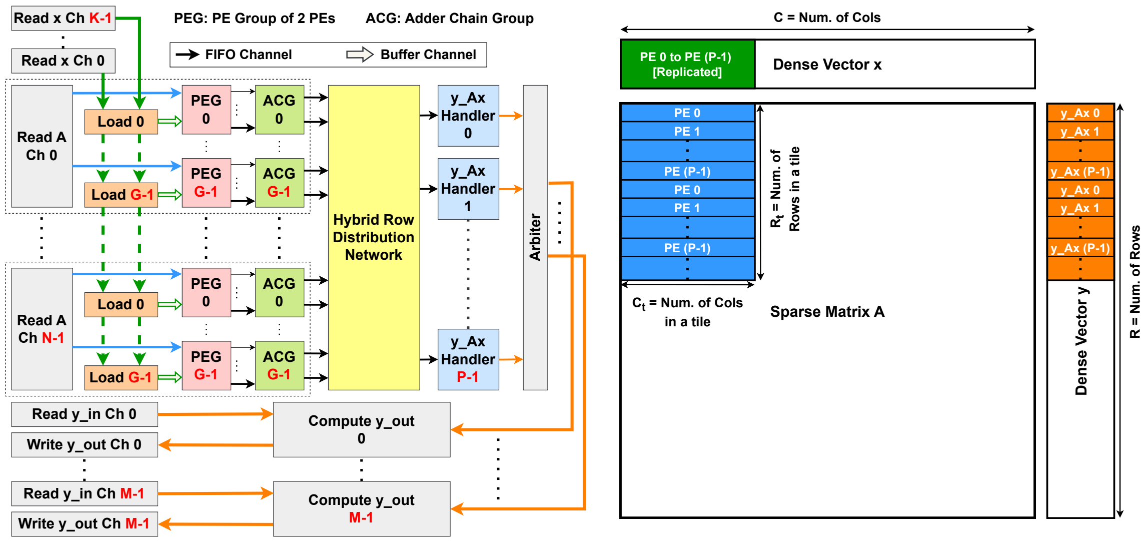


Fig. 5. MAD-HiSpMV architecture overview, total number of PEs is computed as $P = N * G * 2$.

## 4 MAD-HiSpMV Design and Implementation

### 4.1 MAD-HiSpMV Architecture Overview

In our previous work, HiSpMV [33], we introduced optimizations such as *hybrid row distribution* and *adder chain*. Building on that foundation, MAD-HiSpMV incorporates several key enhancements. Firstly, we built a design with a scalable number of HBM ports to load the input vector instead of a single port in HiSpMV [33]. Secondly, we adopted the design configuration based on the input matrix for optimal performance. Thirdly, we designed a PE architecture with a dense overlay to accommodate both SpMV and GeMV on a single kernel.

MAD-HiSpMV (and HiSpMV [33]) is based on state-of-the-art open-source SpMV accelerator Serpens [34], with the support of novel features summarized in Section 3. Figure 5 presents an overview of the MAD-HiSpMV architecture. Initially, we divide the *loading* of input vectors into a distinct module, isolating it from the *processing engine group (PEG)*, and introduce a *buffer channel* to facilitate the communication between these two modules. We relocate the accumulation and output buffer from the PEG into their dedicated *y_Ax handler* module. Moreover, we introduce new modules, including the *adder chain groups (ACG)* and a *hybrid row distribution network* between the PEGs and *y_Ax* handlers. The design utilizes $K$ HBM channels to load input dense vector $\vec{x}$, then chain broadcast to all the load modules that buffer $\vec{x}$ on-chip. We employ a total of $N$ HBM channels for streaming sparse matrix $A$, with each channel serving $G$ PEGs, where each PEG includes 2 PEs. We have $M$ pairs of HBM channels designated for streaming in $\overrightarrow{y_{in}}$ and streaming

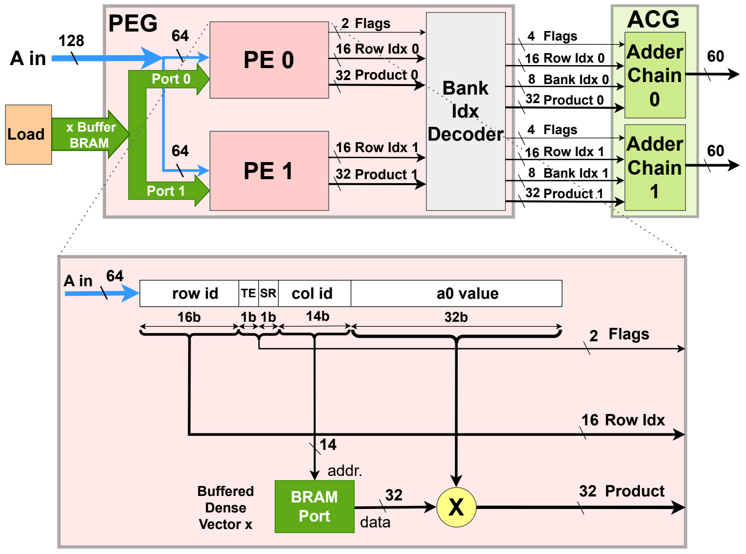


Fig. 6. PEG design and PE without dense overlay

**Algorithm 2** Bank Index Decoder Logic

```
1: if SR then                                          ▷ Split Row: Intra-row distribution
2:     $bank\_idx0, bank\_idx1 \leftarrow row\_idx0, row\_idx0$
3:     $row\_idx0, row\_idx1 \leftarrow row\_idx1, row\_idx1$
4: else                                                ▷ Inter-row distribution
5:     $bank\_idx0, bank\_idx1 \leftarrow PE\_idx0, PE\_idx1$
6:     $row\_idx0, row\_idx1 \leftarrow row\_idx0, row\_idx1$
```

out $\overrightarrow{y_{out}}$. All these channels are 512 bits wide, for optimal bandwidth utilization [30]. Since each PE consumes a data width of 64b, each PEG requires 128b, so we end up with $G = 4$ PEGs per channel (i.e., 8 PEs per channel).

The PEs perform the multiplication of nonzero elements $a_{rc}$ with $x_c$ in the input vector buffer, described as $a_{rc} * buf\_x[c\%C_t]$ in line 7 of Algorithm 1. PEs also decode to which $y\_Ax$ handler (i.e., bank ID) the multiplication product should be routed to, based on the encoded inter-row and intra-row distribution info (encoded during preprocessing). Based on the decoded bank ID, a hybrid row distribution network routes the product to the corresponding $y\_Ax$ handler for accumulation. Note each PE corresponds to one $y\_Ax$ handler, so there are $8 \times N$ $y\_Ax$ handlers in total. Adder chains are optionally placed after the PEGs to independently pre-add products before they are routed to the $y\_Ax$ handlers to avoid pipeline stalls caused by the long-distance RAW dependency. In addition, each $y\_Ax$ handler also employs the register-based circular buffer to reduce the dependency distance to 5. Upon computing $y\_Ax[R_t] = \mathbf{A} \times \vec{x}$, it streams in $\overrightarrow{y_{in}}$, computes $\overrightarrow{y_{out}}$ (line 9 of Algorithm 1), and streams out $\overrightarrow{y_{out}}$.

## 4.2 Processing Element Group (PEG) Design

For each read **A** channel that is 512-bit wide, it serves 4 PEGs. So each cycle, as shown in Figure 6, each PEG reads in 128-bit of **A**, and each PE inside the PEG reads in 64-bit of **A**. Similar to Serpens [34], this 64-bit data includes the 16-bit row id $r$, a 14-bit column id $c$, and a 32-bit nonzero value $a_{rc}$, as well as two flag bits (Tile End, and Split Row, which are

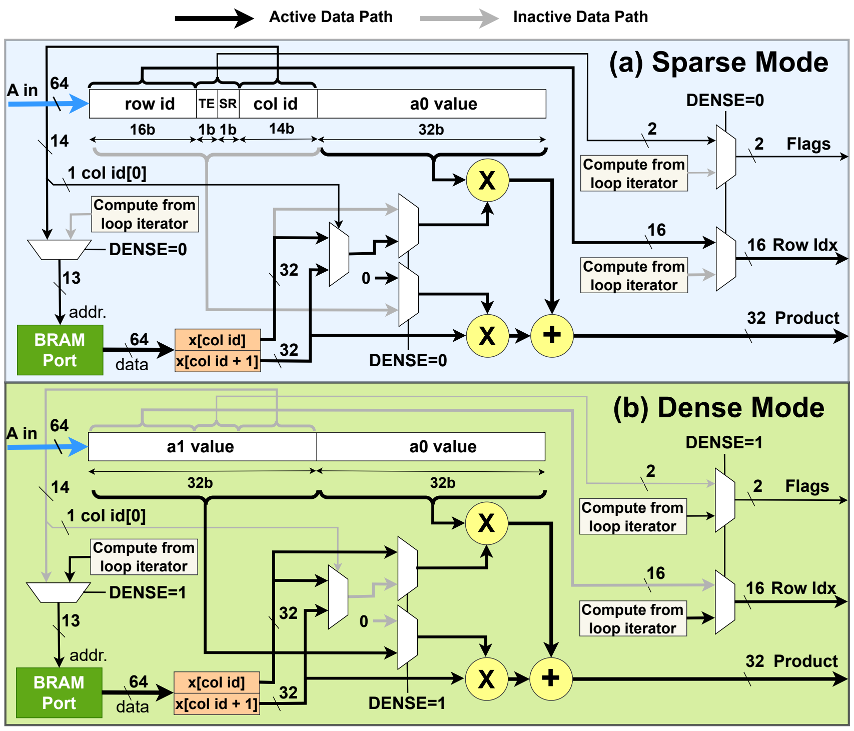


Fig. 7. PE architecture with dense overlay, operation mode determined by the flag 'DENSE' set by host

explained in Section 4.2.2). Using the column id $c$, each PE fetches the corresponding (32-bit) $x_c$ value from the $\vec{x}$ buffer ($buf\_x[C_t]$ inside the BRAM Buffer Channel in Figure 6) and performs the floating-point multiplication of $a_{rc}.x_c$. The product, row index, and flags are then sent to the bank index decoder. Note that the $\vec{x}$ buffer is set to 512-bit wide, as it achieves the best off-chip bandwidth during the load from the HBM channel [30].

*4.2.1 Dense Overlay:* Fig 7 shows how a single hardware block can be shared and operates differently in (a) sparse and (b) dense mode. The operating mode is decided by a single-bit flag named 'DENSE', which is set by the host. In the sparse mode, the PE works as described in the earlier part of Section 4.2. In the dense mode, the 64-bit input to the PE is packed with 2 elements from **A**, which belong to the adjacent columns of the same row. Each PE is assigned a set of rows from the input matrix **A** divided cyclically, similar to the sparse mode. The elements are streamed in column-major order, meaning the PEs will first receive the elements in column 0 and column 1 of all the rows assigned to them within a tile. The predetermined order, along with the dimensions of the matrix, is used to determine the row index and column index of the elements within the PE. As a dense matrix is always balanced, no balancing is needed with intra-row mode; hence, the flag *SR* (Split Row) is always set to $false$ in dense mode. With the row and column indices already computed, we can use the tile dimensions to calculate the flag *TE* (Tile End).

Since each of the $\vec{x}$ buffers is 512b wide, we can access up to 16 (fp32) elements (512b aligned) in parallel. By packing two adjacent columns, for example, columns 0 and 1, we make sure that the $\vec{x}$ buffer accesses can be satisfied with a single port, as they are part of the same 512-bit aligned address. Therefore, we can compute 2 products and accumulate them into a single result as both elements belong to the same row.

*4.2.2 Bank Index Decoder:* The bank index determines the specific $y\_Ax$ handler the product should be routed to for accumulation, which is encoded during preprocessing based on the intra-row and inter-row distribution. No special encoding is needed for inter-row distribution, the row indices do not change, and the bank index for the $y\_Ax$ handler is the same as the PE index. For an intra-row distribution that works on the same row $r$, for every pair of adjacent nonzeros $a_{rc_0}$ and $a_{rc_1}$, we encode their bank index in the row id of $a_{rc_0}$ and their row index in the row id of $a_{rc_1}$. Accordingly, the decoder logic in Figure 6 works as follows: Besides the 2 flags $TE$ ("Tile End": indicate the end of a tile in the matrix data) and $SR$ ("Split Row" (SR) indicates whether the row belongs to inter-row mode or intra-row mode), the bank index decoder also sends 2 additional flags "Last" (indicating the end of a kernel run to the downstream tasks) and "Dummy" (indicating this packet is used only for signaling, and not a valid result).

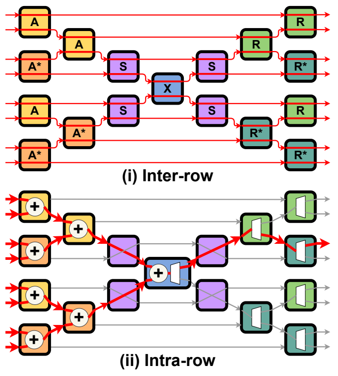
### 4.3 Hybrid Row Distribution Network

Based on the decoded bank ID, a hybrid row distribution network routes the product from each PE to its corresponding $y\_Ax$ handler for accumulation. Since there are $8 \times N$ PEs and $8 \times N$ $y\_Ax$ handlers in total, this network has $8 \times N$ inputs and $8 \times N$ outputs. To make it easy for placement and routing, we built this network based on small blocks with 2 inputs and 2 outputs. Figure 8a illustrates the functionality of an example 8x8 network in two modes. *i) Inter-row distribution:* When PEs work on separate rows (cyclically), all the blocks forward packets directly from inputs 0 and 1 to outputs 0 and 1, respectively, without any additional processing. *ii) Intra-row distribution:* When PEs work on the same row, the network first accumulates all products using a reduction tree and then routes the sum to the corresponding $y\_Ax$ handler using a reversed reduction tree; the route is highlighted in red in Figure 8a. Shown in Figure 8a, the network is constructed using the following small blocks. **Adder blocks 'A' and 'A*'** add the results from two inputs and direct the sum to output 1 and 0, respectively. The other output will just be a dummy packet. **Switch blocks 'S'** switch the outputs, forwarding results from input 0 to output 1 and from input 1 to output 0. **Routing blocks 'R' and 'R*'** facilitate proper $y\_Ax$ handler routing: they route input 1 for 'R' and input 0 for 'R*' to either output 0 or 1 based on the target $y\_Ax$ handler (bank index). **Fused block 'X'** combines the functionalities of both adder and routing blocks: it adds the results from two inputs and routes the sum to the appropriate output (0 or 1) based on the target $y\_Ax$ handler (bank index). The decision of which mode to process each row is made in the preprocessing step, and the detailed implementation is explained in Section 5.

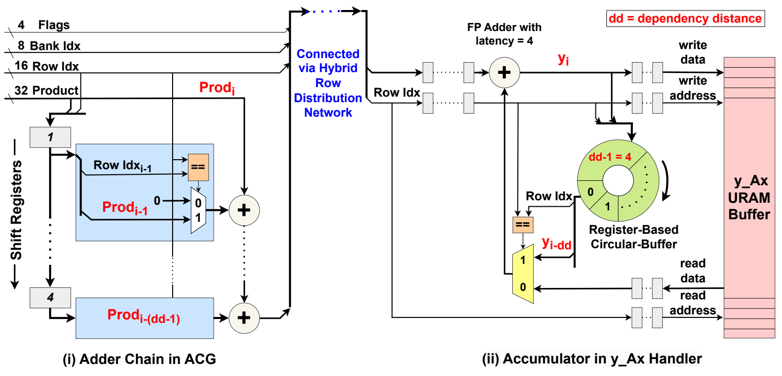
### 4.4 Pipelined Floating-Point Accumulation

To achieve a fully pipelined floating-point accumulation, we employ a combination of a register-based circular buffer and an adder chain.

*4.4.1 Circular Buffer:* Accumulating directly on buffers like URAM or BRAM introduces read/write latencies. Therefore, as shown in Figure 8b. (ii), we utilize a local register-based circular buffer to store temporary accumulation values to avoid such latencies. As presented in Section 3.2, one common approach is to schedule $dd - 1$ ($dd$: dependency distance) nonzeros from other rows to fill the pipeline gap during the floating-point accumulation. Therefore, the size of this circular buffer is set to be $dd$, so that it can buffer $dd - 1$ accumulation results for other rows and one for the current row that started $dd$ cycles ago. When a new product for the current row comes, it knows the prior accumulation result for this row started $dd$ cycles ago (if it is not a new row) and can precisely find its location in the circular buffer. Thus, it only needs one row index check to confirm if it is a new row: if not, it can use the result retrieved from the circular buffer to add to the new product; otherwise, it starts the accumulation for a new row. Vitis HLS synthesizes



(a) 8x8 hybrid row distribution network



(b) Fully pipelined floating-point accumulation with adder chain and circular buffer

Fig. 8. (a) An example 8x8 hybrid row distribution network. $A, A^*$ are adder blocks, $R, R^*$ are routing blocks, $S$ is switch block, and $X$ is fused block. (b) Fully pipelined floating-point accumulation with adder chain and circular buffer.

the floating-point adder with a minimum latency of 4 cycles, and an additional cycle is needed to write the output, resulting in a total dependency distance of $dd = 5$.

*4.4.2 Adder Chain:* Here, we consider a scenario when products sent from the PEs belong to the same row and there is no out-of-order scheduling. Let $Prod_i$ be the $i^{th}$ product in that row and $y_i$ be the accumulated sum of all these products up to that iteration. This time, the circular buffer could store $y$ results from the same row. The key idea here is to employ an **adder chain** to independently pre-add $\mathbf{A} \times \vec{x}$ product results during the last $dd$ iterations (including the current $i^{th}$ iteration) while waiting for the accumulation of $y_{i-dd}$ to finish. Let us denote the pre-added result $Q$ of the adder chain as: $Q = \sum_{i-(dd-1)}^{i} Prod_i$. Then in the next cycle, we can retrieve $y_{i-dd}$ that started $dd$ cycles ago from the circular buffer, and add it with the pre-added result $Q$ to get the new accumulation result $y_i = Q + y_{i-dd}$. As a result, we can achieve full pipelining of the floating-point accumulation process without the need for out-of-order scheduling or bubbles in the pipeline. To implement this adder chain in hardware, as shown in Figure 8b. (i), we employ $dd - 1 = 4$ shift registers along with $dd - 1 = 4$ adders. The shift registers are used as temporary memory to hold products from the previous $dd - 1$ iterations. These products are then summed together using an adder chain consisting of $dd - 1$ adders. Additionally, we store the row indices of the products, ensuring that while adding previous products, only the ones belonging to the current row that we are operating on are included.

## 5 Preprocessing

This section outlines the major steps involved in the software preprocessing necessary to prepare the input matrix for hardware acceleration on the FPGA. The focus is primarily on the balancing algorithm, which determines the optimal processing mode for each row within a tile, and the re-ordering algorithm used to avoid the RAW conflicts of the floating-point accumulation on the output buffer for designs without the *adder chain*.

### 5.1 Balancing Algorithm

As stated in Section 4.3, the objective of the balancing algorithm (Algorithm 3) is to identify which rows within a tile should be processed in the `intra-row` mode to ensure that the workloads between the PEs are balanced. Initially (lines

1-3 in Algorithm 3), all the rows are assigned to be processed in `inter-row` mode, where each PE works on different rows. We compute the total workload of this assignment according to Equation 6.

$$\textbf{workload}_{\textbf{TOT}}(InterRows, IntraRows) = \max\{\textbf{workload}_{\textbf{PE}}(InterRows, j) \mid \forall j \in [0, P)\} + \textbf{extraload}(IntraRows) \quad (6)$$

The total workload consists of two parts; the first part is the maximum PE workload from rows assigned to be processed in `inter-row` mode, which directly correlates to the imbalance in the PE workloads. The workload of a PE *PE_ID* from rows assigned to `inter-row` mode is defined by Equation 7 as the sum of the number of nonzeros (**nnz**) of all the rows in `inter-row` mode.

$$\textbf{workload}_{\textbf{PE}}(InterRows, PE\_ID) = \sum_{\substack{r_i \in InterRows \\ r_i \% P = PE\_ID}} \textbf{nnz}(r_i) \quad (7)$$

The second part is the extra workload assigned to each PE from rows assigned to be processed in `intra-row` mode, which is defined by Equation 8 as the sum of the number of nonzeros (**nnz**) of each row in `intra-row` mode divided by the total number of PEs (*P*), as in this mode all the PEs collectively work on a single row.

$$\textbf{extraload}(IntraRows) = \sum_{r_i \in IntraRows} \left\lceil \frac{\textbf{nnz}(r_i)}{P} \right\rceil \quad (8)$$

After computing the total workload of the initial assignment, the next part of Algorithm 3 focuses on finding the row assignments that result in the lowest total workload. For each PE *p*, the workload assigned to it from `inter-row` mode rows is computed and stored as $w_p$ (lines 4-7 in Algorithm 3). For all the PEs *q* other than *p*, we keep removing the row with the largest number of nonzeros from `inter-row` mode rows and add it to the `intra-row` mode rows until the workload assigned to the PE *q* from `inter-row` mode rows, denoted by $w_q$, becomes smaller than or equal to $w_p$ (lines 8 - 14 in Algorithm 3). This method of row mode assignment provides *P* different solutions, out of which we identify the assignment with minimum workload (lines 15 - 19 in Algorithm 3) as the final assignment.

**Algorithm 3** MAD-HiSpMV balancing algorithm

```
1:  $IntraRows \leftarrow \{\}$                                          ▷ initialize as empty
2:  $InterRows \leftarrow \{\forall i \mid i \in [0, R_t)\}$             ▷ initialize with all the rows within the tile
3:  $W_{best} \leftarrow \textbf{workload}_{\textbf{TOT}}(InterRows, IntraRows)$   ▷ compute workload of initial assignment
4:  for all $p \in [0, P)$ do                                          ▷ for all the PEs p, where P = #PEs
5:      $IntraRows_p \leftarrow \{\}$
6:      $InterRows_p \leftarrow \{\forall i \mid i \in [0, R_t)\}$
7:      $w_p \leftarrow \textbf{workload}_{\textbf{PE}}(InterRows_p, p)$      ▷ compute inter-row mode workload of the current PE p
8:      for all $q \in [0, P), q \neq p$, do                             ▷ for all PEs q other than p
9:          $w_q \leftarrow \textbf{workload}_{\textbf{PE}}(InterRows_p, q)$  ▷ compute inter-row mode workload of the other PE q
10:         while $w_q > w_p$ do                                         ▷ while workload of q is larger than p
11:             $r_i \leftarrow \arg\max [\textbf{nnz}(i) \mid \forall i \in [0, R_t) \mid i\%P = q]$   ▷ Identify largest row $r_i$ assigned to q
12:             $IntraRows_p \leftarrow IntraRows_p \cup \{r_i\}$          ▷ add $r_i$ to intra-row
13:             $InterRows_p \leftarrow InterRows_p \setminus \{r_i\}$     ▷ remove $r_i$ from inter-row
14:             $w_q \leftarrow w_q - \textbf{nnz}(r_i)$                   ▷ update workload of q
15:     $W_p \leftarrow \textbf{workload}_{\textbf{TOT}}(InterRows_p, IntraRows_p)$   ▷ compute workload of the current assignment
16:     if $W_p < W_{best}$ then                                         ▷ if current assignment is better
17:         $IntraRows \leftarrow IntraRows_p$                             ▷ store current assignment
18:         $InterRows \leftarrow InterRows_p$
19:         $W_{best} \leftarrow W_p$                                      ▷ update best workload to current
```

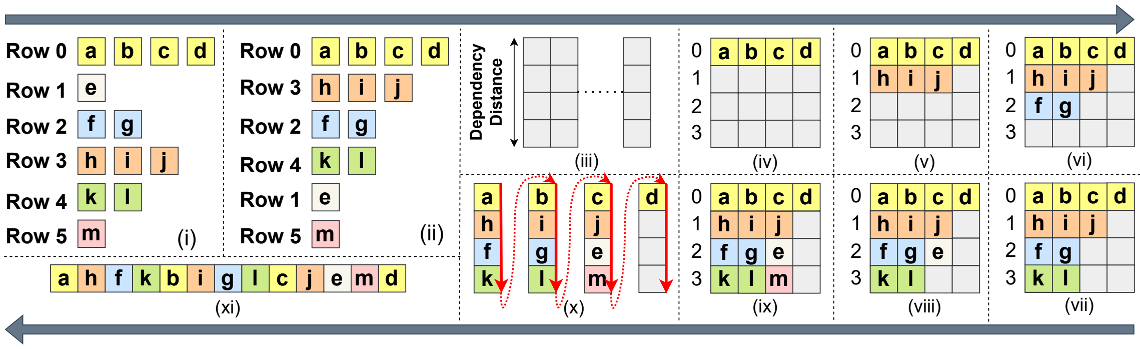


Fig. 9. Reordering algorithm for scheduling of nonzeros in (i) within a tile of a PE to ensure fixed distance (dependency distance = 4) between nonzeros of the same row, as shown in the final scheduled order (xi).

## 5.2 Re-ordering Algorithm

For some input matrices, having adder chain groups results in insignificant performance speedup compared to re-ordering, while adding extra resource overhead. Hence, for MAD-HiSpMV configurations, targeting such matrices is built without an adder chain. When the hardware does not have the adder chain, scheduling nonzeros from the same row consecutively leads to Read-After-Write (RAW) conflict on the long-latency floating-point accumulation of the output vector, as described in detail in Section 3.2. In Serpens [34], the re-ordering algorithm is used to ensure the nonzeros from the same row within a tile are scheduled with a minimum distance of $dd$ (dependency distance). Whereas, in our implementation, the nonzeros from the same row within a tile are scheduled at a fixed distance of $dd$. The constant distance condition ensures that we only need to check the entry in the circular buffer that is $dd$ distance away from the current entry when accumulating; more details are included in Section 4.4. Figure 9 depicts the scheduling of six rows assigned to a PE within a tile with a dependency distance of 4 as an example.

To schedule nonzeros of the rows assigned to a PE within a tile (Figure 9.i), we first sort the rows in descending order of number of nonzeros (Figure 9.ii). Next, we create $dd = 4$ bins and assign the rows in the sorted order to the bin with the least number of nonzeros (Figure 9.iii - 9.ix). Then the final order of the scheduling (Figure 9.ix) is formed by following the pattern that goes across the bins as shown in Figure 9.x.

## 6 Automation Tool

Figure 10 shows the complete automation flow for building the optimized MAD-HiSpMV accelerator, given the FPGA platform information, including available resources, HBM properties (such as channel width, number of channels, and frequency), and the input matrix. The design space exploration (DSE) in the tool ① in Figure 10 adopts the design configuration to maximize performance for the given matrix while ensuring the resource consumption is within the given limits. The design configuration comprises of the optimal number of HBM channels for sparse matrix $A$ ($N$), and dense vectors $\vec{x}$ ($K$), $\overrightarrow{y\_in}$ / $\overrightarrow{y\_out}$ ($M$), along with which of the optimizations among *adder chain* and *hybrid row distribution* is needed. This configuration info is then used by the code generator ② in Figure 10 to output source code for both host and kernel. The kernel source code utilizes PASTA [26] HLS, which is a task-parallel programming model with support for buffers and streams as communication channels between the tasks. PASTA ③ in Figure 10 will transform this code to Vitis HLS and compile to RTL code using Vitis (vendor tool) for each task individually; and performs the coarse-grained floorplanning for such task-parallel HLS designs, and the vendor Vivado tool ④ performs the final placement and routing.

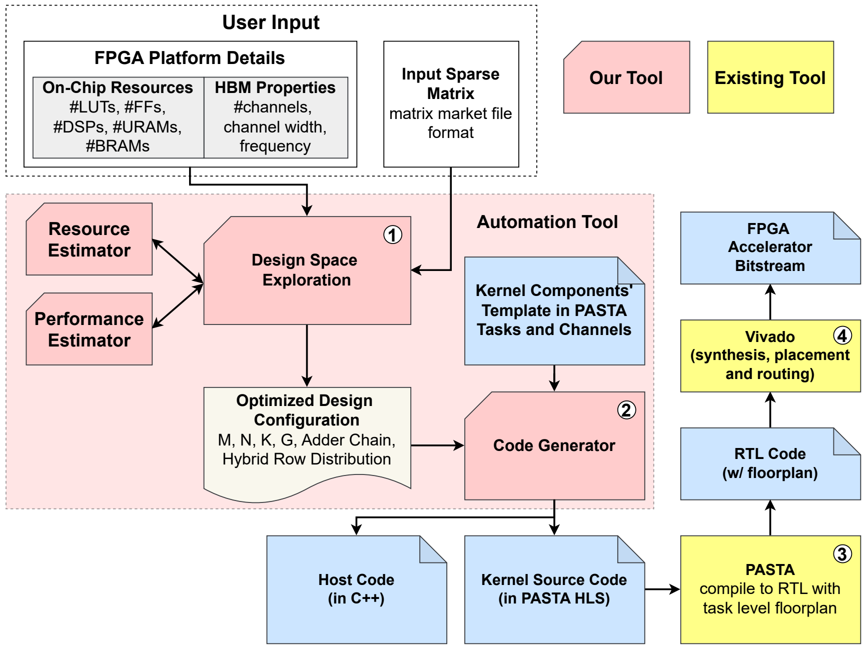

User Input
FPGA Platform Details
On-Chip Resources
#LUTs, #FFs, #DSPs, #URAMs, #BRAMs
HBM Properties
#channels, channel width, frequency
Input Sparse Matrix
matrix market file format
Our Tool
Existing Tool
Automation Tool
Resource Estimator
Performance Estimator
Design Space Exploration
1
Kernel Components' Template in PASTA Tasks and Channels
Optimized Design Configuration
M, N, K, G, Adder Chain, Hybrid Row Distribution
Code Generator
2
FPGA Accelerator Bitstream
Vivado (synthesis, placement and routing)
4
RTL Code (w/ floorplan)
PASTA
compile to RTL with task level floorplan
3
Host Code (in C++)
Kernel Source Code (in PASTA HLS)

Fig. 10. complete automation flow of MAD-HiSpMV, integrated with the recent PASTA [26] programming framework

The PASTA tool performs the coarse-grained floorplanning by abstracting the task parallel HLS design as a directed graph, with tasks as the nodes and communication channels as the edges. The key idea of coarse-grained floorplanning is to divide the datacenter FPGA into multiple smaller regions and do local place and route for each node within a smaller region to improve the timing closure. Using synthesis estimates of the tasks, the local footprint of each node is used to find an available region on the FPGA fabric. For the edges crossing two 'regions', pipeline registers are added on the edges so that the latency of all paths between two respective nodes remains the same. Finally, the edge connectivity is optimized globally such that the number of added pipeline registers and the number of edges crossing the regions are at a minimum, while also ensuring the resource utilization of all the regions is relatively balanced.

To perform the design space exploration, we build our analytical resource and performance estimators and search iteratively for different designs.

### 6.1 Resource Estimation

MAD-HiSpMV design is composed of a small set of unique tasks as shown in Figure 5, and connected with communication channels (buffers/FIFOs). When generating different configurations, only the number and types of tasks and communication channels used will change.

*6.1.1 Tasks:* By profiling the resource utilization of all different tasks of MAD-HiSpMV beforehand, the tool can estimate the overall resource utilization accurately from the configuration info. The resource consumption per instance and number of instances of all the unique tasks are presented in Table 3. Note that the resources of the *adder chain* task are only applicable if the hardware configuration is built with this optimization. Similarly, tasks like adder blocks, fused blocks, routing blocks, and switching blocks are applicable if the hardware is built with *hybrid row distribution* optimization. To add support for new FPGAs, the resource consumption of each task should be profiled and updated.

Table 3. Number of instances of different tasks in MAD-HiSpMV design, and resource consumption per instance on Ultrascale+ FPGAs at 235MHz target frequency. N, K, and M are the number of HBM channels for the spare matrix, input vector, and output vector, respectively. G is the number of PE groups per channel, F is the number of floating-point numbers per channel, and P is the total number of PEs.

| **Task** | **No. of** | **Resource Consumption per Instance** | | | | |
|---|---|---|---|---|---|---|
| | **Instances** | **LUT** | **FF** | **DSP** | **URAM** | **BRAM** |
| Read A | N | 98 | 87 | - | - | - |
| Read x | K | 59 | 103 | 1 | - | - |
| Read y_in | M | 56 | 139 | - | - | - |
| Write y_out | M | 66 | 143 | - | - | - |
| Load | N*G | 240 | 245 | - | - | - |
| PEG (w/o dense overlay) | N*G | 553 | 740 | 6 | - | - |
| PEG (w/ dense overlay) | N*G | 1410 | 1740 | 16 | - | - |
| Adder Chain Group | N*G | 2100 | 2000 | 16 | - | - |
| Adder blocks (A/A*) | P-1 | 485 | 407 | 2 | - | - |
| Fused block (X) | 1 | 485 | 407 | 2 | - | - |
| Routing blocks (R/R*) | P-1 | 82 | 129 | - | - | - |
| Switch blocks (S) | P-4 | 82 | 129 | - | - | - |
| Arbiter | 1 | 1000 | 1000 | 2 | - | - |
| y_Ax Handler | P | 849 | 686 | 3 | 2 | - |
| Compute y_out | M | (414 * F) + 75 | (587 * F) + 166 | (8 * F) + 2 | - | - |

*6.1.2 Communication Channels:* To estimate the resource consumption of the register-based FIFO (i.e, streams), we utilize the resource modeling of Autobridge [17]. For a FIFO channel of width $w$ and depth $d$, the flip-flop (FF) usage is given by

$$FF = 7 + (3 \times \lfloor \log d \rceil)$$

and the look-up-table (LUT) usage, which comprises of both LUT_LOGIC and LUT_RAM is given by

$$LUT\_LOGIC = 15 + (3 \times \lfloor \log d \rceil) \quad LUT\_RAM = w \times \left\lceil \frac{d}{16} \right\rceil.$$

For the BRAM-based buffer channels between the tasks *Load* and *PEG*, to match the throughput of the off-chip access $\vec{x}$, the buffer channel needs to have parallel access to $K \times CW$ bits, where $K$ and $CW$ are the number of HBM channels and their width used to load input vector $\vec{x}$, respectively. Since the BRAM36K blocks provide 2-port 32-bit interfaces, the total number of BRAM36K blocks per buffer channel is $\frac{K \times CW}{32 \times 2}$. There are a total of $N \times G$ buffer channels, where $N$ and $G$ are the number of HBM channels for streaming sparse matrix $A$ and PE groups per channel of $A$, respectively. Hence, the total number of BRAM blocks used by the design is given by:

$$\#BRAMs = (N \times G) * \left( \frac{K \times CW}{32 \times 2} \right)$$

For an HBM channel width ($CW$) of $512b$ as in our designs on U50 and U280, the value of $G$ will be 4 (explained in Section 4.1), the computation of the total number of BRAMs simplifies to $\#BRAMs = N \times K \times 32$.

## 6.2 Performance Estimation

In MAD-HiSpMV, the total workload can be divided into three operations, each bound by the off-chip memory throughput. The total time is given by Equation 9.

$$t_{cc} = \lceil \frac{NNZ \times \delta}{N \times 8} \rceil + \left( \lceil \frac{C}{K \times 16} \rceil \times \lceil \frac{R}{R_t} \rceil \right) + \lceil \frac{R}{M \times 16} \rceil \tag{9}$$

1). **Compute Time ($t_A$):** This is the time taken to stream in the nonzero values of the matrix **A** and compute $\mathbf{A} \times \vec{x}$, given by $t_A = \lceil \frac{NNZ}{P} \times \delta \rceil$, where $NNZ$ represents the number of nonzero elements, $P = 8 \times N$ is the number of processing elements, and $\delta$ is the imbalance ratio after optimizations.
2). **Buffering Time ($t_x$):** This is the time required to load $\vec{x}$ into on-chip buffers. It can be calculated as $t_x = \lceil \frac{C}{K \times 16} \rceil \times \lceil \frac{R}{R_t} \rceil$, where $C$ is the number of columns, and $\lceil \frac{R}{R_t} \rceil$ is the number of tiles along the rows. This operation is performed using $K$ 512-bit ports, allowing 16 (fp32) values to be loaded in parallel from each port, where there are $K$ such ports.
3). **Output Streaming Time ($t_y$):** This represents the time needed to stream in $\overrightarrow{y\_in}$, compute $\overrightarrow{y\_out} = \alpha \cdot \overrightarrow{y\_Ax} + \beta \cdot \overrightarrow{y\_in}$, and stream it out. It can be calculated as $t_y = \lceil \frac{R}{M \times 16} \rceil$, where $M$ is the number of ports used to stream in/out $\overrightarrow{y\_in}$ / $\overrightarrow{y\_out}$. Each port is 512-bit wide and can access 16 float values in parallel.

## 6.3 Design Space Exploration

In MAD-HiSpMV, we define the HBM channel constraint in Equation 10. The terms $M, N, K$ are defined earlier in Section 4.1, and the term $HC$ here corresponds to the total number of HBM channels the design is constrained to.

$$(2 \times M) + N + K \leq HC \tag{10}$$

A simple way to find optimum values for the parameters would be to do an exhaustive search for all possible values of $M, N$, and $K$ (i.e, between 1 and $HC$), along with trying designs with and without the *adder chain group* and *hybrid row distribution* optimizations. However, for a fixed value of $M$, $K$ and a given set of optimizations, the configuration with the best performance is the largest value of $N$ that satisfies the resource constraint. Furthermore, the range of possible different values of $M$ and $K$ is limited to powers of two between 1 and $HC$, which ensures that the addressing logic for on-chip access to the dense vectors only uses shifting operations (more resource efficient than integer division and remainder operators). For every design configuration that satisfies the HBM channel constraint in Equation 10 and the resource utilization limits (Table 5), we identify the configuration with the best performance (i.e, minimize total clock cycles given by Equation 9).

## 7 Evaluation

### 7.1 Experimental Setup (SpMV)

We extensively compare the performance of MAD-HiSpMV with previous HiSpMV [33] designs and state-of-the-art open-source SpMV FPGA designs such as CoSpMV+ [38], HiHiSpMV [37], Cuper [42], Serpens [34], and HiSparse [11], as well as optimized Intel MKL library on CPU [21] and Nvidia cuSparse library on GPU [31]. We evaluate them using a diverse set of 20 matrices from the widely used SuiteSparse [10], including 10 balanced (M1-M10) and 10 imbalanced matrices (M11-M20). Our selection of balanced matrices aligns with those used in Serpens [34], and their imbalance ratios are typically between 1 and 1.5. For imbalanced matrices, we deliberately choose a variety of matrices with varying imbalance ratios (more than 2 and up to 33) and densities from different application domains. Details about these matrices and their properties are presented in Table 4. We generate MAD-HiSpMV configurations for all 20 benchmark matrices on three different FPGAs, which include AMD/Xilinx U280, U50, and V80 HBM-based FPGAs. The Vitis version 2023.2 was used to build and run the designs on U280 and U50. For V80, we relied on the estimations for the comparisons, as we were unable to test the design on the hardware due to limited accessibility.

Table 4. Benchmark sparse matrices and their properties, along with imbalance ratio (for P=128 without any optimization).

| ID | Filename | Size (R=C) | NNZ | Density | Imbalance Ratio | ID | Filename | Size (R=C) | NNZ | Density | Imbalance Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **M1** | TSOPF_RS_b2383 | 38,120 | 16,171,169 | 1.11E-02 | 1.01 | **M11** | c-52 | 23,948 | 202,708 | 3.53E-04 | 2.28 |
| **M2** | crystk03 | 24,696 | 1,751,178 | 2.87E-03 | 1.01 | **M12** | language | 399,130 | 1,216,334 | 7.64E-06 | 2.29 |
| **M3** | nd6k | 18,000 | 6,897,316 | 2.13E-02 | 1.05 | **M13** | analytics | 303,813 | 2,006,126 | 2.17E-05 | 3.05 |
| **M4** | crankseg_2 | 63,838 | 14,148,858 | 3.47E-03 | 1.07 | **M14** | nxp1 | 414,604 | 2,655,880 | 1.55E-05 | 4.39 |
| **M5** | ford2 | 100,196 | 544,688 | 5.43E-05 | 1.08 | **M15** | poli_large | 15,575 | 33,033 | 1.36E-04 | 4.40 |
| **M6** | thread | 29,736 | 4,444,880 | 5.03E-03 | 1.09 | **M16** | lowThrust_7 | 17,378 | 211,561 | 7.01E-04 | 5.05 |
| **M7** | PFlow_742 | 742,793 | 37,138,461 | 6.73E-05 | 1.14 | **M17** | hangGlider_3 | 10,260 | 92,703 | 8.81E-04 | 13.47 |
| **M8** | Si41Ge41H72 | 185,639 | 15,011,265 | 4.36E-04 | 1.21 | **M18** | boyd2 | 466,316 | 1,500,397 | 6.90E-06 | 18.40 |
| **M9** | mouse_gene | 45,101 | 28,967,291 | 1.42E-02 | 1.21 | **M19** | trans5 | 116,835 | 749,800 | 5.49E-05 | 20.30 |
| **M10** | soc-Pokec | 1,632,803 | 30,622,564 | 1.15E-05 | 1.22 | **M20** | ASIC_680k | 682,862 | 2,638,997 | 5.66E-06 | 32.82 |

Table 5. HBM interface and FPGA resource usage limits in DSE of the automation tool

| FPGA | FPGA Kernel ↔ HBM Interface | | | FPGA Resource Usage Limit (%) | | | | | FPGA Total Available Resource | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Num. of Channels Usage Limit (HC) | Channel Width | Frequency | LUT | BRAM | DSP | URAM | FF | LUT | BRAM | DSP | URAM | FF |
| **U50** | 28 | 512b | 225 MHz | 70 | 75 | 70 | 70 | 70 | 870,720 | 1,344 | 5,952 | 640 | 1,743,360 |
| **U280** | 28 | 512b | 225 MHz | 62 | 75 | 70 | 70 | 62 | 1,303,680 | 2,016 | 9,024 | 960 | 2,607,360 |
| **V80** | 64 | 256b | 400 MHz | 70 | 75 | 70 | 70 | 70 | 2,574,208 | 3,741 | 10,848 | 1,925 | 5,148,416 |

## 7.2 MAD-HiSpMV Best Configurations by DSE

Table 5 gives detailed information on the HBM and FPGA kernel interface properties and the FPGA resource utilization limit we set for generating MAD-HiSpMV in the automation tool. The HBM on U280 and U50 has 32 channels, each with a bus width of 256-bit at 450 MHz. To maximize the bandwidth of each channel, we choose a channel width of 512-b and a frequency of 225 MHz according to the benchmark study [30]. Although the HBM on U280 and U50 has 32 channels, we limit the total number of channels to 28, as we observe that the designs above this were often not able to reach the target frequency consistently. The FPGA resource limitations were set based on the collective observation of different designs that were able to achieve the target frequency consistently with our flow. The HBM on V80 has 16 channels, each with a bus width of 128-bit at 3.2 GHz. To fully utilize the bandwidth, each HBM channel can be shared by four 256-bit AXI ports at 400MHz (128-bit * 3.2GHz/400MHz / 4 ports = 256-bit). Each AXI port provides up to 256-bit * 400MHz = 102.4Gb/s = 12.8GB/s bandwidth, and 64 AXI ports use up all available HBM bandwidth. The FPGA resource limits were set to the maximum we were able to utilize on U280 and U50, as V80 is equipped with a dedicated NoC operating at 1GHz for better routability. We present the MAD-HiSpMV configurations on U280, U50, and V80 generated by the DSE for the benchmark matrices in Table 6. Note that V80 has a hardened floating-point (fp32) unit, DSP58 [1], which can process one floating-point multiply/add in one cycle. Therefore, it eliminates the necessity of the *adder chain* optimization, which was used to overcome long-latency floating-point accumulation. Additionally, the tool can also generate a dense overlay design when specified by the user. We build these designs on both U280 (Config-5) and U50 (Config-4) with the configurations shown in Table 6.

## 7.3 Automation Tool Estimation Accuracy

*7.3.1 Resource Estimation Accuracy:* Figure 11a depicts the errors in the resource estimation of the automation tool (explained in Section 6) for U280 and U50 configs in Table 6. The resources estimated for BRAM, DSP, and URAM are highly accurate as their computation is very simple and unaffected by placement and routing. Albeit, there is one minor outlier for the DSP estimation, caused by Vitis HLS opting to implement the FP accumulator with LUTs instead of DSP, despite explicit instructions to use DSP. In contrast, the resources LUTs and FFs exhibit greater variability due to

Table 6. DSE picked best MAD-HiSpMV configurations on U280, U50 and V80 for benchmark matrices along with configurations with dense overlay on U280 and U50

| Matrices | Config-Id | Dense Overlay | Adder Chain Group | Hybrid Row Distr. | N | K | M | G | P (#PEs) |
|---|---|---|---|---|---|---|---|---|---|
| M5, M11-15, M17-20 | U280 Config-0 | ✗ | ✓ | ✓ | 16 | 2 | 4 | 4 | 128 |
| M10, M16 | U280 Config-1 | ✗ | ✗ | ✓ | 20 | 2 | 2 | 4 | 160 |
| M7 | U280 Config-2 | ✗ | ✓ | ✗ | 20 | 2 | 2 | 4 | 160 |
| M4, M6, M8, M9 | U280 Config-3 | ✗ | ✗ | ✓ | 24 | 1 | 1 | 4 | 192 |
| M1, M2, M3 | U280 Config-4 | ✗ | ✓ | ✗ | 24 | 1 | 1 | 4 | 192 |
| Dense | U280 Config-5 | ✓ | ✗ | ✓ | 24 | 1 | 1 | 4 | 192 |
| M2, M7, M10, M16-17 | U50 Config-0 | ✗ | ✓ | ✓ | 8 | 2 | 4 | 4 | 64 |
| M5, M11, M13-14, M18-20 | U50 Config-1 | ✗ | ✓ | ✓ | 12 | 2 | 2 | 4 | 96 |
| M1, M3, M4, M6, M8-9 | U50 Config-2 | ✗ | ✗ | ✓ | 16 | 1 | 4 | 4 | 128 |
| M12, M15 | U50 Config-3 | ✗ | ✗ | ✓ | 18 | 1 | 1 | 4 | 144 |
| Dense | U50 Config-4 | ✓ | ✗ | ✓ | 16 | 1 | 2 | 4 | 128 |
| M5, M11-15, M17-20 | V80 Config-0 | ✗ | ✗ | ✓ | 32 | 8 | 8 | 2 | 128 |
| M16 | V80 Config-1 | ✗ | ✗ | ✓ | 40 | 8 | 4 | 2 | 160 |
| M2, M4, M6-8, M10 | V80 Config-2 | ✗ | ✗ | ✓ | 56 | 4 | 2 | 2 | 224 |
| M1, M3, M9 | V80 Config-3 | ✗ | ✗ | ✓ | 60 | 2 | 1 | 2 | 240 |

placement and routing optimization; hence, these estimations have a larger error range compared to other resources. Nevertheless, these errors are still within the magnitude of 4%, with even lower deviations observed in the majority of the designs.

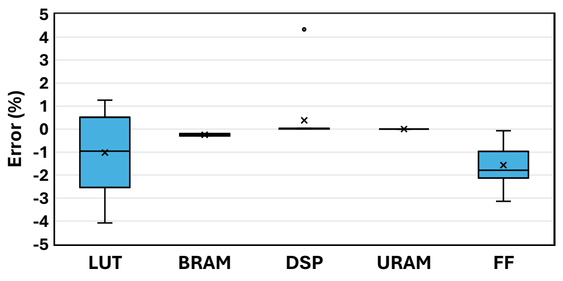
*7.3.2 Execution Time Estimation Accuracy:* The performance estimation error of the automation tool for all 20 benchmark matrices on U280 and U50 FPGAs is shown in Figure 11b. The estimation on devices has an average error of −6%, which indicates that the tool is underestimating the total execution time. This underestimation is due to the lack of consideration of initiation latency in the performance estimation given by Equation 9. However, for the configuration, U280 Config-4, the error of $M1$ and $M3$ matrices is close to 0%, and the error of $M2$ is 4%, unlike all the other configurations, which are very close to −6%. When we repeat the kernel for multiple iterations to amortize the kernel invocation overhead and get accurate kernel execution, the use of a larger FIFO depth in this particular design between PEGs and y_AX handlers leads to a slight overlap between the consecutive kernel iterations, hence leading to the measured kernel time to be lower than the estimated value.

## 7.4 Performance of MAD-HiSpMV and Prior FPGA Studies (SpMV)

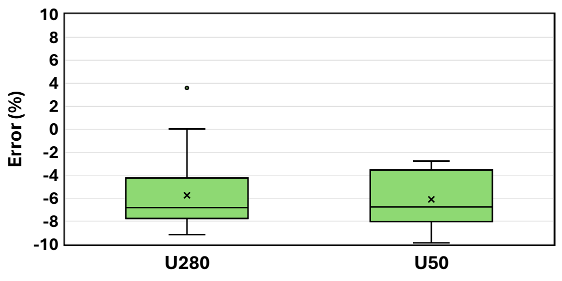
The formula we use to compute GFLOPS is given by:

$$GFLOPS = \frac{2 \times (NNZ + R)}{10^9 \times \text{execution time}}$$

which is a direct reflection of the execution time as $NNZ + R$ comes from the matrix and remains the same across implementations. We conducted extensive benchmarking, running the matrices with kernels from prior works such as HiSparse [11], Serpens [34], our previous generic designs (HiSpMV-16 and HiSpMV-20) on U280 published in FPGA 2024 [33], and recent works such as Cuper [42], HiHiSpMV [37], and CoSpMV+ [38], as well as our own MAD-HiSpMV

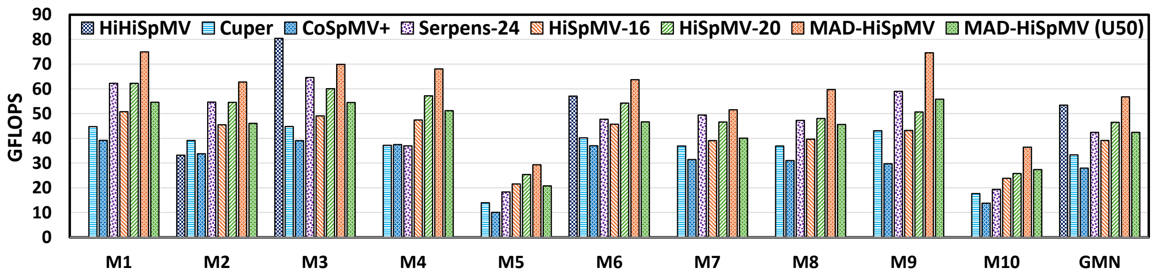


(a) Resource Estimation

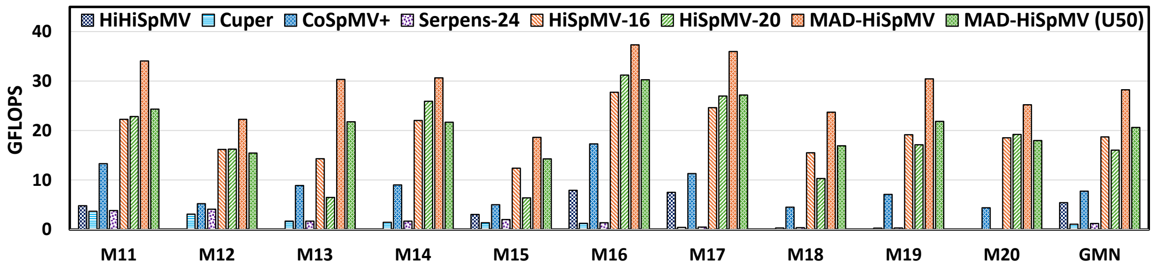


(b) Performance Estimation

Fig. 11. Automation tool resource and performance estimation error on U50 and U280. Here, '·' is the outliers, 'x' represents the median, the boxes are drawn from the 25th percentile to the 75th percentile, the horizontal line inside the box is the mean value, and the error bars represent the range of the data (excluding the outliers).



(a) SpMV kernels with balanced matrices on U280 FPGA (unless specified otherwise)



(b) SpMV kernels with imbalanced matrices on U280 FPGA (unless specified otherwise)

Fig. 12. Comparison of SpMV kernels on U280 FPGA (unless specified otherwise)

DSE picked best configurations on U280 and U50. We repeat the kernel multiple times for a single kernel call from Xilinx Runtime (XRT) to ensure the overhead of the XRT call is amortized.

The performance comparisons are illustrated in Figure 12a for balanced matrices and Figure 12b for imbalanced matrices. For balanced matrices, our MAD-HiSpMV DSE picked best configurations (on U280) achieves a geomean speedup of:

- 5.81x (up to 10.24x) and 5.76x (up to 13.87x) over HiSparse-RI and HiSparse-PB designs [11]
- 1.71x (up to 2.21x) and 1.34x (up to 1.88x) over Serpens-16 and Serpens-24 designs [34]
- 1.45x (up to 1.73x) and 1.22x (up to 1.47x) over HiSpMV-16 and HiSpMV-20 designs [33] (our previous designs)
- 1.06x (up to 1.89x), 1.7x (up to 2.1x), and 2.03x (up to 2.9x) over the recent works HiHiSpMV [37], Cuper [42], and CoSpMV+ [38], respectively

Furthermore, for imbalanced matrices, our MAD-HiSpMV DSE picked best configurations (on U280) achieves a geomean speedup of:

- 33.26x (up to 169.22x), and 25.27x (up to 37.16x) over HiSparse-RI, and HiSparse-PB designs [11]
- 23.07x (up to 98.26x), and 23.00x (up to 97.98x) over Serpens-16, and Serpens-24 designs [34]
- 1.51x (up to 2.12x), and 1.76x (up to 4.69x) over HiSpMV-16, and HiSpMV-20 designs [33] (our previous designs)
- 5.20x (up to 7.08x), 26.79x (up to 110.87x), and 3.43x (up to 5.27x) over the recent works HiHiSpMV [37], Cuper [42], and CoSpMV+ [38], respectively

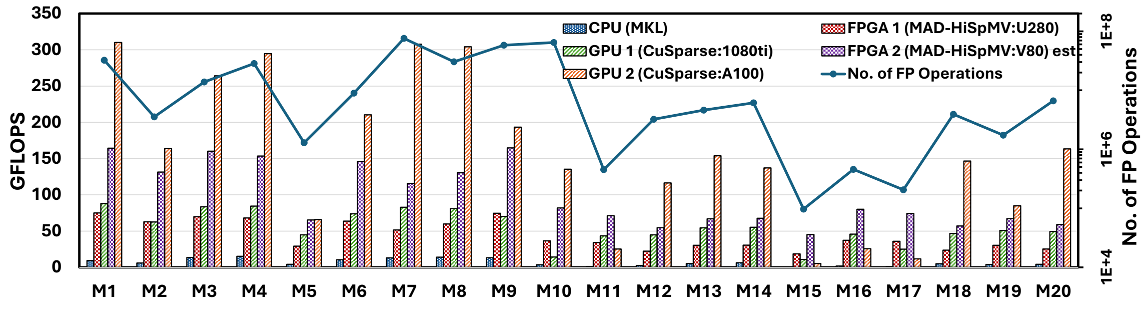
Notably, the HiHiSpMV [37] design, which outperforms our design by 1.15x on M3, was only able to execute for 3 balanced and 4 imbalanced matrices from our benchmark matrices. The said design can outperform our designs, as it utilizes all 32 HBM channels on U280. HiHiSpMV uses 30% LUTs, 57% BRAMs, and 13% URAMs, compared to our design's resource utilization, which goes up to 60% LUTs, 73% BRAMs, and 40% URAMs. The HiHiSpMV design encounters lower routing congestion and hence, it can utilize all HBM channels. Due to the high resource utilization and high congestion in the bottom SLR, our designs are constrained to utilize only up to 28 channels. However, HiHiSpMV does not support tiling and can only support smaller matrices (up to row size of 40K). Whereas our design can support up to a row size of 1.5M in a single tile and can support multiple tiles, the only limitation for the row size is the storage limitation of the HBM. Moreover, for imbalanced matrices, our designs outperform other works (including the HiHiSpMV), justifying the importance and effectiveness of our work.

Moreover, our MAD-HiSpMV DSE-picked best configurations on U50 and achieves a geomean performance of 0.74x (from 0.69x to 0.81x) compared to the U280 designs, which correlates with the fact that U50 has 0.67x of the total resources in U280, and due to the smaller area (reduced placement and routing complexity) it achieves slightly higher relative utilization than U280 designs.

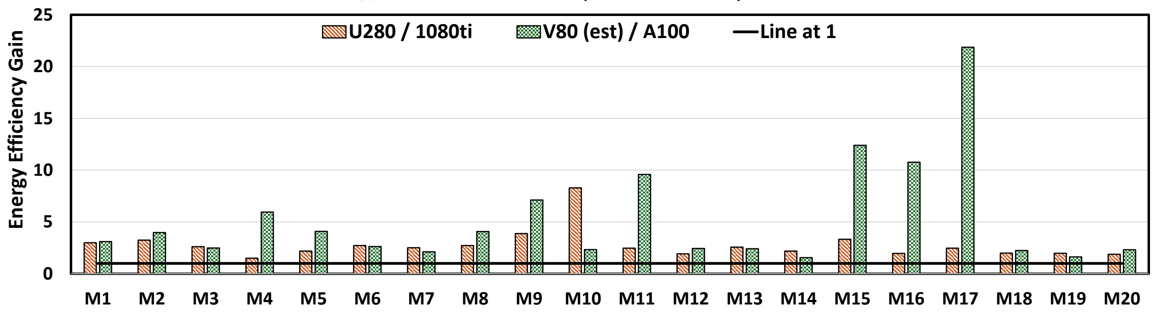
The peak performance of MAD-HiSpMV on U280 for balanced matrices is approximately 75 GFLOPS, and for imbalanced matrices is approximately 36 GFLOPS, while the theoretical peak throughput is 86 GFLOPS. These losses come from the time taken by the hardware to load the input vector and update the output vector, since the actual computation and these stages run sequentially, one after another. As the density of the matrix decreases, this loss increases.

## 7.5 Comparison to CPU and GPU (SpMV)

We conducted benchmarking on both CPU and GPU platforms using the Intel MKL library [21] on a Xeon Silver 4214 CPU with 24 cores and the Nvidia cuSparse library [31] on GTX 1080ti and A100 GPUs. For a fair GPU and FPGA comparison, we compare 1) the 16nm U280 FPGA with 460GB/s bandwidth against the 16nm 1080ti GPU with 484.4GB/s bandwidth, and 2) the 7nm Versal (largest HBM-based) V80 FPGA [3] with 819.2GB/s bandwidth against the 7nm A100 GPU with 1,555GB/s bandwidth. Besides the V80 FPGA, all performance and power results are measured on the actual boards. The board power consumption is measured using vendor-provided APIs (*NVML* for GPUs and *XRT* for FPGAs) during the kernel runtime. All the kernels on FPGA were repeatedly run for 60 seconds to get the power consumption and performance. For GPU measurement, we repeated the kernel 10000 times to get power and performance metrics, as it was not possible to know how many times to repeat the kernel to get it to run exactly 60 seconds. The reason for repeating the kernels is to make sure the power consumption reaches a stable value, as the power sensors on these boards have a very low refresh rate.



(a) CPU, FPGA, and GPU performance comparison



(b) Energy efficiency improvement of FPGA over GPU

Fig. 13. Performance and Energy Efficiency Comparison

*7.5.1 Projection for V80 FPGA Results:* Based on the MAD-HiSpMV configurations of V80, we use the performance estimation given by Equation 9 explained in Section 6.2, along with an additional 5% to account for the estimation errors observed on the U280 and U50 as presented in Figure 11b. For the power consumption, we build scaled-down designs with 3 different scaling factors (linearly spaced) for each configuration of V80 in Table 6 on U280 with channel-width of 256-b and kernel frequency of 400MHz. We measure the power consumption of these designs on the corresponding benchmark matrices and extrapolate the power for larger designs by assuming the power scales linearly with the design size (i.e., $M, N, K$). To account for the different technology nodes of V80 (7nm) compared to U280 (16nm), we reduce the power by 65%, as observed in the case study conducted by Taiwan Semiconductor Manufacturing Company (TSMC) [8].

*7.5.2 Comparison Results:* Figure 13a compares the performance (GFLOPS) between Intel MKL, NVIDIA cuSparse on 1080ti (GPU 1) and A100 (GPU 2), MAD-HiSpMV on U280 (FPGA 1) and V80 estimate (FPGA 2). Compared to the CPU, MAD-HiSpMV on U280 achieves a geomean speedup of 6.19x (from 3.95x up to 10.59x) for balanced matrices and 12.52x (from 4.78x up to 71.65x) for imbalanced matrices.

**1080ti GPU vs. U280 FPGA.** On average (geomean), U280 achieves 0.81x (from 0.5x up to 2.57x) performance of the 1080ti GPU, over all the benchmark matrices. On the other hand, Figure 13b compares their energy efficiency in terms of GFLOPS/Watt. U280 achieves better energy efficiency over 1080ti across all matrices; the geomean energy efficiency improvement is 2.57x (from 1.5x up to 8.28x). The marginally lower performance of U280 can be attributed to the underutilization of the available HBM bandwidth. To start with, the 1080ti has slightly higher memory bandwidth. Additionally, the on-chip routing resources of the U280 fail to utilize all the HBM channels for larger designs. We anticipate the hardened Network-on-Chip (NoC) in V80 to address this issue and better utilize all the available bandwidth from its HBM.

Table 7. GeMV performance per watt comparison of CPU and FPGA

| Square Matrix | CPU (MKL 24 threads) | | | FPGA (MAD-HiSpMV U280) | | |
|---|---|---|---|---|---|---|
| Size | GFLOPS | Power (watts) | GFLOPS/Watt | GFLOPS | Power (watts) | GFLOPS/Watt |
| **512** | 85.34 | 57.6 | 1.48 | 108.28 | 55.59 | 1.95 |
| **1,024** | 177.21 | 59.2 | 2.99 | 119.1 | 55.4 | 2.15 |
| **2,048** | 235.97 | 62.07 | 3.80 | 137.29 | 55.03 | 2.49 |
| **4,096** | 107.78 | 80.58 | 1.34 | 141.11 | 54.6 | 2.58 |
| **8,192** | 78.58 | 78.73 | 1.00 | 144.91 | 53.1 | 2.73 |

**A100 GPU vs. V80 FPGA.** On average (geomean), V80 achieves 0.85x (from 0.36x up to 8.45x) performance of the A100 GPU, for all of the benchmark matrices. However, the energy efficiency of V80 is, on average (geomean) 3.88x (from 1.55x up to 21.86x) better than the A100. Even with a nearly 2x smaller HBM bandwidth, V80 achieves only marginally lower performance on average. And for certain imbalanced matrices M11, M15, M16, and M17, V80 has better performance than A100. The reason is that these matrices have a much lower number of floating-point (FP) operations (shown in the line of Figure 13a); thus, A100 has lower utilization of GPU cores and larger performance overhead. This also leads to the A100 performing worse than the older 1080ti GPU (with fewer cores) for these matrices. On average, the A100 GPU has improved the performance over the 1080 Ti GPU by 2.13x, while the bandwidth has increased nearly 4x. Meanwhile, the V80 FPGA has improved its performance over the U280 FPGA by 2.23x with only a 2x bandwidth increase.

## 7.6 Performance of MAD-HiSpMV on GeMV and Mixed Workloads

*7.6.1 GeMV Performance Comparison:* To evaluate the performance of our design on GeMV, we build MAD-HiSpMV configurations with a dense overlay on both U280 and U50. The detailed configuration of the designs is presented in Table 6 (U280 Config-5 and U50 Config-4). We compare the performance of our aforementioned configurations against AMD/Xilinx Vitis L2 GeMV on U50 and U280 for square matrices of sizes 512x512, 1024x1024, 2048x2048, 4096x4096, and 8192x8192. The performance numbers for Vitis L2 GeMV on U280 and U50, with the number of HBM channels for streaming dense matrix ($N$) set to 16, are taken from the official Vitis benchmark [2], and the performance number for Vitis L2 GeMV on U280 with $N = 32$ from a recent work [13]. We also included the Intel MKL GeMV kernel's performance on a 24-core Xeon Silver 4214 CPU for the same matrix dimensions.

The GFLOPS achieved by all the different designs are presented in Figure 14. The performance of the designs on the FPGA starts low for smaller matrix dimensions and increases along with the matrix dimensions, eventually saturating at their peak performance. Whereas, in the case of the CPU kernel, the performance peaks at an intermediate matrix size of 2048x2048, which is likely due to the optimum usage of cache memory. The CPU outperforms our FPGA designs by 1.31x and 1.55x for the dimensions 1024x1024 and 2048x2048, respectively, while our designs outperform the CPU at both smaller and larger dimensions by 1.43x, 1.41x, and 1.99x for 512x512, 4096x4096, and 8192x8192, respectively. When compared to the Vitis L2 GeMV design, our design consistently performs better, and at peak performance, our design achieves a speedup of 2.6x compared to Vitis L2 GeMV for both U50 and U280. Additionally, we also measured the average power consumption of both MKL (24 threads) and MAD-HiSpMV on U280 for different sizes of the input matrix. The GeMV kernel for all the sizes was run continuously for 60 seconds on both devices to get the reliable power measurements presented in Table 7. The power consumption on the CPU increases along with the size of the matrix, likely due to the increased memory transfers. However, the power consumption on the FPGA stays almost the same

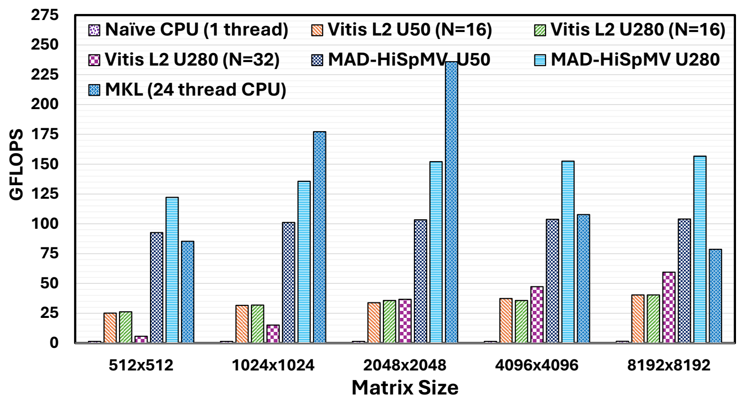


Fig. 14. GeMV performance comparison of MAD-HiSpMV designs with Vitis L2 GeMV (on U50 and U280) and Intel MKL on 24-core Xeon Silver 4214 CPU.

for all the sizes, hence making FPGAs high performing and energy efficient at larger matrix sizes than the CPU. For GFLOPS/Watt, we observe that the MKL on CPU outperforms our design on U280 by 1.39x and 1.52x for the matrix sizes of 1024x1024 and 2048x2048, respectively, while our design on U280 outperforms MKL on CPU by 1.31x, 1.93x, and 2.73x for the matrix sizes 512x512, 4096x4096, and 8192x8192, respectively. These results highlight that, while MKL achieves higher raw throughput for medium-sized matrices, MAD-HiSpMV delivers superior performance and performance-per-watt at larger matrix sizes, with energy efficiency gains becoming increasingly significant at larger problem sizes.

*7.6.2 Mixed Workload Performance Comparison:* For evaluating the end-to-end performance of a mixed workload, we profiled the inference time of a demo neural network model with three linear layers, including one dense layer and two sparse layers. We implement the model using PyTorch [32]. For inference on CPU, we use the default PyTorch implementation of the dense layer as it is well optimized. Whereas for sparse layers, the *sparse.mm* from PyTorch is not very well optimized, hence we utilize a more optimized MKL sparse dot implementation [22]. For inference on FPGA, we implement a custom layer for performing both sparse and dense linear layers, which handles all the data transfers between CPU and FPGA, as well as kernel invocation. The detailed layer configurations, as well as inference time of different implementations for batch sizes varying from 1 to 8, are presented in Table 8.

Note that for linear layers with a batch size greater than 1, the underlying operation is a matrix-matrix multiplication (matmul) instead of matrix-vector multiplication. To support a batch size greater than 1, we invoke the FPGA kernel multiple times. Since the CPU (both PyTorch and MKL) implementations are already using matmul kernels that have better data reuse, the speed-up of FPGA over CPU decreases (Table 8) as the batch size increases. Although our design is at a disadvantage for large batch sizes, it performs 224.13x better than the CPU implementation (with *sparse.mm*), and retains a better overall performance than the MKL sparse dot implementation for small batch sizes. When considering individual layers, for the dense layer (layer 1), the FPGA performs better than the CPU (PyTorch) for small batch sizes. For the sparse layer with a density of 0.1 (layer 2), the FPGA consistently outperforms the CPU (MKL sparse dot), which can be attributed to the relatively larger dimensions. For the sparse layer with a density of 0.25 (layer 3), the FPGA holds a better performance for small batch sizes. While our current design is best suited for a batch size of 1, in the future, we plan to extend the design to support matmuls that can better utilize the data reuse of larger batch sizes.

Table 8. End-to-end mixed workload performance comparison of MAD-HiSpMV on U280 vs PyTorch and Intel MKL on 24-core Xeon Silver 4214 CPU.

| **Batch Size** | **Inference Latency (in milliseconds)** | | | | | | | | **Overall Speed Up (over PyTorch)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Layer 1 Dense Linear activation=ReLU in_features=4096 out_features=8192** | | **Layer 2 Sparse Linear (density=0.1), activation=ReLU in_features=8192 out_features=8192** | | | **Layer 3 Sparse Linear (density=0.25) activation=ReLU in_features=8192 out_features=1024** | | | | |
| | **PyTorch (CPU)** | **MAD-HiSpMV (FPGA)** | **PyTorch sparse.mm (CPU)** | **MKL dot (CPU)** | **MAD-HiSpMV (FPGA)** | **PyTorch sparse.mm (CPU)** | **MKL dot (CPU)** | **MAD-HiSpMV (FPGA)** | **MKL dot (CPU)** | **MAD-HiSpMV (FPGA)** |
| 1 | 1.782 | 0.589 | 250.923 | 0.996 | 0.351 | 3.152 | 0.298 | 0.201 | 83.17x | **224.13x** |
| 2 | 1.904 | 1.276 | 265.834 | 1.729 | 0.734 | 3.403 | 0.633 | 0.406 | 63.57x | **112.22x** |
| 4 | 3.651 | 2.239 | 266.905 | 3.208 | 1.266 | 3.499 | 1.080 | 0.709 | 34.52x | **65.02x** |
| 8 | 4.775 | 4.467 | 284.287 | 5.163 | 2.552 | 5.092 | 1.622 | 1.357 | 25.45x | **35.12x** |

Table 9. Resource utilization and frequency of MAD-HiSpMV configurations on U280 and U50. Effective frequency is the lowest of the kernel frequency and half the HBM frequency.

| **Config-Id** | **Design Configuration Info** | | | | | | | **Frequency (MHz)** | | | **Resource Utilization (%)** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Dense Overlay** | **Adder Chain** | **Hybrid Row Distr.** | **N** | **K** | **M** | **G** | **HBM** | **Kernel** | **Effective** | **LUT** | **BRAM** | **DSP** | **URAM** | **FF** |
| **U280 Config-0** | × | ✓ | ✓ | 16 | 2 | 4 | 4 | 446 | 235.0 | **223.0** | 60.1 | 60.7 | 28.5 | 26.7 | 29.6 |
| **U280 Config-1** | × | × | ✓ | 20 | 2 | 2 | 4 | 430 | 221.0 | **215.0** | 55.0 | 73.4 | 20.7 | 33.3 | 26.3 |
| **U280 Config-2** | × | ✓ | × | 20 | 2 | 2 | 4 | 450 | 229.0 | **225.0** | 58.9 | 73.4 | 27.8 | 33.3 | 28.5 |
| **U280 Config-3** | × | × | ✓ | 24 | 1 | 1 | 4 | 450 | 235.0 | **225.0** | 54.1 | 48.0 | 22.8 | 40.0 | 28.2 |
| **U280 Config-4** | × | ✓ | × | 24 | 1 | 1 | 4 | 446 | 210.5 | **210.5** | 60.4 | 48.0 | 31.3 | 40.0 | 31.6 |
| **U280 Config-5** | ✓ | × | ✓ | 24 | 1 | 1 | 4 | 447 | 235.0 | **223.5** | 61.1 | 48.0 | 33.4 | 40.0 | 32.5 |
| **U50 Config-0** | × | ✓ | ✓ | 8 | 2 | 4 | 4 | 413 | 235.0 | **206.5** | 55.6 | 51.7 | 26.0 | 20.6 | 28.5 |
| **U50 Config-1** | × | ✓ | ✓ | 12 | 2 | 2 | 4 | 419 | 229.0 | **209.5** | 70.0 | 70.8 | 30.3 | 30.6 | 34.1 |
| **U50 Config-2** | × | × | ✓ | 16 | 1 | 4 | 4 | 450 | 235.0 | **225.0** | 66.4 | 51.7 | 30.3 | 40.6 | 34.6 |
| **U50 Config-3** | × | × | ✓ | 18 | 1 | 1 | 4 | 422 | 235.0 | **211.0** | 66.3 | 56.5 | 26.5 | 45.6 | 34.1 |
| **U50 Config-4** | ✓ | × | ✓ | 16 | 1 | 2 | 4 | 410 | 239.0 | **205.0** | 70.6 | 51.7 | 32.4 | 40.6 | 38.0 |

## 7.7 Resource Utilization and Frequency

The resource utilization and frequencies achieved by all the configurations of U280 and U50 are listed in Table 9. To fully leverage the maximum bandwidth of the HBM (at 450MHz) with 512-bit AXI ports, our kernels only need to achieve a frequency of 225MHz [30]. The primary resource constraints in our current designs are LUTs and BRAMs, with utilization reaching up to 70% for LUTs on U50 and 73.4% for BRAMs on U280. All the designs still achieve frequencies very close to this requirement, thanks to the task parallel design with FIFO/buffer channels enabled by the PASTA framework [26]. Note that the resource utilization presented in Table 9 also includes the Vitis platform shell's resources, which are required for the HLS Kernels to run on the FPGA. While the designs on U50 reach higher relative utilization than U280, the absolute utilization of designs on U280 is higher, as U280 has 1.5x more resources than U50.

## 8 Conclusion and Future Work

In this work, we conducted an in-depth analysis of the new challenges involved in accelerating imbalanced SpMV on HBM-based FPGAs and handling mixed workloads with both GeMV and SpMV. To address these challenges, we proposed the MAD-HiSpMV architecture. First, we designed a hybrid row distribution network to achieve a more balanced workload partition via both inter-row and intra-row distribution. Second, we implemented two techniques—a register-based circular buffer and an adder chain—to achieve fully pipelined floating-point accumulation. Third, we developed an automation tool equipped with design space exploration, which configures the design based on the input matrix to provide the best performance. In addition, we enhanced the design to perform GeMV with the same kernel. Extensive experimental results demonstrated the performance advantage of our design over state-of-the-art generic SpMV accelerators on FPGA (such as HiSpMV, HiHiSpMV, CoSpMV+, Cuper, Serpens, HiSparse, and AMD design) and Intel MKL on CPU, as well as energy efficiency gains over the Nvidia cuSparse on GPU. Moreover, the GeMV performance of our overlay design achieves 2.6x better performance than the Vitis L2 GeMV design, and for an end-to-end mixed workload, our design performs 2.7x better than a 24-core Xeon Silver 4214 CPU.

**Future Work:** While the current work focuses on matrix-vector multiplication, as discussed in Section 7.6.2, the workloads in machine learning and deep learning often require support for matrix-matrix multiplication. Additionally, the bigger models are quantized during inference to run at a lower precision. However, our design currently only supports full precision floating-point format. In the future, we aim to add support for different data types and sparse matrix-matrix multiplication, along with building the actual hardware on the latest V80 FPGA.

Currently, the hardware accelerator relies on software preprocessing to encode the sparse matrix in a custom format. While this is acceptable for applications where the matrix is reused multiple times, it would be a bottleneck for applications where the sparse matrix is used only once. Hence, further work should be done to minimize and accelerate the preprocessing of the sparse matrix.

## Acknowledgments

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund, and Paderborn Center for Parallel Computing (for GPU access), Germany.

## References

[1] AMD. 2022. Versal ACAP DSP Engine Architecture Manual (AM004). https://docs.xilinx.com/r/en-US/am004-versal-dsp-engine/DSPFP32-Unisim-Primitive Last accessed Dec 23, 2023.

[2] AMD. 2022. Vitis BLAS library L2 GEMV benchmark. https://xilinx.github.io/Vitis_Libraries/blas/2021.2/user_guide/L2/L2_benchmark_gemv.html Last accessed March 3, 2025.

[3] AMD. 2023. Versal™ Architecture and Product Data Sheet: Overview (DS950). https://docs.xilinx.com/v/u/en-US/ds950-versal-overview Last accessed Dec 23, 2023.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

[5] John T. Betts. 2010. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming, Second Edition* (second ed.). Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718577

[6] Roland Bulirsch, Edda Nerz, Hans Josef Pesch, and Oskar von Stryk. 1993. *Combining Direct and Indirect Methods in Optimal Control: Range Maximization of a Hang Glider.* Birkhäuser Basel, Basel, 273–288. doi:10.1007/978-3-0348-7539-4_20

[7] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. doi:10.1145/3431920.3439290

[8] Richard Chung. 2018. TSMC's Industry-first and Leading 7nm Technology Enters Volume Production. https://esg.tsmc.com/en/update/innovationAndService/caseStudy/9/index.html

[9] Connor W. Coley, Wengong Jin, Luke Rogers, Timothy F. Jamison, Tommi S. Jaakkola, William H. Green, Regina Barzilay, and Klavs F. Jensen. 2019. A graph-convolutional neural network model for the prediction of chemical reactivity. *Chem. Sci.* 10 (2019), 370–377. Issue 2. doi:10.1039/C8SC04228D

[10] 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. doi:10.1145/2049662.2049663

[11] 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* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. doi:10.1145/3490422.3502368

[12] I. S. Duff, Roger G. Grimes, and John G. Lewis. 1989. Sparse Matrix Test Problems. *ACM Trans. Math. Softw.* 15, 1 (mar 1989), 1–14. doi:10.1145/62038.62043

[13] Federico Favaro, Ernesto Dufrechou, Juan P Oliver, and Pablo Ezzatti. 2024. Evaluation of dense and sparse linear algebra kernels in FPGAs. *XI Southern Programmable Logic Conference* (04 2024). https://www.researchgate.net/publication/379832256_Evaluation_of_dense_and_sparse_linear_algebra_kernels_in_FPGAs

[14] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. arXiv:1902.09574 [cs.LG]

[15] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 64–67. doi:10.1109/FCCM.2015.30

[16] 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. doi:10.1145/3609335

[17] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 81–92. doi:10.1145/3431920.3439289

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

[19] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[20] 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)*. 1–9. doi:10.1109/ICCAD51958.2021.9643582

[21] Intel. 2023. Intel-Optimized Math Library for Numerical Computing on CPUs & GPUs. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html Last accessed Oct 1, 2023.

[22] Chris Jackson. 2025. sparse-dot-mkl: Accelerated sparse matrix multiplication using Intel MKL. https://pypi.org/project/sparse-dot-mkl/ Version 0.9.8, Last accessed: March 18, 2025.

[23] 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)*. 127–132. doi:10.1109/FPL50879.2020.00031

[24] Abhishek Kumar Jain, Chirag Ravishankar, Hossein Omidian, Sharan Kumar, Maithilee Kulkarni, Aashish Tripathi, and Dinesh Gaitonde. 2023. Modular and Lean Architecture with Elasticity for Sparse Matrix Vector Multiplication on FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–143. doi:10.1109/FCCM57271.2023.00023

[25] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC.2016.7761646

[26] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 12–22. doi:10.1109/FCCM57271.2023.00011

[27] 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)*. 1–9. doi:10.1109/ICCAD51958.2021.9643453

[28] 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* (Tokyo, Japan) *(ASPDAC '23)*. Association for Computing Machinery, New York, NY, USA, 33–38. doi:10.1145/3566097.3567839

[29] Jun Liu, Shulin Zeng, Li Ding, Widyadewi Soedarmadji, Hao Zhou, Zehao Wang, Jinhao Li, Jintao Li, Yadong Dai, Kairui Wen, Shan He, Yaqi Sun, Yu Wang, and Guohao Dai. 2025. FlightVGM: Efficient Video Generation Model Inference with Online Sparsification and Hybrid Precision on FPGAs. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 2–13. doi:10.1145/3706628.3708864

[30] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 105–115. doi:10.1145/3431920.3439284

[31] Nvidia. 2023. Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs. https://docs.nvidia.com/cuda-libraries/index.html Last accessed Oct 1, 2023.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[33] 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* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 154–164. doi:10.1145/3626202.3637557

[34] 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* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 211–216. doi:10.1145/3489517.3530420

[35] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 531–543. doi:10.1109/HPCA.2018.00052

[36] Endri Taka, Ning-Chi Huang, Chi-Chih Chang, Kai-Chiang Wu, Aman Arora, and Diana Marculescu. 2025. Systolic Sparse Tensor Slices: FPGA Building Blocks for Sparse and Dense AI Acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 159–171. doi:10.1145/3706628.3708867

[37] Abdul Rehman Tareen, Marius Meyer, Christian Plessl, and Tobias Kenter. 2024. HiHiSpMV: Sparse Matrix Vector Multiplication with Hierarchical Row Reductions on FPGAs with High Bandwidth Memory. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 32–42. doi:10.1109/FCCM60383.2024.00014

[38] Minghao Tian, Yue Liang, Bowen Liu, and Dajiang Liu. 2025. CoSpMV: Towards Agile Software and Hardware Co-design for SpMV Computation. *IEEE Trans. Comput.* (2025), 1–14. doi:10.1109/TC.2025.3547136

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[40] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 2082–2090.

[41] 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)*. 570–583. doi:10.1109/HPCA51647.2021.00055

[42] Enxin Yi, Yiru Duan, Yinuo Bai, Kang Zhao, Zhou Jin, and Weifeng Liu. 2024. Cuper: Customized Dataflow and Perceptual Decoding for Sparse Matrix-Vector Multiplication on HBM-Equipped FPGAs. In *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546672

[43] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. 2024. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 223–234. doi:10.1145/3626202.3637562

[44] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264. doi:10.1109/TPDS.2019.2910068