HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs

Manoj B. Rajashekar
mba151@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Xingyu Tian
xingyu_tian@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Zhenman Fang
zhenman@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in numerous applications such as scientific computing, machine learning, and graph analytics. While recent studies have made great progress in accelerating SpMV on HBM-equipped FPGAs, there are still multiple remaining challenges to accelerate imbalanced SpMV where the distribution of non-zeros 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, and (3) new bottleneck due to the often-overlooked input vector after the SpMV acceleration.

To address those challenges, we propose 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) hybrid buffering to improve memory access for input vector, and (4) an automation framework to automatically generate the optimized HiSpMV accelerator. Experimental results demonstrate that HiSpMV achieves a geomean speedup of 15.31x (up to 61.66x) for highly imbalanced matrices, compared to state-of-the-art SpMV accelerator Serpens on the AMD-Xilinx Alveo U280 HBM-based FPGA. Compared to Intel MKL running on a 24-core Xeon Silver 4214 CPU, HiSpMV achieves a geomean speedup of 8.30x. Compared to cuSparse running on an Nvidia GTX 1080ti GPU, HiSpMV achieves a geomean of 1.93x better performance per watt. HiSpMV will be released soon at https://github.com/SFU-HiAccel/HiSpMV.

CCS CONCEPTS

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

KEYWORDS

SpMV, Workload Imbalance, Hybrid Design, Hybrid Buffering, FPGA

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0418-5/24/03…$15.00
https://doi.org/10.1145/3626202.3637557

ACM Reference Format:
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 (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637557

1 INTRODUCTION

SpMV is a fundamental mathematical operation used in various fields, including scientific computing [7, 12], circuit simulation [10], machine learning [11, 15, 16, 29, 30], and graph analytics [3, 21]. It mainly involves multiplying a sparse matrix (with a large number of zero entries) by a dense vector, resulting in a new dense vector. More specifically, the SpMV operation is described in Equation 1.

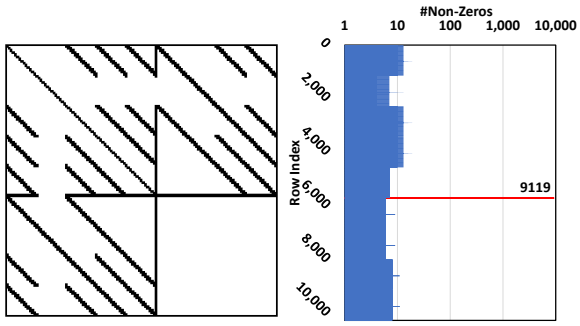
$$\vec{y} = \alpha.\mathbf{A} \times \vec{x} + \beta.\vec{y} \tag{1}$$

where $\mathbf{A}$ is a sparse matrix, $\vec{x}$ and $\vec{y}$ are dense vectors, α and β are scalar constants. Since the sparse matrix has no data reuse and has irregular distribution of non-zero 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] ← 0⃗                                  ▷ buffer temporary output y_Ax⃗
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] ← x[c : c + C_t)             ▷ load x⃗ buffer
6:             for all (r%P = p, a_rc ∈ A, a_rc ≠ 0) do ▷ stream a_rc
7:                 y_Ax[r%R_t] += a_rc * buf_x[c%C_t]  ▷ A × x⃗
8:     for (i = r; i < r + R_t; i++) do                ▷ stream & compute y_out⃗
9:         y_out[i] ← α * y_Ax[i%R_t] + β * y_in[i]
```

Recent studies [9, 20, 27] 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, non-zero 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 non-zero 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



(a) Non-Zero Structure (b) Row Distribution

Figure 1: Motivation: highly imbalanced non-zero distribution of the hangGlider_3 matrix in optimal control solver [5]

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 **A**.

However, existing approaches [9, 20, 27] no longer work effectively when accelerating *imbalanced SpMV* where the distribution of non-zeros in the sparse matrix is highly imbalanced across different rows. For instance, Figure 1 illustrates the non-zero distribution of the *hangGlider_3* matrix used in the optimal control solver [5], where one row contains nearly 1,000 times more non-zero elements compared to the average number of non-zeros in other rows. Such imbalanced matrices are commonly used in multiple application domains such as optimal control solver [5], orbit transfer [4], circuit simulation [8], and natural language processing [8], which creates multiple new challenges for efficient SpMV acceleration.

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 [9, 20, 27] (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 non-zero 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 non-zero elements from the same row.
Existing studies [9, 27] usually use re-ordering techniques to schedule the processing of non-zeros 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 non-zeros in other rows available to fill in the pipeline stalls between the non-zeros in the densest row.

3). **Input Dense Vector Buffering:** In matrices with extremely low density (e.g., 10^{-4} or lower), the off-chip memory access for the input dense vector $\vec{x}$ buffer could become a new performance bottleneck after the acceleration of SpMV, which is often overlooked in prior studies [9, 20, 27]. While ping-pong buffering looks like a straightforward solution, it necessitates additional on-chip memory resources, which could result in a lower number of PEs and potentially compromise performance for other matrices where computation is still the primary bottleneck.

In this paper, we propose HiSpMV to accelerate imbalanced SpMV 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 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 non-zeros from other rows.

3). **Hybrid Buffering for Input Vector:** We propose a hybrid buffering technique to enable the same set of on-chip buffers to be dynamically utilized by two PEs, either as private partitioned buffers or as shared ping-pong buffers. As a result, it enables the same HiSpMV accelerator to effectively hide the latency bottleneck caused by input vector loading in very low-density matrices (using shared ping-pong buffering), without affecting the performance of denser matrices where computation is still the primary bottleneck (using duplicate private buffering).

4). **Automation Tool:** To enable users to conduct fast design space exploration on a given FPGA platform, we develop an automation tool to automatically generate the optimized HiSpMV design in Vitis high-level synthesis (HLS), where users can select specific aforementioned optimizations and configure the design size (e.g., number of HBM channels) based on their needs. The automation tool is integrated with the recent scalable task-parallel HLS programming framework PASTA [22] to improve the timing closure on modern multi-die FPGAs and leverage its buffer channel feature to realize hybrid buffering.

We build HiSpMV on top of state-of-the-art open-source SpMV accelerator Serpens [27] due to its superior performance among existing studies, with the aforementioned new features to accelerate imbalanced matrices. We evaluated the performance of HiSpMV using a dataset comprising 10 balanced and 10 imbalanced matrices from the widely used SuiteSparse [8], with the balanced matrices chosen from prior SpMV studies [20, 27]. On the AMD-Xilinx Alveo U280 HBM-based FPGA, our 128 PE design, HiSpMV-16, achieves geomean speedups of 15.31x, 14.70x, and 19.26x over prior studies Serpens-16 [27], HiSparse-PB [9], and Hi-Sparse-RI [9], respectively, for imbalanced matrices. Even for balanced matrices, our HiSpMV-16 outperforms others with geomean speedups of 1.18x, 3.97x, and 4.00x compared to Serpens-16, HiSparse-PB, and HiSparse-RI. Our 160 PE design, HiSpMV-20, achieves a geomean speedup of 12.82x compared to Serpens-24 (192 PEs) [27] for imbalanced matrices,

Table 1: Comparison of HiSpMV with recent SpMV accelerators on HBM-based FPGA

Accelerator	#PEs (P)	Imbalanced Workload	RAW Dependency		Input Vector Buffering		GFLOPS	
			Distance Reduction	Resolution Technique	PE Access	Latency Hiding	Balanced (mouse_gene)	Imbalanced (trans5)
Serpens-16 [27]	128	X	No Optimization (Distance: 10)	Re-Ordering	Private	X	42.26	0.31
Serpens-24 [27]	192	X	No Optimization (Distance: 10)	Re-Ordering	Private	X	57.96	0.31
HiSparse-PB [9]	128	X	IFWQ (Distance: 7)	Additional Buffers	Dynamic Sharing	X	25	0.82
HiSparse-RI [9]	128	X	IFWQ (Distance: 7)	Re-Ordering	Dynamic Sharing	X	13.1	0.18
AMD design [20]	128	X	N / A	Dynamic Stall	Dynamic Sharing	X	38	N / A
HiSpMV-16 (ours)	128	**Hybrid Row Distribution**	**Circular Buffer (Distance: 5)**	**Adder Chain**	**Hybrid Buffering**	**Hybrid Buffering**	43.11	**18.76**
HiSpMV-20 (ours)	160	**Hybrid Row Distribution**	**Circular Buffer (Distance: 5)**	Re-Ordering	**Hybrid Buffering**	**Hybrid Buffering**	50.56	**16.73**

and geomean speedups of 1.31x and 1.10x compared to the recent AMD design [20] and Serpens-24 [27] for balanced matrices.

Moreover, HiSpMV-16 achieves geomean speedups of 4.27x and 8.30x over Intel MKL running on a 24-core Xeon Silver 4214 CPU, for balanced and imbalanced matrices, respectively. It also achieves a geomean of 2.21x and 1.93x better performance per watt (GFLOPS/watt) compared to Nvidia cuSparse running on the GTX 1080ti GPU, for balanced and imbalanced matrices, respectively.

2 MOTIVATION AND HIGH-LEVEL IDEAS

In this section, we present an in-depth analysis of new challenges in accelerating imbalanced SpMV and our high-level ideas to address those challenges. Table 1 summarizes novel contributions of our HiSpMV design with a comparison to three state-of-the-art (balanced) SpMV accelerator designs on HBM-based FPGAs, including Serpens [27], HiSparse [9], and the latest AMD design [20].

2.1 Imbalanced Workload

2.1.1 Analysis of the Problem: First of all, we analyze how much impact the workload imbalance can have on the performance of existing SpMV accelerators [9, 20, 27] that cyclically assign rows onto P number of PEs. Assume the input matrix has R rows and C columns, NNZ is the number of non-zeroes, and the density ρ 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** δ 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 non-zeros could be allocated to one PE. But since a PE only gets R/P rows, it can only have a maximum of $R * C/P$ non-zeroes. 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 ρ becomes lower, the imbalance ratio's upper bound becomes higher; 2) having more PEs also increases the imbalance ratio upper bound. We have profiled a set of sparse matrices from the widely used SuiteSparse [8] with a different imbalance ratio δ (Table 2 in Section 4.1). For imbalanced matrices with $\delta \geq 2$, state-of-the-art SpMV accelerator Serpens-16 [27] only achieves a geomean of 1.22 GFLOPS, which is about 27.2x lower than that for balanced matrices.

2.1.2 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 3.3 and 3.4.

2.2 Long-Distance RAW Dependency

2.2.1 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 rth 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, 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 [27] and HiSparse-RI [9] adopt an out-of-order scheduling, shown in Figure 2(b), to schedule non-zeros 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 non-zeros available from other rows to fill this pipeline gap. On the other hand, HiSparse-PB [9] introduces additional y_Ax buffer copies to resolve this issue, which incurs more on-chip buffer overhead. The latest AMD design [20] dynamically stalls the pipeline using a hazard resolution back-pressure unit, which does not effectively avoid the stalls.

Another alternative is to reduce this 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 [9] uses an *in-flight-wait-queue (IFWQ)* to store results in local registers and reduces dd from

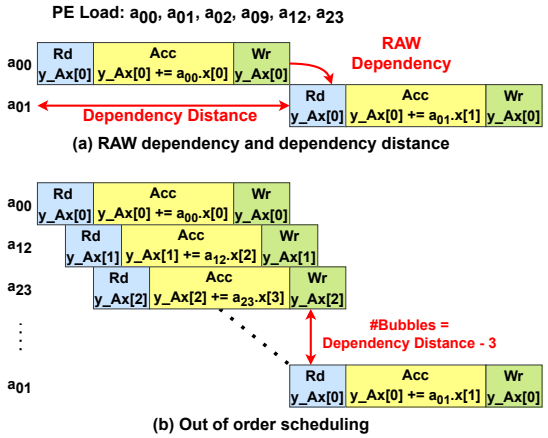


Figure 2: RAW dependency for floating-point accumulation on $y_Ax[r\%R_t]$ and out-of-order scheduling to avoid stalls

10 (in Serpens [27]) 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.

2.2.2 Proposed Solution: In HiSpMV, 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 [9], 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 3.5.

2.3 New Bottlenecks on Dense Vectors

2.3.1 Analysis of the Problem: After streaming the sparse matrix A and optimizing the compute 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 [9, 20, 27].

For the first bottleneck, ping-pong buffering looks like a straightforward solution to hide the off-chip access latency of loading the input vector $\vec{x}$. However, it doubles on-chip memory resource usage for buffering $\vec{x}$, which could result in a lower number of PEs and thus potentially compromise performance for other matrices where computation is still the primary bottleneck. For the second bottleneck, it could be easily addressed by using more HBM channels for streaming $\overrightarrow{y_in}$ and $\overrightarrow{y_out}$, and duplicating the PEs to compute $\overrightarrow{y_out}$, which will be detailed in Section 3.1.

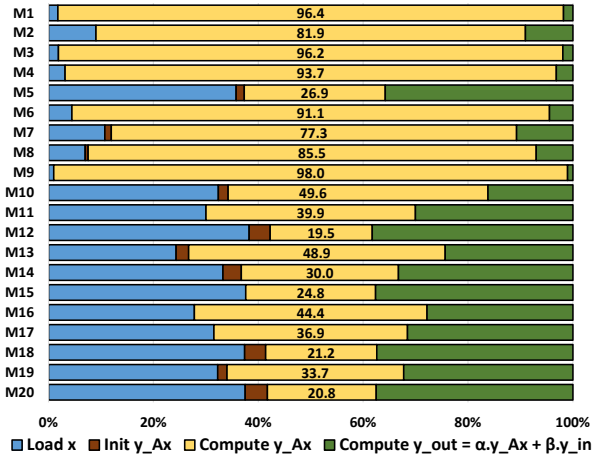


Figure 3: Execution cycle breakdown for each component in the well-optimized SpMV accelerator

2.3.2 Proposed Solution: To address the first bottleneck of loading the input dense vector $\vec{x}$, we propose a **hybrid buffering** technique to enable the same set of on-chip buffers to be dynamically utilized by two PEs in two modes. In the **private partitioned buffering mode**, loading the two input vector buffers and computing $\mathbf{A} \times \vec{x}$ happen sequentially; but each PE has access to its own private buffer copy. In the **ping-pong buffering mode**, while it loads one input vector buffer, it computes on the other input vector buffer in parallel; but two PEs have to share one input vector buffer during the computation. As a result, the same HiSpMV accelerator can effectively work for all matrices whether the loading of the input vector becomes a bottleneck or not.

It is non-trivial to implement this hybrid buffering in the original Vitis HLS, which would require extensive code transformations. In this paper, we implement the hybrid buffering by leveraging the novel buffer channel feature proposed in the recent scalable task-parallel HLS programming framework PASTA [22]. Its detailed design and implementation will be presented in Section 3.2.

3 HISPMV DESIGN AND IMPLEMENTATION

3.1 HiSpMV Architecture Overview

We build HiSpMV based on state-of-the-art open-source SpMV accelerator Serpens [27], with the support of novel features summarized in Section 2 and an automation tool support in Section 3.6. Figure 4 presents an overview of the 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 and implement our hybrid buffering technique. We relocate the accumulation and output buffer from the PEG into their dedicated *y_Ax handler* module. Additionally, we introduce new modules, including the *adder chain groups (ACG)* and a *hybrid row distribution network* between the PEGs and *y_Ax* handlers.

We use a single HBM channel 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 channels for streaming sparse matrix A, with each channel serving 4 PEGs where each PEG includes 2 PEs. Furthermore, we have M channels designated for streaming in $\overrightarrow{y_{in}}$ and

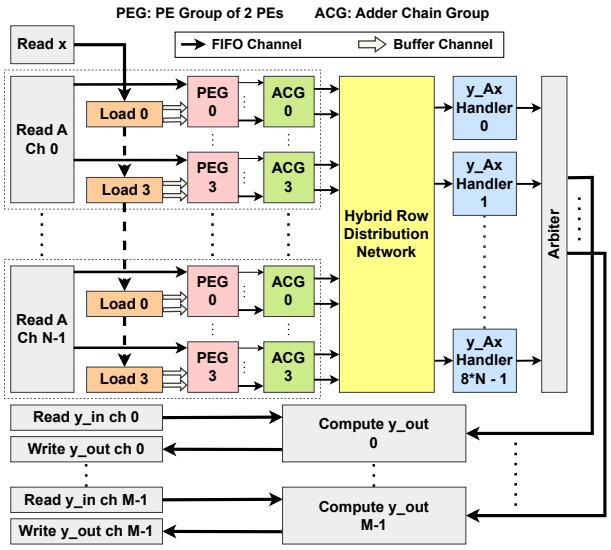


Figure 4: HiSpMV architecture overview

streaming out $\overrightarrow{y_{out}}$ to improve its performance. All these channels are 512 bits wide, a choice made due to its optimal performance [25].

The PEs perform the multiplication of non-zero 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, 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 $\overrightarrow{y_{out}}$.

3.2 Hybrid Buffering for Input Vector

3.2.1 Hybrid Buffering Using PASTA Buffer Channels: The recent scalable task-parallel HLS programming framework PASTA [22] introduced support for buffer channels based on top of prior TAPA and Autobridge framework [13, 14]. We leverage this task-parallel programming and buffer channel support to implement our novel hybrid buffering technique in HiSpMV, which would require extensive coding efforts to implement using the original Vitis HLS. Specifically, PASTA provides an explicit **acquire** API and an implicit **release** API to easily control the buffer channel access.

Algorithm 2 shows the pseudo-code to implement our hybrid buffering. We separate loading x and PEG computation into individual PASTA tasks, which communicate with each other via an array of two buffer channels ($buf_ch[2]$ declared in line 2).

1). In ping-pong buffering mode, while the load task acquires one buffer channel and writes it (lines 5-8), the PEG task can acquire the other buffer channel and reads it (lines 17-20). This allows the two tasks to run in parallel and two PEs to share one buffer channel during the computation.

2). In private partitioned buffering mode, the load task first acquires both buffer channels and writes them (lines 10-13). Only after the load task releases both buffer channels, the PEG task can acquire them and read them (lines 22-25). This allows each PE to access its own private buffer channel, but the load and PEG tasks run sequentially.

Algorithm 2 Pseudo-code of load and PEG w/ hybrid buffering

```
1:  tapa::buffer buf_ch[2]                          ▷ PASTA supports acquire & release
2:  load task: i ← 0
3:  for (r = 0; r < R; r = r + R_t) do
4:      for (c = 0; c < C; c = c + C_t) do
5:          if PING_PONG then
6:              buf_x ← acquire(buf_ch[i%2])
7:              write to buf_x[C_t]
8:              release(buf_ch[i++%2])
9:          else
10:             buf_x0 ← acquire(buf_ch[0])
11:             buf_x1 ← acquire(buf_ch[1])
12:             write to buf_x0[C_t], buf_x1[C_t]
13:             release(buf_ch[0], buf_ch[1])
14: PEG task: i ← 0
15: for (r = 0; r < R; r = r + R_t) do
16:     for (c = 0; c < C; c = c + C_t) do
17:         if PING_PONG then                       ▷ Both PEs share a channel
18:             buf_x ← acquire(buf_ch[i%2])
19:             PE_0, PE_1 read buf_x[C_t]
20:             release(buf_ch[i++%2])
21:         else                                    ▷ Each PE has a private channel
22:             buf_x0 ← acquire(buf_ch[0])
23:             buf_x1 ← acquire(buf_ch[1])
24:             PE_0 read buf_x0[C_t], PE_1 read buf_x1[C_t]
25:             release(buf_ch[0], buf_ch[1])
```

The buffering mode can be dynamically switched by configuring the $PING_PONG$ flag (lines 5 and 17) based on the bottleneck.

3.2.2 When to Use Each Mode: When two PEs share one buffer channel in the ping-pong buffering mode, it can no longer guarantee that they will operate without stalls. While we can dynamically stall/pause the PE pipeline (still with II=1), it impacts the compute time. Our buffer channel width is set to 512 bits, meaning each cycle it can access 16 floating-point numbers packed together. Therefore, if the accessed values by two PEs are in the same pack, they can work in parallel; otherwise, a stall is required between them. Let t_c denote the compute time without stalls (PEG task with private buffer), t_b denote the buffer time (load task), and t'_c denote the compute time in the worst case where a stall occurs in every cycle, doubling the compute time, i.e., $t'_c = 2.t_c$.

Let t_s and t_p represent the total time for buffering and computing in private partitioning and ping-pong buffering modes, then:

$$t_s = t_c + t_b; \quad t_p = Max\left(t'_c,\ t_b\right) = Max\left(2.t_c,\ t_b\right) \tag{6}$$

We can decide when to ping-pong buffering with equation 7:

$$\text{Buffer method} = \begin{cases} \text{ping-pong buffering} & \text{if } t_p \le t_s \\ \text{private buffering} & \text{otherwise} \end{cases} \tag{7}$$

By substituting equation 6 in equation 7 and simplifying it, we get

$$\text{Buffer method} = \begin{cases} \text{ping-pong buffering} & \text{if } t_c \le t_b \\ \text{private buffering} & \text{otherwise} \end{cases} \tag{8}$$

Note when $t_c > t_b$, ping-pong buffering may still get better performance as two PEs may access values in the same 512-bit pack in one cycle. However, such behavior is dynamic and it is infeasible

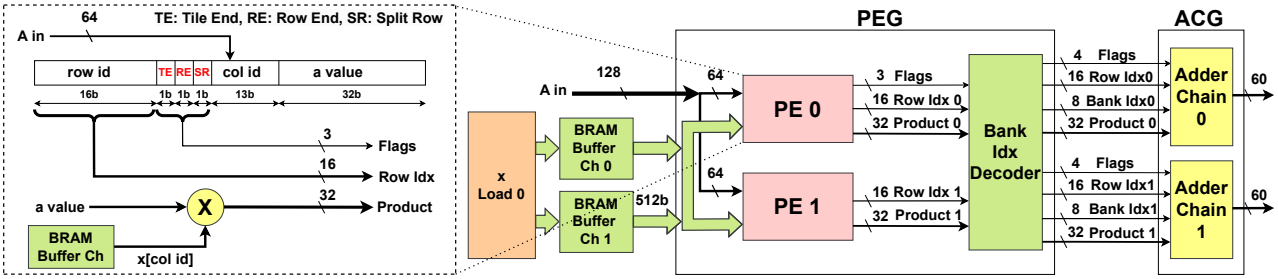


Figure 5: Architecture of the processing element (PE) and PE group (PEG) design in HiSpMV

to derive the actual compute time. Therefore, we conservatively assume the worst case and choose private buffering when $t_c > t_b$.

3.3 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 5, each PEG reads in 128-bit of **A**, and each PE inside the PEG reads in 64-bit of **A**. This 64-bit data includes the 16-bit row id r, a 13-bit column id c, and a 32-bit non-zero value a_{rc}, as well as three flag bits (Tile End, Row End, and Split Row). Using the column id c, each PE fetches the corresponding (32-bit) x_c value from the $\vec{x}$ buffer ($buf_x[C_t]$) 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 for two reasons: 1) it achieves the best off-chip bandwidth during the load from the HBM channel [25]; 2) in the ping-pong buffering mode, it reduces the actual PE pipeline stalls since two PEs may share two values in the same 512-bit pack.

3.3.1 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. Basically, 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 non-zeros 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 5 works as follows:

```
if (SR):  // Split Row: intra-row distribution
    bank_idx0, bank_idx1 = row_idx0, row_idx0
    row_idx0, row_idx1 = row_idx1, row_idx1
else:                // inter-row distribution
    bank_idx0, bank_idx1 = PE_idx0, PE_idx1
    row_idx0, row_idx1 = row_idx0, row_idx1
```

Besides the 3 flags received from the PE, the bank index decoder also sends an additional flag called "Last," indicating termination for other tasks. The specific functionality of these flags is not crucial to the novelty of the work and is omitted due to space constraints.

3.4 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 build this network based on small blocks with 2 inputs and 2 outputs. Figure 6a 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 6a.

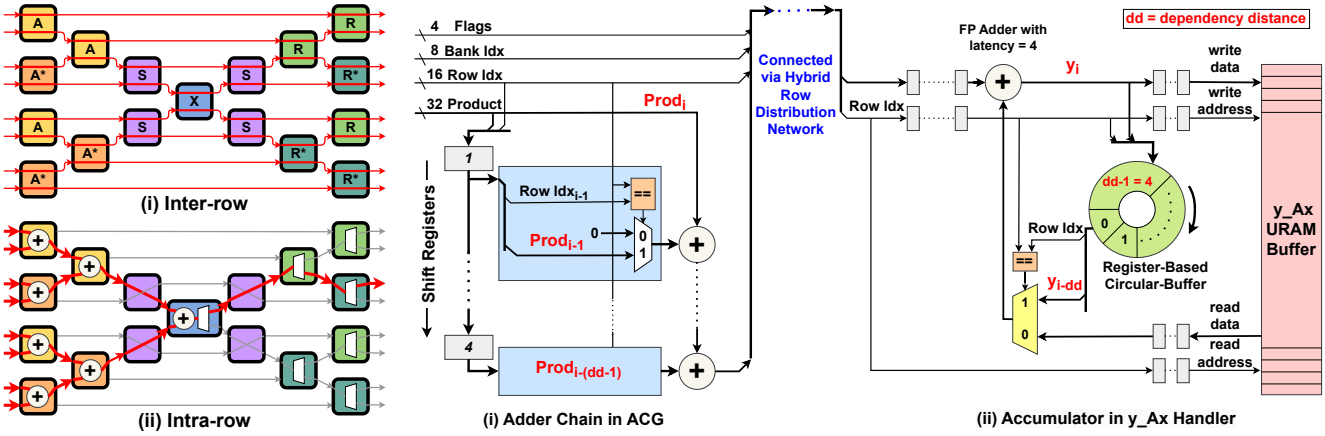
Shown in Figure 6a, the network is constructed using the following small blocks. **Adder blocks 'A' and 'A*'** add the results from two inputs and direct the sum to one output 1 and 0, respectively. The other output will just be a dummy packet. **Switch blocks 'S'** switch the outputs, forwarding results 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).

3.4.1 When to Use Each Distribution Mode: In the preprocessing, we decide in which mode (inter-row or intra-row distribution) each row should be processed to achieve a good balance. An initial imbalance ratio δ is first calculated assuming all rows are assigned in the `inter-row` mode. Then we iteratively refine δ until the decrease on δ between two consecutive iterations becomes marginal, i.e., less than a predefined threshold Δ ($\Delta = 0.01$ in our design). Within each iteration, the row with the highest number of non-zeros in the `inter-row` mode is identified. A new δ' is then recalculated for all remaining rows in the `inter-row` mode without considering the identified row. If $\delta - \delta' < \Delta$, stop the process. Otherwise, the identified row is moved from `inter-row` mode to `intra-row` mode. Then it updates $\delta = \delta'$ and continues to the next iteration.

3.5 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.

3.5.1 Circular Buffer: Accumulating directly on buffers like URAM or BRAM introduces read/write latencies. Therefore, as shown in Figure 6b.(ii), we utilize a local register-based circular buffer to store temporary accumulation values to avoid such latencies. As presented in Section 2.2, one common approach is to schedule $dd - 1$ (dd: dependency distance) non-zeros 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



(a) 8x8 hybrid row distribution network (b) Fully pipelined floating-point accumulation with adder chain and circular buffer

Figure 6: (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.

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 with the new product; otherwise, it starts the accumulation for a new row.

Vitis HLS synthesizes 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 $dd = 5$.

3.5.2 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 ith 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 6b.(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.

3.6 Automation Tool

Shown in Figure 7, we develop a user-friendly automation tool to generate optimized HiSpMV accelerators on HBM-based FPGAs.

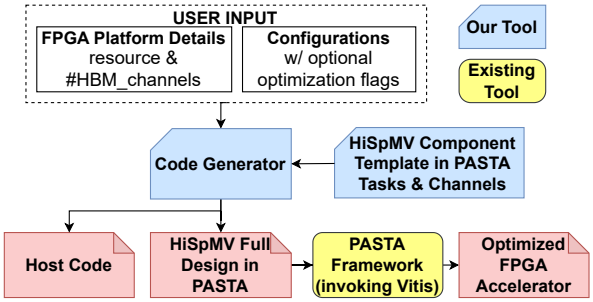


Figure 7: HiSpMV automation flow, integrated with the recent PASTA [22] programming framework

Given FPGA platform information, including available resources, the number of HBM channels, and user configurations, our code generator can generate HiSpMV designs for any number of sparse matrix (A) and output vector ($\overrightarrow{y_in}$ and $\overrightarrow{y_out}$) channels (denoted as M and N, respectively). Additionally, users have the flexibility to opt for designs with or without each of our presented optimizations to perform design space exploration and gain architecture insights. These parameters will be decided by our code generator based on available resources, if not specified by users.

Utilizing the PASTA [22] task-parallel HLS programming model, including APIs for defining tasks and buffer and FIFO channels, we build each type of the HiSpMV components as a template. When scaling up or down, only the quantity of these components and their connections change. With profiled resource utilization metrics of each component, our code generator can make a highly reliable estimation for the full design. Users can also customize the configuration based on the estimation. Finally, the optimized HiSpMV accelerator is built with the PASTA framework, which performs coarse-grained floorplanning and pipelining optimizations before invoking Vitis to generate the bitstream.

Building on top of the PASTA programming framework makes it easier to 1) realize hybrid buffering using its buffer channel feature, 2) scale the design based on the task-parallel programming model, and 3) improve the timing closure on modern multi-die FPGAs.

Table 2: Benchmark matrices and their properties (size, number of non-zeros, density, and imbalance ratio), speedup breakdown of HiSpMV-16 over Serpens-16, and HiSpMV-16 performance and bandwidth

Matrix Properties						Speedup Breakdown (based-on Cycle Counts)					Performance	Bandwidth
ID	Matrix Name	Size (Rows=Cols)	NNZ	Density (ρ)	Imb. Ratio (δ)	Row Distri.	Reduced Dep. Dist.	Adder Chain	Comp. y_out	Hybrid Buffer	GFLOPS	GB/S
Balanced Matrices					**Geomean →**	**1.00**	**1.15**	**1.00**	**1.05**	**1.01**	**39.14**	**176.90**
M1	TSOPF_RS_b2383	38,120	16,171,169	1.11E-02	1.01	1.00	1.13	1.00	1.01	1.00	50.75	207.09
M2	crystk03	24,696	1,751,178	2.87E-03	1.01	1.00	1.16	1.00	1.05	1.00	45.51	186.95
M3	nd6k	18,000	6,897,316	2.13E-02	1.05	1.00	1.11	1.00	1.01	1.00	49.05	207.10
M4	crankseg_2	63,838	14,148,858	3.47E-03	1.07	1.00	1.52	1.00	1.02	1.00	47.45	203.48
M5	ford2	100,196	544,688	5.43E-05	1.08	1.00	1.05	1.00	1.22	1.12	21.54	100.52
M6	thread	29,736	4,444,880	5.03E-03	1.09	1.00	1.18	1.00	1.02	1.00	45.69	200.06
M7	PFlow_742	742,793	37,138,461	6.73E-05	1.14	1.00	1.00	1.00	1.06	1.00	39.06	180.89
M8	Si41Ge41H72	185,639	15,011,265	4.36E-04	1.21	1.00	1.08	1.00	1.04	1.00	39.66	192.40
M9	mouse_gene	45,101	28,967,291	1.42E-02	1.21	1.00	1.01	1.00	1.01	1.00	43.17	209.32
M10	soc-Pokec	1,632,803	30,622,564	1.15E-05	1.22	1.00	1.38	1.00	1.09	1.00	23.86	125.54
Imbalanced Matrices					**Geomean →**	**9.15**	**1.33**	**1.18**	**1.20**	**1.10**	**18.72**	**104.60**
M11	c-52	23,948	202,708	3.53E-04	2.28	3.51	1.42	1.03	1.18	1.00	22.27	114.22
M12	language	399,130	1,216,334	7.64E-06	2.29	2.26	1.17	1.03	1.25	1.32	16.17	88.99
M13	analytics	303,813	2,006,126	2.17E-05	3.05	1.31	2.72	2.12	1.14	1.00	14.29	132.17
M14	nxp1	414,604	2,655,880	1.55E-05	4.39	9.81	1.03	1.00	1.21	1.04	22.05	103.68
M15	poli_large	15,575	33,033	1.36E-04	4.40	1.97	1.48	1.45	1.23	1.19	12.38	107.41
M16	lowThrust_7	17,378	211,561	7.01E-04	5.05	17.52	1.02	1.01	1.16	1.00	27.73	120.16
M17	hangGlider_3	10,260	92,703	8.81E-04	13.47	44.17	1.02	1.01	1.19	1.00	24.63	106.72
M18	boyd2	466,316	1,500,397	6.90E-06	18.40	16.63	1.42	1.29	1.24	1.27	15.51	88.78
M19	trans5	116,835	749,800	5.49E-05	20.30	34.08	1.33	1.14	1.20	1.00	19.17	101.87
M20	ASIC_680k	682,862	2,638,997	5.66E-06	32.82	46.74	1.23	1.07	1.24	1.28	18.54	90.35

4 EVALUATION

4.1 Experimental Setup

We extensively compare the performance of our design with state-of-the-art open-source SpMV FPGA designs such as Serpens [27] and HiSparse [9], as well as optimized Intel MKL library on CPU [18] and Nvidia cuSparse library on GPU [26]. We evaluate them using a diverse set of 20 matrices from the widely used SuiteSparse [8], including 10 balanced and 10 imbalanced matrices. Our selection of balanced matrices aligns with those used in Serpens [27]. For imbalanced matrices, we deliberately choose a variety of matrices with varying imbalance ratios and densities from different applications. Details about these matrices and their properties are presented in Table 2. Our code generator has been rigorously tested on AMD-Xilinx Alveo U280 and U50 platforms using Vitis versions 2021.2 and 2022.2 to get the best configurations. For the FPGA results reported in this paper, we mainly use the Alveo U280 FPGA that is used in prior studies [9, 20, 27] unless otherwise specified. We leave the detailed CPU and GPU configuration in Section 4.5.

4.2 Performance Breakdown of Optimizations

A detailed breakdown of the speedup achieved by each optimization for our 128-PE design (HiSpMV-16), with Serpens-16 [27] as the baseline, is provided in Table 2. First, we apply the hybrid row distribution network to address imbalances, yielding a geometric mean speedup of 9.15x (up to 46.74x) for imbalanced matrices, with no effect on balanced matrices. Second, we reduce the dependency distance to 5 and utilize a more efficient re-ordering algorithm, resulting in a geometric mean speedup of 1.15x (up to 1.38x) for balanced matrices and a geometric mean of 1.33x (up to 2.72x) for imbalanced matrices. Third, the introduction of the adder chain to remove re-ordering contributes to a geometric mean speedup of 1.18x (up to 2.12x). After these compute optimizations, we focus on optimizing dense vector access. Initially, we deploy two compute

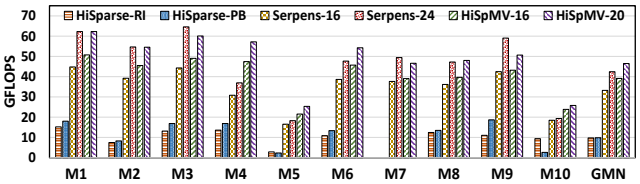


Figure 8: FPGA kernels on balanced matrices

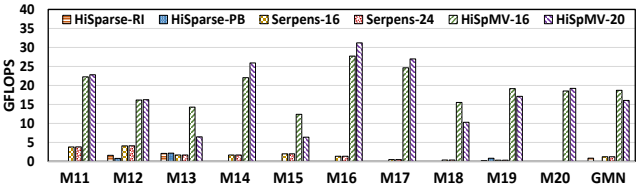


Figure 9: FPGA kernels on imbalanced matrices

$\overrightarrow{y_out}$ modules, providing a geometric mean speedup of 1.05x (up to 1.22x) for balanced matrices and a geometric mean of 1.2x (up to 1.25x) for imbalanced matrices. Hybrid buffering, introduced subsequently, yields a geometric mean speedup of 1.1x (up to 1.32x) for imbalanced matrices and 1.01x (up to 1.12x) for balanced matrices. These optimizations are cumulative, and the total speedup is the product of all individual speedups. The overall geometric mean speedup estimated from clock cycle counts for balanced and imbalanced matrices is 1.22x and 15.84x (excluding M20) respectively. The actual run's speedup closely aligns with this cycle analysis.

4.3 Overall Performance vs. Prior FPGA Studies

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 on the Alveo U280 FPGA board, running the matrices with kernels HiSparse-RI, HiSparse-PB, Serpens-16, Serpens-24, HiSpMV-16, and HiSpMV-20 for 10,000 iterations to ensure consistent execution times across all matrices.

The performance comparisons are illustrated in Figure 8 for balanced matrices and Figure 9 for imbalanced matrices. For balanced matrices, HiSpMV-16 achieves a geomean speedup of 1.18x (up to 1.54x), 3.97x (up to 9.19x), and 4.00x (up to 7.52x) compared to Serpens-16, HiSparse-PB, and HiSparse-RI, respectively. Additionally, HiSpMV-20 achieves a geomean speedup of 1.1x (up to 1.55x) over Serpens-24. For imbalanced matrices, HiSpMV-16 outperforms Serpens-16, HiSparse-PB, and HiSparse-RI with a geomean speedup of 15.31x (up to 61.66x), 14.7x (up to 23.38x), and 7.52x (up to 104.54x), respectively. HiSpMV-20 achieves a geomean speedup of 12.82x (up to 58.08x) over Serpens-24.

Notably, prior works failed to run certain imbalanced matrices due to the exceeding of the HBM memory capacity limit. The reason is that their out-of-order scheduling sometimes added too many bubbles (dummy data) which increases the memory footprint. Our hybrid row distribution and adder chain can effectively avoid such out-of-order scheduling and its overhead.

4.4 Resource Utilization and Design Frequency

Table 3 presents the resource utilization and kernel frequency of HiSpMV-16 and HiSpMV-20, featuring 16 and 20 channels for streaming the sparse matrix. Both designs incorporate 2 compute $\overrightarrow{y_out}$ modules. Notably, HiSpMV-20 circumvents the dependency distance challenge by employing re-ordering techniques, as it does not possess adder chains due to resource constraints.

Table 3: HiSpMV resource utilization and frequency

Design Name	Resource Utilization					Freq.
	LUT	FF	BRAM	URAM	DSP	(MHz)
HiSpMV-16 (U280)	56.8%	30.2%	60.9%	40.0%	25.7%	221
HiSpMV-20 (U280)	57.2%	28.1%	73.7%	33.3%	17.2%	225
HiSpMV-56 est (VH1782)	40.5%	17.5%	57.9%	23.27%	6.51%	400

To fully leverage the maximum bandwidth of the HBM at 450MHz with 512-bit ports, our kernels only need to achieve a frequency of 225MHz [25]. Both designs are on track with this goal: HiSpMV-16 achieved 221MHz, and HiSpMV-20 reached 225MHz. The primary resource constraints in our current designs are LUTs and BRAMs.

4.5 Comparison to CPU and GPU

We conducted benchmarking on both CPU and GPU platforms using the Intel MKL library [18] on a Xeon Silver 4214 CPU with 24 cores and the Nvidia cuSparse library [26] 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) VH1782 FPGA [2] with 819.2GB/s bandwidth against the 7nm A100 GPU with 1,555GB/s bandwidth. Besides the VH1782 FPGA, all performance and power results are measured on the actual boards. The power consumption is measured using vendor tools (*nvml* for GPUs and *xbutil* for FPGAs) during the kernel runtime.

4.5.1 Projection for VH1782 FPGA Results. Compared to U280, VH1782 has significant improvements. First, it improves the HBM bandwidth from 460GB/s to 819.2GB/s and the HBM frequency from 450MHz to 3.2GHz. It has 16 HBM channels, each with a bus width of 128-bit. Second, it has a hardened network-on-chip (NoC) running at 1GHz which connects all HBM channels and up to 76 AXI ports. 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. With such a 256-bit AXI port (instead of 512-bit in U280), the number of PEs required to saturate each AXI port's bandwidth is halved to 4.

Third, it has hardened floating-point (FP) units: one DSP58 [1] unit can process one FP multiply, add, or fused multiply-add in one cycle in a pipelined fashion, running at higher than 500MHz. Each PE uses roughly 1/3 of DSPs compared to that (DSP48) in U280; and VH1782 has 20.2% more DSPs than U280. Moreover, the latency for FP add is only one cycle, which eliminates long-distance RAW dependency and resources for the adder chain. Lastly, VH1782 has 1.86x more BRAM banks than U280 for buffering dense vectors.

Based on the above analysis, we project a HiSpMV-56 design running at 400MHz on the VH1782 FPGA: 56 AXI ports for streaming in matrix **A**, 4 AXI ports for loading $\vec{x}$ and streaming in $\overrightarrow{y_in}$ (time-multiplexing at different stages), and 4 AXI ports for streaming out $\overrightarrow{y_out}$ (under-utilized). The total number of PEs is 56 ports * 4 PEs/port = 224 PEs; in contrast, HiSpMV-16 on U280 has a total of 16 * 8 = 128 PEs. Its resource utilization and frequency estimation are summarized in Table 3. The design is bottlenecked by the off-chip bandwidth. If there was 1.5x more bandwidth (1,228.8GB/s), the design could be further scaled up by 1.5x to HiSpMV-84, where BRAM would become the new bottleneck (86.9% BRAM utilization).

For this projected HiSpMV-56 design on VH1782, we can accurately estimate its performance for an input matrix using an analytical model, as it is a fully pipelined design with II=1. For the power consumption, we assume the static power is the same as that of U280, and estimate its dynamic power via a linear model based on multiple U280 design points: we assume the dynamic power scales linearly with the frequency and the numbers of PEs.

4.5.2 Comparison Results. Figure 10 compares the performance (GFLOPS) between Intel MKL, Nividia cuSparse on 1080ti (GPU 1) and A100 (GPU 2), HiSpMV-16 on U280 (FPGA 1) and estimation of HiSpMV-56 on VH1782 (FPGA 2). Compared to the CPU, HiSpMV-16 achieves a geomean speedup of 4.26x (up to 7.67x) for balanced matrices and 8.3x (up to 47.62x) for imbalanced matrices.

1080ti GPU vs. U280 FPGA. For performance, on average (geomean), 1080ti GPU is 1.6x better for balanced matrices and 2.1x better for imbalanced matrices, compared to U280 FPGA. On the other hand, Figure 11 compares their energy efficiency in terms of GFLOPS/Watt: U280 achieves better energy efficiency over 1080ti across all matrices; the geomean energy efficiency improvement for balanced and imbalanced matrices is 2.21x and 1.9x.

A100 GPU vs. VH1782 FPGA. For performance, on average, A100 GPU is 1.6x better for balanced matrices and 1.1x better for imbalanced matrices, compared to VH1782 FPGA. If VH1782 FPGA had 1.5x more bandwidth (1,228.8GB/s), this gap would become marginally 1.2x and 0.8x. Even with the current VH1782, for certain imbalanced matrices M11, M15, M16, and M17, VH1782 has better performance than A100. The reason is that these matrices have a

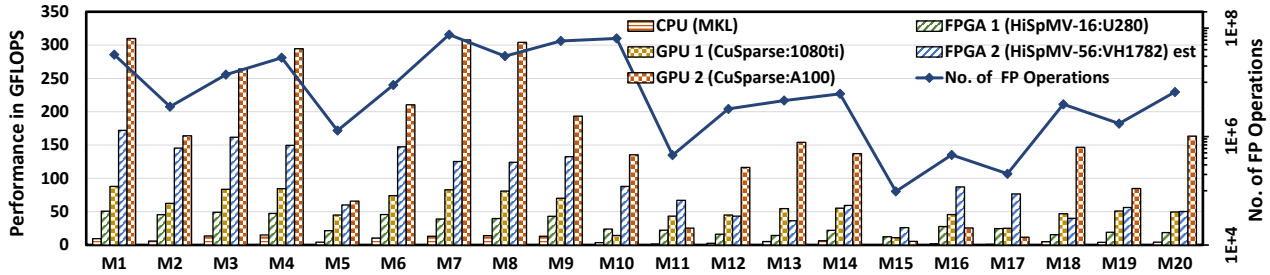


Figure 10: CPU, FPGA, and GPU performance comparison

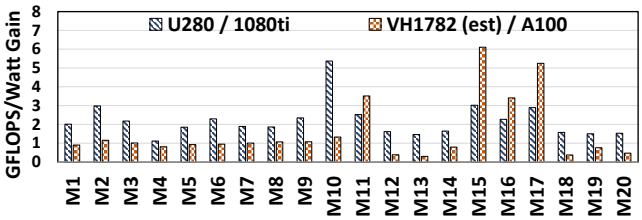


Figure 11: Energy efficiency improvement of FPGA over GPU

much lower number of floating-point (FP) operations (shown in the line of Figure 10); thus, A100 has lower utilization of GPU cores and larger performance overhead. This also leads to that A100 performs worse than the older 1080ti GPU (with fewer cores) for these matrices. On average, A100 GPU has improved the performance over 1080ti GPU by 3.3x and 1.4x for balanced and imbalanced matrices. Meanwhile, VH1782 FPGA has improved its performance over U280 FPGA by 3.2x and 2.7x for balanced and imbalanced matrices.

On the other hand, Figure 11 compares their energy efficiency. For balanced matrices, VH1782 and A100 achieve similar energy efficiency. For imbalanced matrices that have a much lower number of FP operations, i.e., M11, M15, M16, and M17, VH1782 achieves better energy efficiency (4.4x on average) as the GPU cores are underutilized and achieve lower performance. For other imbalanced matrices, A100 achieves better energy efficiency (2.1x on average).

5 RELATED WORK

SpMV Accelerator Design: For the three latest SpMV accelerators on HBM-based FPGAs, including Serpens [27], HiSparse [9], and the latest AMD design [20], we have compared them with in-depth analysis and quantitative comparison to showcase our advantage in Section 2 and 4.

Prior to Serpens, GraphLily [17] aims for graph applications presented in SpMV and designs a general SpMV accelerator with an overlay design, but it only reaches 165MHz with limited performance. HitGraph [32] proposes a graph processing acceleration framework that can perform SpMV. ThunderGP [6] 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. [19] present an accelerator design based on the Xilinx GEMX SpMV engine but the performance is much lower. Liu et al. [24] 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 RAW dependency for accumulation and can hardly scale to larger FPGA. Li et al. [23] propose a novel compressed format tackling inefficient memory access, but how such an approach scales up is not addressed. Other SpMV-related accelerator designs, like SpaceA [31] and GraphR [28], are only evaluated in simulation.

Timing Optimization for HLS Designs: HBM and multiple-dies have been adopted into modern datacenter FPGAs such as AMD-Xilinx Alveo U280. Interconnection crossing dies with long delay introduces more challenges to improve the quality of placement and routing. To tackle this problem, we utilize the recent PASTA [22] framework, which is built on top of TAPA/Autobridge [13, 14]. 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.

6 CONCLUSION

In this work, we have conducted an in-depth analysis of the new challenges to accelerate imbalanced SpMV on HBM-based FPGAs and proposed the HiSpMV design to address those challenges. 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—register-based circular buffer and adder chain—to achieve fully pipelined floating-point accumulation. Third, we realized a hybrid buffering technique to dynamically switch between private partitioned buffering and shared ping-pong buffering to optimize the buffer loading. In addition, we developed an automation framework that automatically generates the optimized HiSpMV accelerator design based on the user configurations. Extensive experimental results demonstrated the performance advantage of our design over state-of-the-art SpMV accelerators on FPGA (such as Serpens, HiSparse, and the latest AMD design) and Intel MKL on CPU, as well as energy efficiency gains over the Nvidia cuSparse on GPU.

ACKNOWLEDGEMENTS

We thank anonymous reviewers and our shepherd Dr. Gabriel Weisz for their insightful comments to help improve our paper. We acknowledge the partial support from NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund and BC Knowledge Development Fund; Huawei Canada, AMD-Xilinx; 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. 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.

[3] 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.

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

[5] 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. https://doi.org/10.1007/978-3-0348-7539-4_20

[6] 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. https://doi.org/10.1145/3431920.3439290

[7] 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. https://doi.org/10.1039/C8SC04228D

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[9] 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. https://doi.org/10.1145/3490422.3502368

[10] 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. https://doi.org/10.1145/62038.62043

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

[12] 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. https://doi.org/10.1109/FCCM.2015.30

[13] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-Parallel Dataflow Programming Framework for Modern FPGAs with Co-Optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages. https://doi.org/10.1145/3609335

[14] 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. https://doi.org/10.1145/3431920.3439289

[15] 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.

[16] 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.

[17] 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. https://doi.org/10.1109/ICCAD51958.2021.9643582

[18] 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.

[19] 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. https://doi.org/10.1109/FPL50879.2020.00031

[20] 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. https://doi.org/10.1109/FCCM57271.2023.00023

[21] 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. https://doi.org/10.1109/HPEC.2016.7761646

[22] 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. https://doi.org/10.1109/FCCM57271.2023.00011

[23] 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. https://doi.org/10.1109/ICCAD51958.2021.9643453

[24] 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. https://doi.org/10.1145/3566097.3567839

[25] 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. https://doi.org/10.1145/3431920.3439284

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

[27] 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. https://doi.org/10.1145/3489517.3530420

[28] 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. https://doi.org/10.1109/HPCA.2018.00052

[29] 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.

[30] 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.

[31] 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. https://doi.org/10.1109/HPCA51647.2021.00055

[32] 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. https://doi.org/10.1109/TPDS.2019.2910068