Chasoň: Supporting Cross HBM Channel Data Migration to Enable Efficient Sparse Algebraic Acceleration

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

Abstract

High bandwidth memory (HBM) equipped sparse accelerators are emerging as a new class of accelerators that offer concurrent accesses to data and parallel execution to mitigate the memory bound behavior of sparse kernels. However, because of their underlying non-zero scheduling scheme, state-of-the-art HBM-based sparse accelerators (e.g. Serpens) suffer from high resource underutilization causing sub-optimal performance, and inefficiency. To solve this challenge, we propose Chasoň, an HBM-based streaming accelerator for sparse kernels, specifically sparse matrix vector multiplication. Chasoň supports our novel non-zero scheduling scheme called Cross-HBM Channel out-of-order (OoO) Scheduling (CrHCS) to enable data migration across HBM channels and mitigate resource underutilization. We implement Chasoň on AMD Xilinx Alveo U55c achieving 301MHz clock frequency and evaluate it based on SuiteSparse and SNAP matrix collections. Chasoň improves the resource utilization and achieves up to 8×, 20.33×, 11.65×, and 2.67× performance improvement and 2.03×, 34.72×, 19.48×, and 14.61× better energy efficiency over Serpens, Nvidia RTX 4090, Nvidia RTX A6000 and Intel Core i9-11980HK, respectively. The source code for Chasoň will be available online soon.

Keywords

High Bandwidth Memory, Sparse Algebra, Streaming Accelerator, Data Migration, PE utilization, FPGA Implementation

1 Introduction

Sparsity has become an important characteristic of data across various domains. With increasing volume of data in applications from different domains for example scientific computing [2, 12, 15, 20, 25, 72], machine learning [47, 74, 80], genomics [63], optimization problems [5, 6, 19, 37, 59] and graph problems [8, 30, 43], the prevalence of sparse data has also increased. . Sparse algebraic operations are prominent in these applications. Sparse algebra is a term that encompasses the operations in which at least one of the operands is sparse. However, processing sparse data introduces unique challenges. Sparse algebraic kernels suffer from memory-bound performance due to non-uniform sparsity patterns. Despite numerous solutions proposed to accelerate sparse algebra, such as specialized hardware accelerators [5, 11, 19, 24, 49, 57, 66, 70, 79] and optimized software techniques [33, 52, 60, 64, 73], sparse data handling remains an active area of research.

One of the primary challenges in implementing sparse algebra is addressing the memory bottleneck, which arises from irregular data access patterns and the limited efficiency of traditional memory architectures. To tackle this, system architects have proposed novel advanced memory technologies, such as high-bandwidth memory (HBM) [29, 31, 34], 3D-stacked DRAM [4, 44], hybrid memory cube (HMC) [28, 55] and resistive RAM (ReRAM) [27, 78]. Among these, HBM has emerged as a cornerstone in accelerating memory-bound sparse algebra operations. An HBM is composed of multiple independent channels and allows for concurrent read and write operations. This parallelism results in significantly higher memory bandwidth compared to conventional DRAM architectures. In addition to high-performance computing platforms such as GPUs, HBM has also been integrated into reconfigurable platforms, enabling the development of domain-specific resource-efficient accelerators. For example, AMD Xilinx Alveo U280 and U55c data center accelerator cards offer peak HBM bandwidth of 273GB/s and 460GB/s respectively and can be deployed in reconfigurable systems.

The high bandwidth and concurrent accesses make HBM particularly well-suited for sparse algebraic operations. Researchers are actively developing innovative sparse algebra accelerators that leverage HBM's capabilities to mitigate memory bottlenecks and accelerate performance in sparse computation workloads [11, 23, 42, 57, 67, 68, 81]. There exist various approaches to integrate HBM into an accelerator's architecture. State-of-the-art accelerators [11, 23, 57, 67, 68] typically utilize HBM in a streaming fashion, where data is continuously streamed from the HBM to the compute units. However, this continuous data stream does not always translate into uninterrupted computations. This is because of the RAW dependencies in sparse algebraic operations. For instance, when an HBM channel streams a non-zero value from a particular row, the processing elements (PEs) require a certain number of clock cycles to complete the multiplication and accumulation (MAC) operation. During this time, processing the next non-zero value from the same row must be delayed to avoid memory bank conflicts and RAW dependencies.

To address this challenge, current solutions [23, 67, 68] have introduced an out-of-order (OoO) scheduling technique for row-based parallelization, known as PE-aware non-zero OoO scheduling. It is built on top of row-based non-zero scheduling. It schedules the non-zero values from rows that were assigned to an HBM channel to different PEs in round-robin fashion. Although this approach demonstrates significant improvements over existing works, it still leaves a considerable number of stalls in PEs, leading to high PE underutilization. The reason lies in its dependence on non-zero values of the rows mapped to the same channel to do the scheduling and fill the stalls. This intra-channel scheduling constraint restricts its ability to fully utilize the available PE resources (more details in Section 2.2).

MICRO 2025, October 18–22, 2025, Seoul, Korea
2025. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

In this work, we propose a novel OoO non-zero scheduling called cross-HBM channel OoO scheduling (CrHCS). CrHCS extends the intra-channel non-zero scheduling to inter-channel non-zero scheduling by allowing cross-HBM channel data migration. The main goal of CrHCS is to improve the PE utilization by reducing the number of PE stalls, achieved through the migration of non-zero values across HBM channels. We also propose our novel architecture Chasoň, to support CrHCS for sparse-matrix vector multiplication. The PEs in Chasoň generate the partial outputs that are associated with the current channel (private channel) as well as other channels (shared channels). We implement Chasoň on AMD Xilinx Alveo U55c using Vitis high-level synthesis (HLS) and TAPA [17] framework. We use Rapidstream Autobridge [18] to implement our design and generate the bitstream file. We evaluate Chasoň on a variety of 800 matrices from SuiteSparse [7] and SNAP [35] collection against Serpens [67], Nvidia RTX 4090, Nvidia RTX A6000 (cuSparse) and Intel Core i9-11980HK (Intel Math Kernel Library). Chasoň achieves 301MHz clock frequency and exhibits up to 8×, 20.33×, 11.65× and 2.67× performance improvement over Serpens [67], Nvidia RTX 4090, Nvidia RTX A6000 and Intel Core i9 respectively. In summary, this paper contributes the following:

- We show that the state-of-the-art PE-aware OoO non-zero scheduling results in a considerable PE underutilization as a result of intra-channel non-zero scheduling.
- We introduce our novel cross-HBM channel OoO scheduling to allow non-zero migration among HBM channels to reduce the number of pseudo stalls in the HBM channels' data streams and improve the PE underutilization.
- We propose Chasoň, a resource efficient architecture to support CrHCS for sparse algebraic kernels.
- We implement Chasoň tailored for SpMV on AMD Xilinx Alveo U55c, demonstrating a real-world deployment across diverse matrices, and getting improved PE utilization, performance and energy efficiency over the baselines.

2 Background, Challenges, and Motivation

2.1 HBM-based Sparse Accelerators

HBM is an advanced memory technology that uses vertically stacked memory dies to allow high read and write bandwidth. HBM is increasingly adopted in FPGAs and GPUs, where the need for rapid memory access and high throughput is critical. Prior studies [11, 23, 57, 67, 68, 81] have been leveraging the parallel accesses to HBM channels to accelerate the memory bound applications such as sparse algebra with certain trade-offs. Each HBM channel is typically assigned to a dedicated architectural module, e.g. processing element group (PEG), ensuring that the architecture aligns seamlessly with the way data was scheduled. For example, a prior study [11] introduces a new sparse matrix format that aids in reducing the read after write (RAW) dependency distance and moving non-zero values from the HBM to the compute cluster units. However, it works poorly on imbalanced matrices. Another accelerator [57] uses HBM channels to compute SpMV in parallel independently of other processing elements. It is an expensive design in terms of on-chip BRAM usage and runs at 221MHz frequency which is less than that of 237MHz in other work [11]. However, it gives better performance for imbalanced workloads. In Serpens [67],

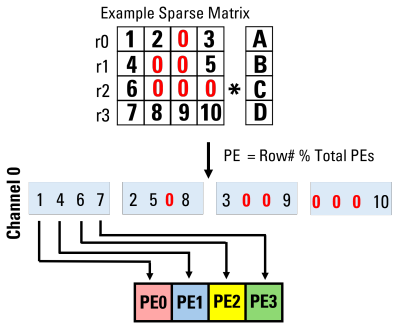

Figure 1: Row-based Non-zero Scheduling–A group of 4 values are mapped to four PEs in-order. These groups may have zeros because of lack of non-zero values in the rows mapped to specific PEs

the non-zero values are scheduled for each HBM channel. Similar to other HBM-based accelerators, it exploits the parallelism offered by HBM by allocating 8 PEs to each HBM channel. This configuration ensures that computations associated with different HBM channels are executed concurrently. Other studies [68] and [23] have proposed SpMM and sparse triangular solver (SpTRSV) accelerators and also work by processing the data streams coming from each HBM in parallel. In all these HBM-based accelerators, the performance is heavily influenced by how data is scheduled in each HBM channel and PEGs. As a result, efficient non-zero scheduling strategies are critical to fully leveraging HBM bandwidth and maximizing computational throughput—a challenge we explore in this work.

2.2 Non-Zero Scheduling

Row-Based Non-zero Scheduling. Another approach is row-based parallelization [3, 71, 76], where all non-zeros from the same row are scheduled for the same PE. Figure 1 shows an example of row-based scheduling. In this example, 4 rows are mapped to channel 0. These four rows are distributed across 4 PEs associated with channel 0. The four PEs are processed in parallel and get a value from a group. Rows are assigned to the PEs according to:

$$PE_{id} = row_{id} \% TotalPEs \tag{1}$$

$$PE[PE_{id}] \leftarrow row_{id} \tag{2}$$

Row 0 is assigned to PE0, row 1 is assigned to PE1 and so on. That is, PE0 gets `(1,2,3)` non-zero values. Figure 2a shows the timeline of PE0 pipeline, wherein row 4, row 8 and row 12 (all mapped to PE0) are also shown. We assume that the partial products are already available and now the PE is doing accumulation of these partial sums. The accumulation operations takes 10 cycles on the Xilinx Alveo U55C, U280, and U250 [23, 57, 67, 68] as shown by the 10 stages of accumulation instruction in the figure. Figure 2a also shows the PE0 instructions based on the row-based non-zero scheduling. We can observe the following:

- This pipeline is not fully utilized because second instruction (**`I2`**) has to wait for the first instruction (**`I1`**). The reason is RAW dependency between these two instructions. **`I2`** needs **`r0_op1`** which is available only after 10 cycles.

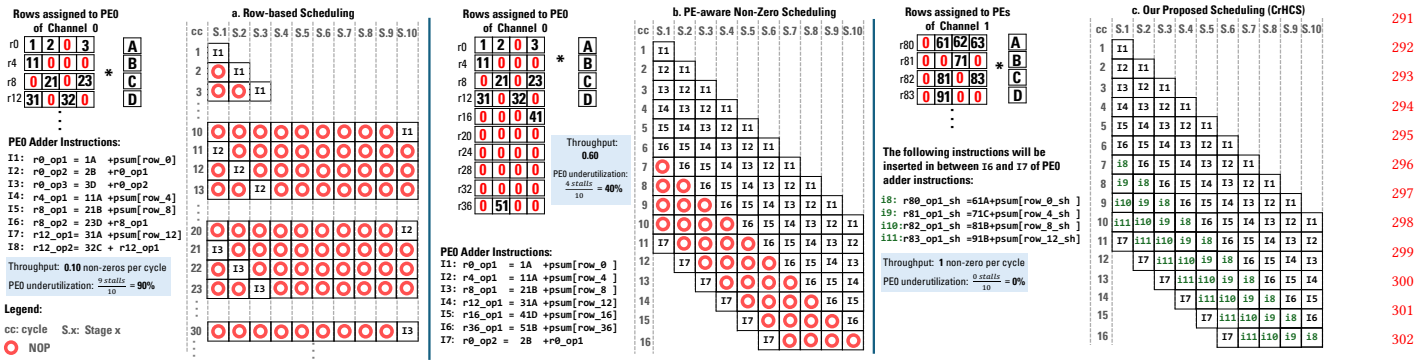

Figure 2: Timeline of PE0 associated with Channel 0 based on different non-zero scheduling schemes

Consequently, PE0 is highly underutilized and outputs only 0.10 non-zeros per cycle. A fair question that arises here is whether **I2** can begin execution based on the intermediate result from stage 1 of **I1**, especially given that PE0 performs accumulation over 10 stages. The answer is no. HLS tools and FPGA architectures do not support such fine-grained, stage-level forwarding. Instead, dependent instructions must wait for the complete output of their predecessors before execution can begin.

PE-Aware Non-Zero Scheduling. State-of-the-art works [23, 67, 68] use an out-of-order scheduling scheme called PE-aware non-zero scheduling. It is built on top of row-based parallelization to reduce the stalls in the pipeline of PEs. The main idea is to map rows to a PE in a round-robin fashion. Figure 2b shows an example of PE-aware non-zero scheduling. We can see that instead of assigning all values of row 0 to PE0 before proceeding to row 4 (similar to row-based non-zero scheduling), PE-aware non-zero scheduling interleaves the mapping—alternating between values from row 0 and row 4 in a round-robin fashion. It can be observed that PE0 can now start **I2** in 2nd cycle. This is an improvement over row-based scheduling which started **I2** in 11th cycle because of RAW dependency. However, this approach introduces a limitation. As shown in Figure 2b, row 20 to row 36 has no non-zero values. This will lead to stalls in the PE pipeline starting in cycle 6 of stage 1. For this example, there will be 5 stalls in the PE pipeline, leading to 50% PE underutilization. Instruction 7 (**I7**) can not start in cycle 7 because it depends on **(r0_op1)** which is being generated by **I1**.

PE-aware non-zero scheduling maps at least 10 rows per PE. Whether this leads to improved PE utilization ultimately depends on the number of non-zero values in those rows. In case it fails to find any non-zero value in a row, PE-aware non-zero scheduling places a zero in the datalist of channel. These zeros are analogous to idle PEs. In the architecture, when a zero is sent to a PE, all associated computations—such as multiplication and accumulation—are skipped, signifying the same stalls as shown in Figure 2b.

In the prior works [23, 67, 68], PE-aware non-zero scheduling explicitly inserts these zeros into the data list of HBM channels to guide the HLS tools to maintain the pipeline initiation interval (II) equal to 1. Otherwise, HLS will conservatively increase the II to 10 cycles [68], resulting in unexpected sub-optimal performance. On the other hand, an architecture designed using register-transfer level (RTL) does not require explicit zeros in the data list. Instead, the pipeline naturally stalls when no valid computation is available, without enforcing an increased initiation interval (II). Hence, avoiding the need to pad the data list explicitly.

Figure 3 shows that, for most real-life datasets, PE-aware non-zero scheduling still leaves around 70% of the PEs underutilized. To get these results, we ran experiments on 800 matrices from SuiteSparse [7] matrix collection. The matrices are from different domains and their density ranges from 10^{-5} % – 10^{1} %. We plot the percentage of stalls (PE underutilization %) in Figure 3 as a probability density function to show the PE underutilization percentage in 800 matrices. The main reason for PE underutilization is that PE-aware non-zero scheduling only uses the non-zeros from the rows that are mapped to a specific HBM channel. In case there are not enough non-zero values in the rows that are assigned to a PE of that HBM channel, it lacks the ability to go fetch a non-zero value that belonged to a row of another HBM channel.

Our Proposed Scheduling. The goal of this work is to realize such data retrievals by enabling non-zero migration across HBM channels. Figure 2c shows the pipeline based on our proposed novel scheduling scheme, CrHCS. To keep the pipeline filled, CrHCS fetches non-zero values from the neighboring channel. CrHCS is able to fetch the values from the rows, irrespective of which PE they are assigned to in their channel. As shown in Figure 2c, PE0 of channel 0 now executes the instructions that belonged to any of the four PEs associated with channel 1. These instructions are inserted in between **I7** and **I6**. Consequently, the pipeline remains filled and the PE underutilization goes to 0%, providing maximum throughput.

2.3 Key Insight

To mitigate the aforementioned challenges, our key insight is that the scheduling mechanism can be simply extended to fetch or migrate values from a neighboring channel to fill the stalls in the PE. Based on this, we introduce a novel scheduling scheme, cross-HBM channel out-of-order scheduling (CrHCS), to enable data migration across HBM channels, thereby reducing the percentage of stalls and effectively improving PE utilization and performance efficiency. On the architecture side, we propose our novel and efficient architecture, Chasoň, to support CrHCS and segregate computations corresponding to data from various HBM channels, ensuring synchronized and functionally correct accelerated sparse algebraic operations.

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

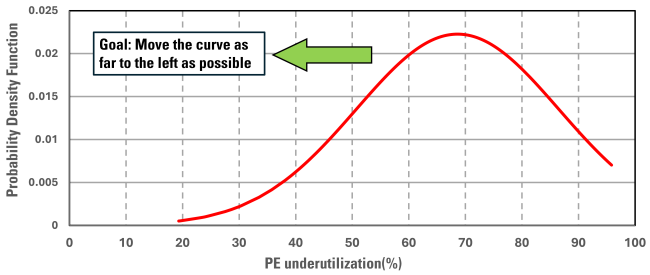

Figure 3: Percentage of stalls based on state-of-the-art PE-aware non-zero scheduling [23, 67, 68] in 800 SuiteSparse matrices (lower is better)– 70% PEs are underutilized for majority of 800 matrices. Our goal is to move the curve to as far left as possible, that is, reduce the PE underutilization.

3 Chasoň – Scheduling

In this section, we introduce our novel OoO non-zero scheduling, CrHCS. The number of zeros in an HBM channel data list are analogus to idle PEs (as descirbed in Section 2.2).

3.1 Cross-HBM Channel OoO Scheduling

CrHCS is built on top of state-of-the-art PE-aware non-zero OoO scheduling [23, 67, 68]. The key idea of CrHCS is to incorporate cross-HBM data migration to improve the PE utilization by migrating the non-zero values from a data list of one channel to the other. CrHCS can migrate the non-zero values from more than one neighboring channel. However, in our discussion and implementation we limit the data migration to the immediate next channel only. Figure 4 shows the overview of the CrHCS. Each HBM channel is associated with a group of 4 PEs, referred to as a Processing Element Group (PEG). These 4 PEs run in parallel. In Figure 4, only the PEG for channel 0 is shown. Initially, the number of stalls in HBM channel 0 is 6. This signifies 6 instances of idle PEs and hence, a PE underutilization of 50% in the PEG of channel 0. CrHCS migrates the values from the first next channel, that is, channel 1 and schedules them in channel 0. As a result, channel 0 is full of non-zero values, and its corresponding PEG will be 0% underutilized. Similar to the prior works, the data lists of each channel will be resized to make them all equal to the longest channel list, allowing synchronized completion of the final computation. The details of CrHCS are discussed below.

3.2 Data Structure for CrHCS

According to [45], the ideal bitwidth of read (Rd) or write (Wr) modules for an HBM channel is 512 bits. Prior works [11, 23, 57, 67, 68] use 64 bits per sparse element. They allocate 32 bits for the non-zero values and 32 bits for the row and column indices. Eight 64-bit values are coalesced and sent to the PEG corresponding to the specific HBM channel. The order of coalescing the eight sparse elements directly determines the PE assigned to it in the PEG. For instance, the first 64-bit value is allocated to PE0, the second 64-bit element to PE1, and so on. This ensures synchronization with the other architectural modules.

In CrHCS, the stalls are replaced by migrating the data from the first next HBM channel. However, it leads to functional incorrectness due to on-chip memory bank conflicts in the partial

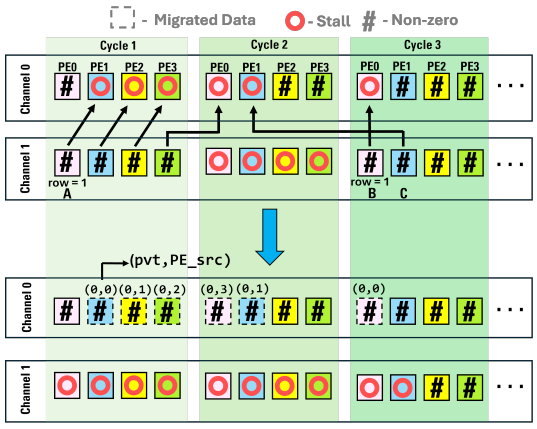

Figure 4: Functionality of CrHCS applied on channel 0– Data is migrated from channel 1 (shared channel) to channel 0 (private channel). CrHCS also respects the RAW dependency among the migrated data to keep the pipeline filled. The stalls are reduced from 6 to 0 in the PEs associated with Channel 0 after applying CrHCS.

sum accumulation step, because the partial outputs of neighboring channel values may accumulate with current channel values. A memory conflict occurs if both sets of values attempt to access the same memory index simultaneously, causing read/write contention. This results in an increased initiation interval and ultimately degrades performance. requires To avoid this, CrHCS uses a unary bit `pvt` flag to track the source of non-zero value. The flag indicates whether a non-zero value belongs to another HBM channel (`pvt=0`) or belongs to the current HBM channel (`pvt=1`). CrHCS also keeps track of the PE the non-zero value originally belongs to using the `PE_src` flag. Given that the PEG consists of eight PEs (details in Section 4.2), `PE_src` only needs 3 bits. Without `pvt` and `PE_src` flags, Chasoň could mistakenly accumulate shared channel partial outputs with private channel partial sums, leading to incorrect accumulation and corrupted results. The overall distribution of one 64-bit element in CrHCS has 32-bit float value, 15-bit row index, 1-bit `pvt` flag, 3-bit `PE_src` flag, and 13-bit column index.

3.3 RAW Dependency in the Migrated Data

CrHCS must respect the dependency distance between non-zero values coming from the neighboring HBM channel. Consider the example in Figure 4. CrHCS is applied on channel 0, that is, the data is migrated from channel 1 to channel 0. We can see that the non-zero values **A** and **B** from channel 1 belong to the same row (row 1). CrHCS schedules **A** in the channel 0 for PE1 in 1st cycle (`(pvt,PE_src) = (0,0)`). If the dependency distance is assumed to be 2 cycles, to avoid RAW dependency and keep the pipeline of a PE1 filled, there must be a distance of 2 cycles before we schedule another non-zero from row 1 in PE1 of channel 0. This is the reason why when B is the candidate to replace the stall in channel 0 for PE1 in 2nd cycle, CrHCS skips B and rather uses C value from channel 1. CrHCS keeps track of the earliest cycle in which a non-zero value is scheduled for a PE in the destination i.e., if the current cycle of the corresponding PE in the destination channel is lower, CrHCS skips the non-zero value in channel 1 and proceeds to the next non-zero

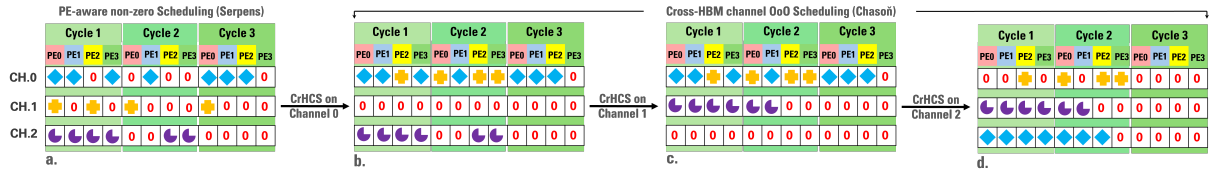

Figure 5: An example of scheduling using CrHCS on top of PE-aware non-zero scheduling

value. Our experiments have shown that CrHCS never fails to find a RAW dependency-free value to migrate.

3.4 Advantage of using CrHCS-Example

Figure 5 shows an example of CrHCS applied on top of PE-aware non-zero scheduling. In this example, there are three channels and each is connected to a PEG that has four PEs. The HBM is capable of sending 4 values every clock cycle, which are sent to four PEs to be processed in parallel. It can be seen that PE-aware non-zero scheduling results in 19 zeros (or equivalently 19 instances of idle PEs) across 3 HBM channels and hence, $19/36 = 52\%$ PE underutilization. The stalls are due to the reasons explained in Section 2.2. In Figure 5b-c, CrHCS is applied on this scheduled data. In this example, we assume that there is no RAW dependence among the migrated data and all the non-zero values can be migrated to a neighboring channel.

In the first step, Figure 5b, CrHCS migrates the non-zero values from channel 1 to channel 0 to fill the stalls. All the values of channel 1 can be migrated as we assumed that there is no RAW dependence in the migrated data,

This results in significantly large number of stalls in channel 1. CrHCS continues to fill these stalls using the values from channel 2 as seen in Figure 5c. It is interesting to observe that all the non-zero values from channel 2 are placed contiguously. This raises an important question: why does CrHCS not take into account the stalls shown in channel 2 in Figure 5a? It is because now the non-zero values will be accumulated in another PE. The partial sums of different PEs of shared channels are segregated using different URAMs (more details in Section 4.2). For example, the channel 2 non-zero value originally scheduled for PE2 in cycle 2 will be now processed in cycle 2 in PE0 associated with channel 1. And based on our assumption that there is no RAW dependency between the migrated value, it is allowed in PE0 of channel 1.

In the last step, CrHCS schedules channel 2 by migrating the values from channel 0 as shown in Figure 5d. Note that CrHCS only migrates the values that originally belonged to channel 0, that is, the blue values. This introduces some stalls in Channel 0 but this step is necessary to ensure a minimal load imbalance between the PEGs associated with Channel 0 and Channel 2.

We can deduce two important advantages of CrHCS from this example. First, all the non-zero values can now be processed in only two cycles. This also reduces the number of transfers from the HBM to the PEs and consequently increases the throughput of the underlying architecture. Second, now that the data is scheduled among two cycles, the number of stalls and equivalently, PE underutilization is reduced to $7/24 = 29\%$.

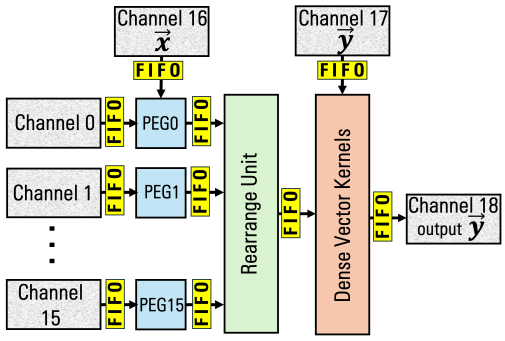

Figure 6: High Level Architecture of Chasoň– Each channel streams eight 64-bit sparse matrix elements to a PEG. The partial output data from each PEG is arranged in a single stream in the Rearrange Unit. FIFO streams are used to move data between different units.

4 Chasoň – Architectural Support

In this section, we introduce our novel architecture Chasoň, which provides the architectural support required by CrHCS in the state-of-the-art OoO SpMV accelerator, Serpens [67]. We will briefly discuss the SpMV processing order and go into more detail about the architectural details of Chasoň.

4.1 High-Level Overview

Chasoň is a streaming accelerator built on the top of Serpens. The high-level architecture is shown in Figure 6. It uses 16 HBM channels to stream in sparse matrix A, which has already been scheduled offline using CrHCS. Each of these 16 HBM channels has a PE group (PEG) that houses 8 PEs. Dense input vector $\vec{x}$, $\vec{y}$ and dense output vector $\vec{y}$ are allocated only one channel each owing to their smaller size as compared to sparse matrix A. Due to limited on-chip memory, the entire dense vector $\vec{x}$ can not be saved in it. For that reason, the problem is partitioned into segments of size W, equal to 8192, as the column indices are reduced to 13 bits (described in 3.2). The instruction order is generated in the preprocessing and passed on to the PEG using one of the HBM channels. The SpMV processing order is similar to Serpens [67].

4.2 Processing Element Group (PEG)

As shown in Figure 7a, PEG has eight PEs and a Reduction Unit. A PE performs the MAC operation and stores the resulting partial sums in the correct on-chip memory location. The Reduction Unit is responsible for gathering and reducing these partial sums. The details are discussed below.

4.2.1 ***Processing Element (PE).*** A PE has a multiplier, an adder unit, and on-chip memory. The non-zero value is multiplied by the

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

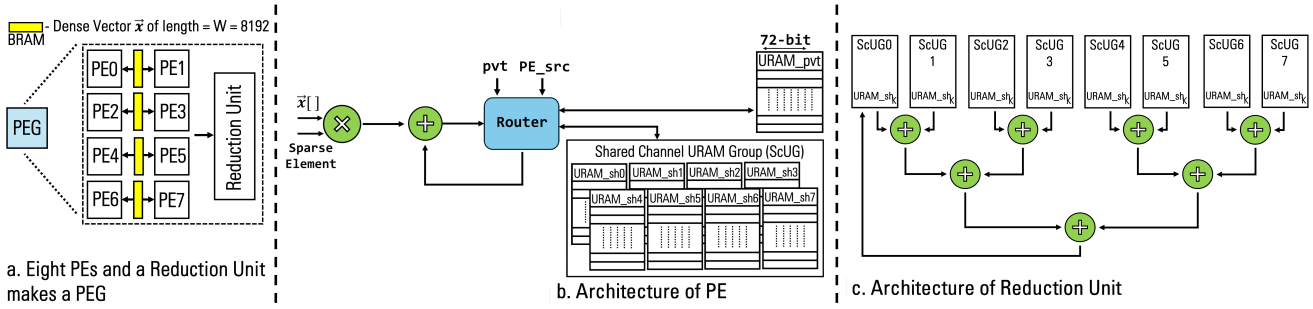

Figure 7: Architecture of a PEG– There are eight PEs and a Reduction Unit in a PEG. Partial outputs associated with private and shared channels are routed to their respective on-chip memory. The Reduction Unit gathers the partial outputs and reduces them using an adder tree configuration.

dense vector $\vec{x}$ saved in on-chip dual-port block RAM (BRAM) similar to Serpens [67]. The BRAMs are dual-port, and hence, Chasoň requires only four of them to store the dense vector $\vec{x}$ as shown in Figure 7a. The accumulated partial sums are stored in the on-chip memory, which is a 72-bit wide ultra RAM (URAM) and can house two FP32 partial sums per slot. This on-chip data storage limits the random accesses to $\vec{x}$ and partial sums within the chip. To support CrHCS, Chasoň has to do the following:

- Segregate the partial sums associated with a neighboring/shared channel from the partial sums belonging to the current/private channel.
- For the non-zeros values coming from the shared channel, it must also segregate the partial sums that belong to different PEs in its origin channel. For example, the partial sums computed in PE0 of channel 0 may belong to PE7 and PE6 of channel 1 and Chasoň must keep track of the partial sums associated with these different PEs.

Chasoň uses `(pvt,PE_src)` flags encoded in the input data to achieve this goal. If the multiplication output belongs to a private channel (`pvt=1`), then the partial sum will be fetched from URAM_pvt for the addition operation. However, if it belongs to a shared channel (`pvt=0`), Chasoň must identify its corresponding PE in the shared channel. This is achieved using `PE_src` flag. The partial sum is fetched from the corresponding `URAM_sh` in Shared Channel URAM Group (ScUG) and sent to the adder unit. For both private and shared URAMs, the output of the adder is written back to the same location from which the partial sum was retrieved. This data routing is achieved by a simple combination of muxes, housed within the Router unit in each PE. Data routing within the PE is necessary to maintain the functional correctness of the SpMV kernel.

4.2.2 ***Reduction Unit.*** Each PE consists of one ScUG, resulting in eight ScUGs per channel as there are eight PEs per channel. Each URAM in the ScUG stores the data for relevant PE in the shared channel. For example, in channel 0, `URAM_sh0` in all eight ScUGs will store the partial sums for PE0 of channel 1. In the Reduction Unit, the architecture sweeps through all the URAMs in the eight ScUGs and reduces their values using an adder tree configuration, consolidating the results into a single URAM per PE as shown in Figure 7c.

As shown in Figure 7c, the k-th `URAM_sh` from all the eight ScUGs will be gathered and reduced, and the final output will be written back to `URAM_sh0`. The final values in this `URAM_sh0` represent the partial sums corresponding to the k-th PE of the shared channel (where $k = 0, 1, ...7$).

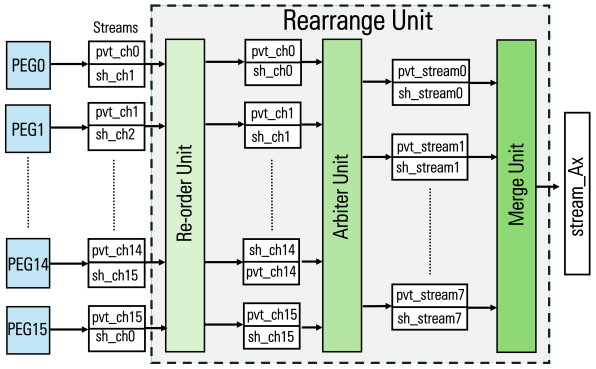

Figure 8: Functionality of Rearrange Unit– The Re-order Unit arranges the streams associated with shared channel in the same order as the streams associated with private channel.

4.3 Rearrange Unit

Each PEG has final partial sums in eight `URAM_pvt` and eight `URAM_sh0`. Their values will be gathered into one stream of 8 FP32 values each, that is, `pvt_ch` and `sh_ch`. As shown in Figure 8, the streams of shared channel (`sh_ch`) coming from PEG do not have the same order as streams of private channel (`pvt_ch`). This occurs because Chasoň employs the CrHCS strategy to perform MAC operations corresponding to the neighboring channels. To ensure functional correctness, the shared streams must correspond to the channel they belong to. The Re-order Unit handles this task by rearranging the streams to ensure that both the private and shared streams are aligned in the correct order, as shown in Figure 8. The Arbiter Unit and Merger Unit works similarly to the ones in Serpens [67] ow they concatenate as well as reduce *two* types of streams, that is, private channel (`pvt_ch`) and shared channel (`sh_ch`) streams. These units only do concatenate operation and solely on private channel streams in Serpens [67]. In addition to collecting (`pvt_ch`) values from 2 neighboring channels, the Arbiter Unit in Chasoň will do the same for (`sh_ch`) and pass them to the Merge Unit. In the Merge Unit eight pvt_streams and eight sh_streams will be added/reduced to make sure all output values corresponding to a channel have been computed. The resultant eight streams will

be then merged into one 16 FP32 values stream (stream_Ax) and passed to the Dense Vector Kernels unit shown in Figure 6. Dense vector kernels are trivial logic and work on a single stream of data.

4.4 Novelty over Serpens Architecture

Serpens [67] employs eight PEs per PEG, similar to Chasoň. However, its PEs can only process data from private channels, as Serpens [67] not support cross-channel data migration. Each PE in Serpens consists of a multiplier, an adder, and a URAM for storing partial outputs. Additionally, Serpens [67] lacks a Reduction Unit. While it includes an Arbiter Unit and a Merger Unit, these components solely concatenate private streams without performing any reduction, unlike Chasoň (as described in Section 4.3). To ensure a fair comparison, Chasoň maintains the same level of parallelism as Serpens [67]. Both architectures use identical HBM and on-chip memory configurations for storing input data, resulting in the same number of PEGs.

4.5 Resource Consumption

Chasoň uses on-chip memory to support CrHCS. The partial sums associated with a private channel are stored in each PEG in one URAM (`URAM_pvt`). The partial sums for shared channel are stored in ScUG in each PEG, and each ScUG has a size of 8 URAMs. Hence, the total number of URAMs required are:

$$\begin{aligned} \text{Number of URAMs} &= (PEG \times 8) + \\ &\quad + (CH_A \times PEG \times ScUG\ Size) \end{aligned} \tag{3}$$

The total number of URAMs is 1024, which is more than the available 960 URAMs on Alveo U55c. To address this, we reduced the design size by decreasing the number of URAMs per ScUG to 4, bringing the total URAM usage down to 512 (52% of available). Theoretically, each PE requires at least one `URAM_sh` in ScUG and one `URAM_pvt`. This reduces the URAM usage to 256, each providing $36KB$ storage on Alveo U55c. Although it does not affect the performance, it results in decreasing the size of the input sparse matrix A that can be processed in a single pass. In such a situation, we partition the bigger sparse matrix A and feed the partitions into Chasoň.

As mentioned earlier, our implementation of Chasoň utilizes a total of 512 URAM blocks. This results in a total on-chip memory requirement of 18 MB for storing partial outputs in Chasoň, which is slightly higher compared to the 13.5 MB employed in Serpens [67]. The increased URAM usage is required to segregate the partial sums of shared and private channels and is fundamental to the architecture of Chasoň. Importantly, this additional URAM is used solely for data storage and does not impact the critical computation path. As such, it does not introduce any performance bottlenecks. In the Alveo U55c platform, URAM blocks have uniform access latency, regardless of quantity, which ensures that scaling up or down URAM usage does not degrade access latency or stall the pipeline. To store the dense vector $\vec{x}$, similar to Serpens [67], Chasoň uses 1024 18Kb dual port BRAM blocks (32 BRAMs blocks per PEG), that is, consuming a total of 2.5MB on-chip memory for $\vec{x}$. Autobridge [18] seamlessly generates the bitstream file for Chasoň. The final layout of Chasoň, shown in Figure 9, achieves a frequency of **301MHz** outperforming the **223 MHz** frequency of Serpens.

Table 1: Xilinx Alveo U55c Resource Consumption for Chasoň and Serpens[67].

	Architecture	
	Serpens	Chasoň
LUT	219K(16%)	346k(26%)
FF	252K(9.6%)	418K(16%)
DSP	798(9.6%)	1254(13%)
BRAM18K	1024(28%)	1024(28%)
URAM	384(40%)	512(52%)

Chasoň achieves an increased frequency due to reduced logic congestion and distributed on-chip memory traffic as compared to Serpens [67] which routes all the partial outputs generated by a PE to only one URAM.

The detailed resource utilization for Chasoň and Serpens [67] is given in Table 1. It compares the FPGA resource utilization of Serpens and Chasoň. For the sake of fair comparison, they both use the same number of PEs and PEGs to maintain equivalent levels of parallelism. The additional FPGA resources used by Chasoň are used to implement the Reduction Unit, Re-order Unit and Router. If we were to perform an iso-area comparison by scaling up Serpens to use the same amount of FPGA resources as Chasoň does, Serpens could have deployed more than eight PEs per PEG. However, such a configuration would not translate to speedup in Serpens because the eight PEs per PEG is selected to offer the right balance between memory and computation speed as 512-bit HBM channel width dictates eight non-zero values per read. While the numbers reported in Table 1 reflect the hardware cost for both Chasoň and Serpens, the reported resource consumption is not directly related to the PE underutilization, the key metric that Chasoň improves. Accordingly, using a resource-constrained FPGA, and improving the percentage of resource consumption, does not imply improved PE utilization.

5 Experimental Setup

In this section, we discuss the methodology used to build and evaluate Chasoň. The experimental methodology is similar to prior works, ensuring a fair and comprehensive comparison.

5.1 Hardware Implementation

Chasoň is implemented using Xilinx Vitis 2023.2 C++ high-level synthesis (HLS) and TAPA [17] framework. We use AMD Xilinx Alveo U55c for hardware implementation. Alveo U55c has 16GB of 32-channel HBM, providing peak memory bandwidth of 460 GB/s (14.37GB/s per channel). Chasoň uses 19 channels and achieves the peak bandwidth of 273GB/s. Alveo U55c is connected via PCIe Gen3 x16 to allow quick data transfer between the host and HBM. We use Rapidstream Autobridge [18] to place and route Chasoň on Alveo U55c. Figure 9 shows the layout of Chasoň. Alveo U55c has three super logic regions that are further divided into blocks (or clock regions). The FPGA resources are distributed across these blocks. For example, the on-chip memory in Figure 9 may appear concentrated on one side but it is actually spread across different SLRs and further different blocks. The HBM memory is distributed in stacks of two, each with 16 channels. Autobridge [18] maps the majority of Chasoň logic, shown in blue, in the first two Super Logic Regions (SLRs). The on-chip memory is mapped on the fixed cell regions shown in orange. Chasoň achieves a frequency of **301MHz** and **48.715W** estimated power. The power distribution is shown in Figure 10. We can observe that Chasoň logic is only

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

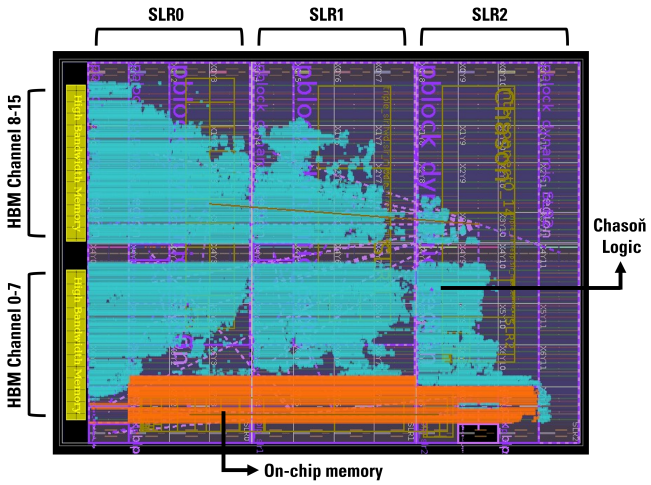

Figure 9: Layout of Chasoň implemented on Xilinx Alveo U55c– The logic of Chasoň is shown in blue with on-chip memory components in orange. Autobridge [18] maps the majority of Chasoň's logic near the HBM channels.

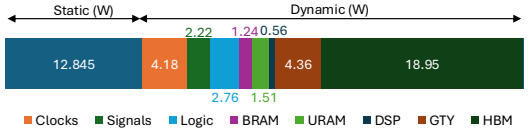

Figure 10: Power distribution of Chasoň implemented on Xilinx Alveo U55c– HBM is taking the most amount of power, with Chasoň's logic only taking 8% of the total power.

taking 8% ($2.76W$) of the total power. The on-chip memory power consumption is also very small, that is, 3% ($1.24W$) and 4% ($1.51W$) for BRAM and URAM, respectively. We also verify that the output of the SpMV kernel is correctness, ensuring end-to-end functional correct of Chasoň. This confirms the reliability and consistency of both CrHCS and its architectural support.

5.2 Baselines

We compare Chasoň with the state-of-the-art OoO SpMV accelerator, Serpens [67], the source code of which is available online. The HBM configuration is kept the same for both Chasoň and Serpens [67]. Serpens [67] also partitions the problem size into a window of $W = 8192$. To ensure a fair comparison, we generate its bitstream file for Alveo U55c using Rapidstream Autobridge [18]. It achieves **223MHz** frequency which is less than **301MHz** frequency of Chasoň. We also compare Chasoň against the official open-source cuSparse library's SpMV implementation [52], running on Nvidia RTX 4090 and Nvidia RTX 6000 Ada with cuda v10.1. The consumer-class Nvidia RTX 4090 offers 24GB of GDDR6X 384-bit wide memory with 1008GB/s bandwidth. However, the server-class Nvidia RTX 6000 Ada offers 48GB GDDR6 384-bit wide memory with 768GB/s bandwidth. They both have 128KB lower-level cache per SM. Their L2-cache is 72MB and 96MB respectively. RTX 4090 features 144 Streaming Multiprocessors (SMs) and RTX A6000 has 84 SMs, with each SM containing 128 CUDA cores. For the CPU baseline, we use Intel oneAPI Math Kernel Library (MKL) v2024.2.2 running on Intel Core i9-11980HK. It runs at the base frequency of 3.3GHz and has 24MB Intel Smart Cache. We run the GPU baselines for 10 iterations and average the performance metrics. For the CPU experiments we calculated an average of 100 measured iterations after 100 warm-up runs. For Chasoň and Serpens, we perform 1000 iterations of each experiment. The higher iteration count than CPU and GPU is necessary to amortize the overhead associated with bitstream transfer and FPGA reconfiguration, allowing us to better evaluate the raw performance of the SpMV kernel itself.

Table 2: SuiteSparse [7] and SNAP [35] matrices.

ID	Dataset	NNZ	Density %
SuiteSparse Matrices			
DY	dynamicSoaringProblem_8	38136	0.303
RE	reorientation_4	33630	0.455
C5	c52	20278	0.00035
MY	mycielskian12	407200	4.31
VS	vsp_c_30_data_data	124368	0.102
TS	TSC_OPF_300	820783	0.859
LO	lowThrust_7	211561	0.0700
HA	hangGlider_3	92703	0.0880
TR	trans5	749800	0.00541
CK	ckt11752_dc_1	333029	0.0138
SNAP Matrices			
WI	wiki-Vote	103689	0.1506
EM	email-Enron	367332	0.0272
AS	as-caida	106762	0.0108
OR	Oregon-2	65406	0.0469
WK	wiki-RfA	188077	0.145
SC	soc-Slashdot0811	905468	0.0151
A7	as-735	26467	0.0444
CM	CollegeMsg	20296	0.562
WB	wb-cs-stanford	36854	0.0374
RE	Reuters911	296076	0.1667

5.3 Evaluated Metrics

To ensure a fair comparison, we adopt the methodology for measuring evaluation metrics that align with the methods employed in prior works [11, 23, 57, 67, 68].

PE Underutilization: It is measured as the percentage of instances the PEs remain idle relative to the instances they are actively engaged in performing computations on non-zeros. Since Chasoň is designed using HLS, an instance of idle PE is represented as a 0 in the data list of HBM channels (described in Section 2.2). As a result, we can conveniently measure the PE underutilization offline by finding the percentage of stalls in data of all 16 sparse matrix A channels. Mathematically, it is given as:

$$\text{PE Underutilization } \% = \frac{\sum_{ch=0}^{15} \text{Number of Stalls}}{NNZ + \sum_{ch=0}^{15} \text{Number of Stalls}} \times 100 \quad (4)$$

Latency: We calculate the time taken by Chasoň and Serpens using Xilinx Run Time and standard C++ time routines. We use `cudaEventElapsedTime` to get the execution time on GPU. We further validate the kernel execution time using the latest Nvidia Nsight Compute. For the CPU, we use Intel MKL built-in profiling functions to measure the execution time. We run the execution multiple times and take an average to nullify the effect of any unwanted overheads.

Throughput: For all the underlying architecture, we measure the throughput (*GFLOPS*) of the SpMV kernel as:

$$\text{Throughput} = \frac{2 \times (NNZ + K)}{Latency(ns)} \quad (5)$$

where, NNZ is the number of non-zeros and K is the size of dense vector $\vec{x}$, or equivalently, the number of columns in sparse matrix A.

Energy Efficiency: For FPGA implementations, we calculate the power consumption using `xbutil`. We utilize `nvidia-smi` for GPU measurements. For CPU, we sample the package-level Intel RAPL counter located at `/sys/class/powercap/intel-rapl:0/energy_uj` and divide it by the time elapsed to get power. Energy efficiency is measured as the throughput harnessed per unit watt. Ideally, we want to harness maximum throughput by spending the smallest amount of energy. It is given as:

$$\text{Energy Efficiency} = \frac{Throughput(GFLOPS)}{Power(W)} \tag{6}$$

Bandwidth Efficiency: It is measured as the throughput harnessed per giga bytes of the transfer data.

$$\text{Bandwidth Efficiency} = \frac{Throughput(GFLOPS)}{Bandwidth(GB/s)} \tag{7}$$

5.4 Dataset

We use 800 matrices from SuiteSparse [7] and SNAP [35] dataset to evaluate Chasoň. The density of these matrices ranges from 10^{-6} to 10^{-1}, with the number of non-zeros (NNZ) varying from 10^3 to 10^6. The selected matrices are small enough to fit within the L2 cache of the GPU and the 24 MB L3 cache of the CPU. This makes it even more challenging for Chasoň to achieve speedup, highlighting the significance of its gains. For an in-depth analysis, we evaluate Chasoň against Serpens [67] on 20 matrices from SuiteSparse [7] and SNAP [35] collection shown in Table 2. These matrices are randomly selected to better reflect the inherent randomness and structural unpredictability of sparse matrices, ensuring an unbiased evaluation of Chasoň.

5.5 Data Precision:

Chasoň uses 32-bit floating-point values, consistent with Serpens. Each non-zero also carries necessary 32 bits of metadata (as detailed in Section 3.2), allowing a 512-bit HBM channel to transfer up to 8 non-zero entries per cycle. The number of non-zeros transferred and the parallelism in each PEG depend directly on the bit precision. Reducing the precision enables more than 8 PEs to be instantiated and operated in parallel, though it increases overall memory demand as more `URAM_sh` will be required per ScUG. Conversely, higher-precision formats reduce the number of parallel operations. For instance, using 64-bit floating-point values with 32-bit metadata limits both Chasoň and Serpens to transferring only 5 non-zero entries per cycle, which means that not all 8 PEs can be fully utilized and the parallelism in each PEG reduces from 8 to 5 PEs and similarly required `URAM_sh` per ScUG reduces to 5.

6 Evaluations and Results

In this section, we evaluate Chasoň with the aforementioned baselines. We give a detailed analysis of improvement in PE underutilization and discuss how it translates to performance gains. In the end, we will discuss the on-chip resource consumption of Chasoň.

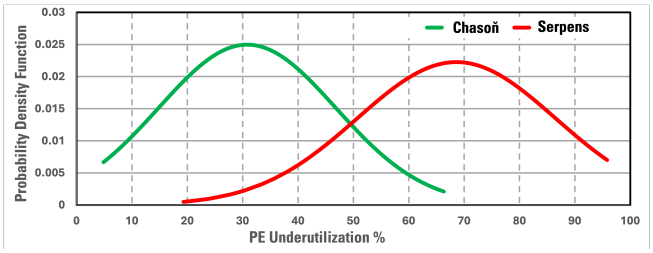

Figure 11: PE underutilization % in Chasoň and Serpens for 800 SuiteSparse [7] and SNAP [35] matrices (lower is better).

6.1 PE Underutilization

One of the main goals of this work is to reduce the PE underutilization, that is, to ensure that the PEs are actively engaged in computations for the majority of the time. We run experiments on 800 matrices to evaluate the PE underutilization exhibited in Chasoň and Serpens [67] as a result of doing CrHCS and the PE-aware non-zero scheduling, respectively. In Figure 11a, we plot the PE underutilization for both of them as a probability density function (PDF). The y-axis shows the likelihood or frequency of a given PE underutilization percentage occurring across the experiments. In Serpens [67], there is a high probability of PE underutilization greater than 50%, with the most likely rate being 69%. Chasoň reduces the PE underutilization to approximately 30%, with the majority of datasets demonstrating resource underutilization below 50%. This is shown in Figure 11a, where underutilization below 50% exhibits a high probability for Chasoň. We also show the PE underutilization for 800 matrices in Figure 11b. Chasoň exhibits PE underutilization in the range of 5% - 66%, which is significantly lower than the range of 19% - 96% in Serpens.

In addition, Figure 12 shows the experiments on 20 datasets from Table 2 for an in-depth analysis of PE underutilization. Figure 12 shows the PE underutilization as a function of PDF across 16 PEGs for each matrix. We observe that in Chasoň, the PE underutilization for each PEG is significantly smaller than Serpens. The widened PDF curve of Chasoň highlights its ability to effectively balance workloads across PEGs, showcasing its adaptability to irregular or imbalance matrix patterns. This flexibility ensures that Chasoň maintains lower PE underutilization, making it particularly suitable for real-world scenarios where workload distribution is unpredictable and highly random. On the other hand, the narrower PDF curve of Serpens [67] reveals its limitations in handling imbalanced matrices, as it struggles to adapt to irregular sparsity patterns and dynamic workload variations. This results in higher PE underutilization, reducing its efficiency in less-structured workloads.

Ideally, to ensure fairness, a scheduling policy should evenly distribute the stalls among PEG, such that no any PEG has significantly disproportionate PE underutilization. Figure 13 shows average PE underutilization in each PEG for 20 matrices listed in Table 2. For Serpens [67], the underutilization reaches 95%. Chasoň reduces this number to 60-65%. We observe that Chasoň distributes the stalls evenly among all the PEGs. The variation between PE underutilization of the PEGs is not that much, which means Chasoň also ensures fairness while distributing stalls among 16 PEGs.

The PE underutilization is significantly reduced in Chasoň as compared to Serpens. However, the underutilization in Chasoň is not entirely eliminated, as achieving 0% resource underutilization

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

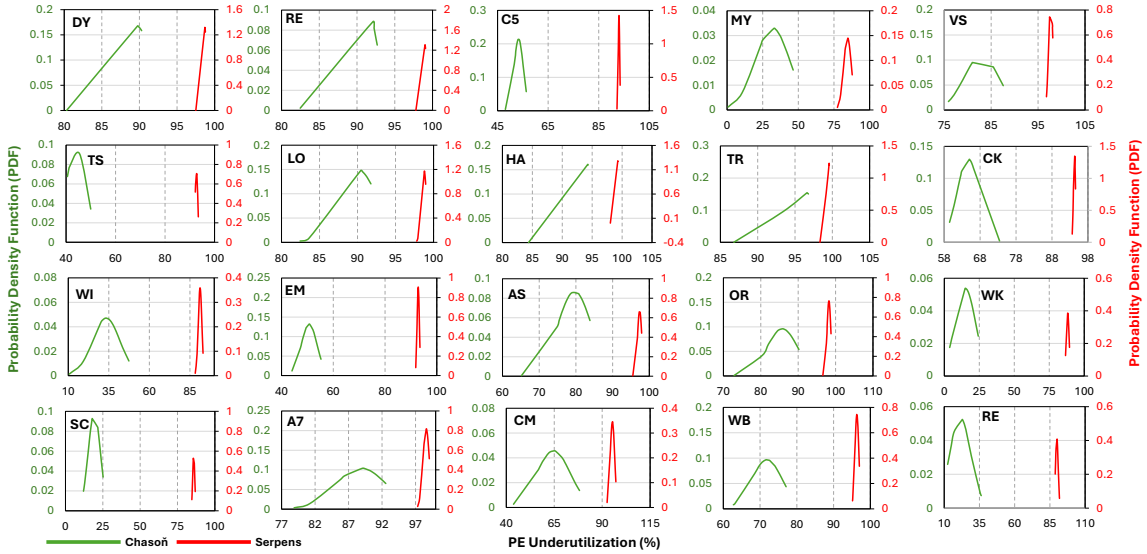

Figure 12: PE underutilization % across 16 PEGs for the matrices listed in Table 2– CrHCS reduces the number of stalls in the data list of each channel, resulting in significantly smaller PE underutilization in Chasoň as compared to Serpens [67].

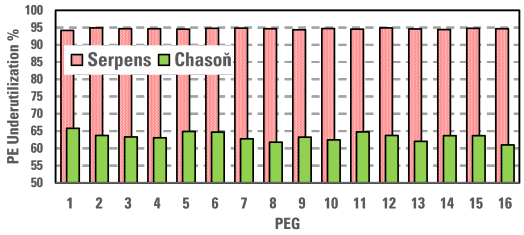

Figure 13: Average PE underutilization % for each PEG for the matrices listed in Table 2.

remains unattainable. The reason is that CrHCS fails to find enough non-zero values in the first next channel to fill the stalls in a private channel. CrHCS can get more data by referring to second next channel. However, segregating the partial outputs for all shared channels and the private channel requires more on-chip memory in Chasoň. Given the limited on-chip resources available on the Alveo U55c, we limit the data migration in CrHCS to only a single next channel. If additional on-chip memory is available—for instance, on a larger FPGA—the scheduling scope can be extended beyond a single next channel to include two or even three subsequent channels. This broader scheduling window can help fill idle cycles and further reduce the peak PE underutilization in Chasoň, which currently reaches up to 60–65%.

6.2 Performance Improvement

Now that Chasoň significantly reduces the PE underutilization compared to Serpens [67], we analyze how this improvement translates to the actual performance gains in Chasoň relative to the baselines. The performance gains are achieved through the combined effects of CrHCS and our novel architectural support. These two components are interdependent and cannot function effectively without each other, and should therefore be considered together when evaluating the sources of speedup.

6.2.1 ***Over GPU and CPU.*** We compare Chasoň over Nvidia RTX 4090, Nvidia RTX A6000 and Intel Core i9-11980HK over a wide range of matrices. We use 800 matrices from SuiteSparse [7] and SNAP [35] collection with varying density and size as described in Section 5.4. The peak throughput for the 800 matrices is 30.23 *GFLOPS* for Chasoň, 19.83 *GFLOPS* for Nvidia RTX 4090, 44.20 *GFLOPS* for Nvidia RTX A6000 and 23.88 *GFLOPS* for Intel Core i9-11980HK. Figure 14 shows the performance speedup over GPU and CPU baselines. The geometric mean speedup over Nvidia RTX 4090 is approximately 4× with a peak speedup of 20.33×. The geometric mean speedup over Nvidia RTX A6000 is approximately 1.28× with a peak speedup of 11.65×. Due to the highly sparse nature of data, Nvidia RTX 4090 and RTX A6000 fail to effectively use their peak compute capabilities. One of the reasons is the underutilized ALU pipeline in streaming multiprocessors.

The peak speedup over Intel Core i9 is 2.67×. The geometric mean speedup, however, is less than 1. Interestingly, the Intel Core i9 outperforms Nvidia GPUs for SpMV. While GPUs excel at dense, highly parallel workloads like GEMM, they struggle with sparse kernels due to irregular memory access patterns and limited on-chip memory. Additionally, our use of vendor-optimized libraries shows that Intel's MKL handles SpMV more efficiently than Nvidia's cuSparse. Intel MKL uses advanced threading models and effectively maximizes CPU core utilization for parallel computations. Intel MKL also leverages the large on-chip memory available in CPU (24MB in Intel Core i9-11980HK) to optimize the random memory access patterns typically associated with sparse kernels. However, this comes at the expense of increased power consumption. Ideally, we want to harness maximum throughput by consuming the least amount of energy. On average Nvidia RTX 4090 and Nvidia RTX A6000 consume 70W and 65W, respectively, while Intel Core i9 takes 132W of power. Figure 14 shows that the peak energy efficiency gain of Chasoň is 34.72×, 19.48× and 14.61× over Nvidia

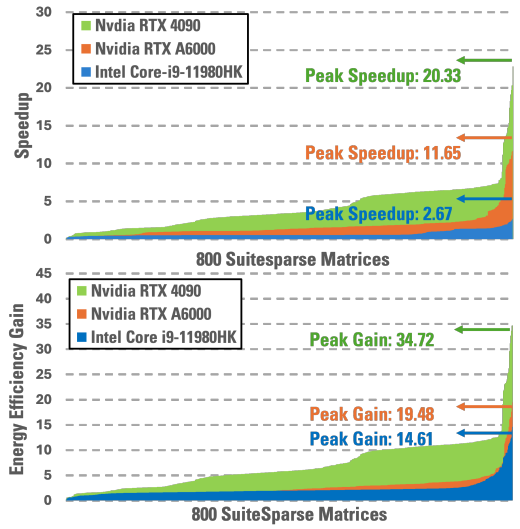

Figure 14: Performance over GPU and CPU baselines (higher is better)– Top: Latency Speedup. Bottom: Energy Efficiency $(GFLOPS/W)$ **Gain.**

RTX 4090, Nvidia RTX A6000, and Intel Core i9. The high energy efficiency of Chasoň over Intel Core i9 and Nvidia RTX A6000 justifies their relatively low-performance speedup gain. It is worth noting that Chasoň consumes less energy than both GPUs and CPUs while delivering better performance. The reduction in power consumption translates to lower operational costs, improved thermal management, and increased system longevity, making it a worthwhile choice for energy-conscious deployments.

6.2.2 ***Over Serpens.*** Figure 15 shows the (latency or throughput) speedup achieved over Serpens[67]. The geometric mean speedup is 6.1× and 4.1× for SuiteSparse [7] and SNAP [35] matrices, respectively, and can go up to 8.4×. The detailed performance numbers are given in Table 3. Both Chasoň and Serpens [67] reach a peak throughput of 30.28 *GFLOPS* and 7.08 *GFLOPS* for SuiteSparse [7] matrices respectively. The peak improvement is 8.4× for reorientation_4 (RE) matrix with throughput approximately equal to only 0.5 *GFLOPS* in Serpens [67]. Similarly, for graph dataset from SNAP [35] matrix collection, Chasoň and Serpens [67] reaches a peak throughput of 27.36 *GFLOPS* and 6.504 *GFLOPS* respectively with 5.84× peak improvement. The Serpens paper reports a peak throughput of 46.43 *GFLOPS*, but does not specify which particular matrix this performance is based on. However, for the 12 matrices listed in the Serpens paper, Chasoň achieves a geometric mean speedup of 1.17×, with 43.27 *GFLOPS* and 41.11 *GFLOPS* peak throughput for Chasoň and Serpens respectively. The speedup is less pronounced in these cases due to RAW dependencies in the migrated data which reduce the opportunity for CrHCS to fully exploit its advantages, as explained in Section 3.3. In our evaluation, we faithfully use the open-source Serpens artifact and run all experiments on the Alveo U55c platform, which differs from the Alveo U280 used in the original Serpens paper. Both Chasoň and Serpens[67] are streaming accelerators and continuously stream data to the PEGs, hence they utilize peak memory bandwidth of $14.37GB/s$ per channel. As a result, Chasoň achieves the same bandwidth efficiency improvement, as illustrated in Figure 15.

Table 3: Detailed performance numbers of Chasoň and Serpens evaluated on 20 matrices from SuiteSparse and SNAP .

ID	Latency (ms)		Throughput (GFLOPS)		Bandwidth Efficiency (GFLOPS/(GB/s))		Imp.	Energy Efficiency (GFLOPS/W)		Imp.
	Chasoň	Serpens	Chasoň	Serpens	Chasoň	Serpens		Chasoň	Serpens	
SuiteSparse Matrices										
DY	0.017	0.125	4.876	0.666	10.599	1.447	7.353	0.125	0.04	3.125
RE	0.017	0.144	4.289	0.505	9.324	1.097	8.471	0.11	0.03	3.667
C5	0.033	0.141	13.569	3.219	29.499	6.998	4.273	0.348	0.194	1.794
MY	0.027	0.116	30.289	7.086	65.847	15.404	4.296	0.777	0.428	1.815
VS	0.037	0.243	7.386	1.116	16.055	2.425	6.568	0.189	0.067	2.821
TS	0.068	0.493	24.433	3.37	53.116	7.327	7.25	0.626	0.204	3.069
LO	0.063	0.392	7.319	1.167	15.912	2.537	6.222	0.188	0.07	2.686
HA	0.069	0.513	3.001	0.401	6.525	0.872	7.435	0.077	0.024	3.208
TR	0.88	6.473	1.971	0.268	4.284	0.582	7.356	0.051	0.016	3.188
CK	0.074	0.288	10.307	2.66	22.407	5.783	3.892	0.264	0.161	1.64
SNAP Matrices										
WI	0.01	0.056	21.898	4.027	47.604	8.754	5.6	0.561	0.243	2.309
EM	0.056	0.223	14.515	3.631	31.553	7.893	3.982	0.372	0.219	1.699
AS	0.043	0.177	6.416	1.563	13.948	3.398	4.116	0.165	0.094	1.755
OR	0.026	0.147	6.059	1.053	13.171	2.289	5.654	0.155	0.064	2.422
WK	0.015	0.067	26.785	5.993	58.229	13.028	4.467	0.687	0.362	1.898
SC	0.101	0.302	19.42	6.504	42.217	14.138	2.99	0.498	0.393	1.267
A7	0.013	0.076	5.158	0.898	11.212	1.953	5.846	0.132	0.054	2.444
CM	0.003	0.018	12.986	2.524	28.231	5.488	6	0.333	0.152	2.191
WB	0.01	0.043	9.797	2.164	21.298	4.705	4.3	0.251	0.131	1.916
RE	0.023	0.119	27.365	5.182	59.489	11.265	5.174	0.702	0.313	2.243

We also measure the energy consumed by Chasoň and Serpens during these tests. The actual power consumption comes out to be approximately $39W$ and $36W$. The increased power consumption by Chasoň can be attributed to the following:

- Lower PE underutilization by the virtue of CrHCS.
- Slightly increased on-chip memory requirements for storing partial outputs of the shared channel.
- Higher achieved frequency of Chasoň ($301MHz$).

To evaluate energy efficiency, we calculate the number of *GFLOPS* each architecture achieves per watt of energy consumed. On average, Serpens is giving 0.16 $GLOPS/W$ while Chasoň gives 0.33 $GFLOPS/W$, that is, 2.03× improvement, making it an energy efficient architecture. The detailed energy efficiency numbers are given in Table 3.

Reasons for Performance Improvement: In Section 3.4, we highlight that a key advantage of CrHCS is its ability to reduce the number of data transfers from HBM to the PEs. As a result, Chasoň sends more non–zero per transfer, minimizing unnecessary HBM accesses. This optimization allows Chasoň to perform the same amount of computation with fewer data transfers compared to Serpens, which directly contributes to its performance speedup. Figure 15 also shows the data transfer reduction by Chasoň. Chasoň transfers significantly less data compared to Serpens. For both SuiteSparse and SNAP matrices, on average, Chasoň transfers approximately 7× less data than Serpens. While both architectures process the same number of non-zero values, Chasoň avoids transferring redundant zeros by leveraging our novel CrHCS scheduling. In contrast, Serpens uses PE-aware non-zero scheduling, which results in additional zero-padding and higher data movement. This reduction in unnecessary data transfer leads to better performance of Chasoň over Serpens.

Chasoň does not necessarily achieve speedup equal to the factor of data transfer reduction. This is due to the additional logic introduced to support CrHCS—specifically, the Reduction Unit, Reorder Unit, and Router. While these units introduce some latency, they are fundamental to enable our enhanced scheduling. For example, the data transfer reduction factor is 6.7× for the $C5$ matrix and 4.4× for the MY matrix, $C5$ does not yield higher speedup. This is

Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari
University of Maryland, College Park
{ubaidb, namjoo, bahar}@umd.edu

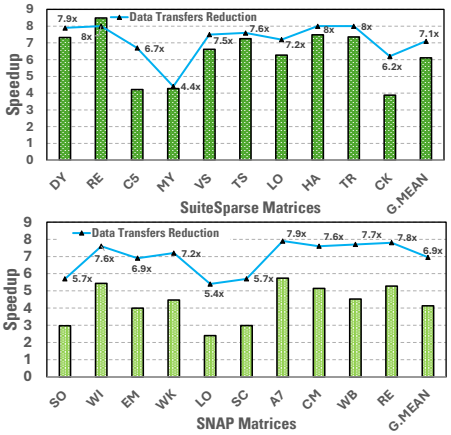

Figure 15: Speedup over Serpens [67] for SuiteSparse [7] and SNAP [35] matrices.

because $C5$ has $23K$ columns compared to $3K$ in MY, resulting in deeper URAMs in the ScUG and a larger number of partial sums to be accumulated in the Reduction Unit. As a result, the latency in C5 increases and that offsets the benefits of data transfer savings. Consequently, MY achieves better overall performance despite a smaller data transfer reduction factor.

7 Discussions

7.1 Overview of Other Related Studies

SpMV Accelerators. The main goal of Chasoň is to improve PE underutilization by reducing the number of stalls in the processing pipeline. Chasoň is an HBM-based SpMV accelerator. However, there are other categories of SpMV accelerators as well that use different techniques to accelerate the SpMV kernel. For instance, [39] optimizes SpMV on FPGAs by reordering non-zero elements to maximize data reuse, reducing the need for additional memory requests by efficiently utilizing already fetched data. There has been an increased focus on novel sparse data compression methods to analyze their impact on accelerating SpMV [1, 54, 61]. Sparstition [65] is an example of incorporating HW/SW optimizations via different partitioning schemes to efficiently parallelize SpMV kernel. PrSpMV [14] is another example. It proposes an SpMV kernel that makes predictions about memory accesses to data regularity and locality to improve prefetching the accuracy of branch predictors in SpMV computational loop. DASP [48] also makes the irregular data patterns in sparse matrices regular to exploit the matrix multiply accumulator units. Multi-stage tree-based decisions are also used for SpMV acceleration by Two-Step [62], MeNDA [13], Spica [58] and [41]. Recent works are also exploring advanced computing architecture to efficiently support SpMV. For example, SpaceA explores processing-in-memory (PIM) architectures and build functionality to order the memory access patterns before sending the non-zero values to actual compute units [77]. With the penetration of machine learning applications in the industry, there has been a resurgence in systolic arrays. Sparse TPU [22] and Flex-TPU [21] reuse a 2D tensor processing unit (TPU) to reduce the memory access overhead and accelerate SpMV. Another systolic-array-based accelerator, Conveyor [32], addresses the challenge of the inherent mismatch between the dense structure of systolic arrays and the unstructured sparsity found in data. To overcome this issue, it introduces chunk propagation for parallelism, PE grouping, and dynamic load balancing.

Other Sparse Algebraic Accelerators. As discussed in Section 2, sparse algebra encompasses different sparse kernels. Researchers have been proposing novel architectures to accelerate different sparse algebraic kernels. For example, there has been extensive work on SpMM acceleration [10, 40, 50, 51, 53, 56, 68, 69, 79, 83]. Instead of using well-known compression formats[33, 60, 64, 73], MatRaptor [69] proposes using a new format called C2SR to accelerate sparse-matrix sparse multiplication (SpGEMM). Tensaurus [70] is another HBM-based vectorized accelerator that supports SpMV, SpMM, SpTTMc, and SpMTTKRP kernels by proposing a new compression format for the sparse matrix. Researchers are also inducing sparsity in different fields to take leverage of existing sparse solutions. [9, 26, 36, 38, 46, 75, 82] are some of the DNN accelerators specifically designed to cater sparse kernels. For example, SparTen [16] proposes a bitmask representation of non-zero values and corresponding architecture to support sparsity in CNNs.

7.2 Chasoň for SpMM

Chasoň can also be extended to other sparse algebraic kernels, for example, SpMM. Mathematically, SpMM is given as:

$$C = \alpha AB + \beta C \tag{8}$$

where A is the input sparse matrix, B is the dense matrix, and C is the output matrix. Similar to the prior OoO HBM-based SpMM accelerator [68], Chasoň enables SpMM by utilizing 8 HBM channels for the input matrices A and C, while allocating 4 channels for matrix B. The output is written back through the 8 channels designated for the C matrix. In total, Chasoň utilizes 29 HBM channels. Because of the dense matrix multiplication, Chasoň requires more on-chip memory to buffer dense matrix B. The size of `URAM_sh` in an ScUG will be increased to hold partial sums corresponding to dense matrix B values. The Reduction Unit and Re-order Unit will be configured trivially to support multiple partial outputs per `URAM_sh`.

8 Conclusions

This paper analyzed state-of-the-art PE-aware OoO non-zero scheduling scheme and showed that it results in highly underutilized PEs. The reason being its limited intra-channel scheduling to fill the stalls in the execution pipeline. We introduced our novel OoO scheduling scheme, CrHCS, that extended the intra-channel scheduling to inter-channel scheduling by allowing data migration across HBM channels. We also introduced our novel HBM-based streaming architecture, called Chasoň to support CrHCS. Chasoň used on-chip memory and an adder tree to store and reduce the partial sums. It also uses a rearrange logic to ensure functionally correct execution of the underlying kernel. We implemented Chasoň on AMD Xilinx Alveo U55c and evaluated it on a diverse set of SuiteSparse [7] and SNAP [35] matrices. Extensive experiments showed reduced PE underutilization that resulted in improved performance,

memory and energy efficiency over state-of-the-art SpMV accelerator Serpens [67], Nvidia RTX 4090, Nvidia RTX A6000 (cuSparse), and Intel Core i9-11980HK (MKL).

References

[1] Bahar Asgari, Ramyad Hadidi, Joshua Dierberger, Charlotte Steinichen, Amaan Marfatia, and Hyesoon Kim. 2021. Copernicus: Characterizing the performance implications of compression formats used in sparse workloads. In IISWC. IEEE, 1–12.

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

[3] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 1–11. https://doi.org/10.1145/1654059.1654078

[4] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In 2012 Design, Automation Test in Europe Conference Exhibition (DATE). 33–38. https://doi.org/10.1109/DATE.2012.6176428

[5] Thomas Chen, Jacob Botimer, Teyuh Chou, and Zhengya Zhang. 2020. A 1.87-mm 2 56.9-GOPS accelerator for solving partial differential equations. IEEE Journal of Solid-State Circuits 55, 6 (2020), 1709–1718.

[6] Xi Chen, Weike Pan, James T. Kwok, and Jaime G. Carbonell. 2009. Accelerated Gradient Method for Multi-task Sparse Learning Problem. In 2009 Ninth IEEE International Conference on Data Mining. 746–751. https://doi.org/10.1109/ICDM.2009.128

[7] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) 38, 1 (2011), 1.

[8] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr. Knight, and Andre DeHon. 2006. GraphStep: A System Architecture for Sparse-Graph Algorithms. In 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. 143–151. https://doi.org/10.1109/FCCM.2006.45

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

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

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

[12] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

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

[14] Gelin Fu, Tian Xia, Shaoru Qu, Zhongpei Luo, Shuyu Li, Pengyu Cheng, Runfan Guo, Yitong Ding, and Pengju Ren. 2023. PrSpMV: An Effecient Predictable Kernel for SpMV. In 2023 IEEE International Conference on Computing Design. IEEE.

[15] Sukhpal Singh Gill, Huaming Wu, Panos Patros, Carlo Ottaviani, Priyansh Arora, Victor Casamayor Pujol, David Haunschild, Ajith Kumar Parlikad, Oktay Cetinkaya, Hanan Lutfiyya, Vlado Stankovski, Ruidong Li, Yuemin Ding, Junaid Qadir, Ajith Abraham, Soumya K. Ghosh, Houbing Herbert Song, Rizos Sakellariou, Omer Rana, Joel J.P.C. Rodrigues, Salil S. Kanhere, Schahram Dustdar, Steve Uhlig, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2024. Modern computing: Vision and challenges. Telematics and Informatics Reports 13 (2024), 100116. https://doi.org/10.1016/j.teler.2024.100116

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

[17] 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 Transactions on Reconfigurable Technology and Systems 16, 4 (Dec. 2023), 1–31. https://doi.org/10.1145/3609335

[18] 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

[19] Ning Guo, Yipeng Huang, Tao Mai, Sharvil Patil, Chi Cao, Mingoo Seok, Simha Sethumadhavan, and Yannis Tsividis. 2016. Energy-efficient hybrid analog/digital approximate computation in continuous time. IEEE Journal of Solid-State Circuits 51, 7 (2016), 1514–1524.

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

[21] Xin He, Kuan-Yu Chen, Siying Feng, Hun-Seok Kim, David Blaauw, Ronald Dreslinski, and Trevor Mudge. 2023. Squaring the Circle: Executing Sparse Matrix Computations on FlexTPU—A TPU-Like Processor. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (Chicago, Illinois) (PACT '22). Association for Computing Machinery, New York, NY, USA, 148–159. https://doi.org/10.1145/3559009.3569665

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

[23] Zifan He, Linghao Song, Robert F. Lucas, and Jason Cong. 2024. LevelST: Stream-based Accelerator for Sparse Triangular Solver. In Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2024, Monterey, CA, USA, March 3-5, 2024, Zhiru Zhang and Andrew Putnam (Eds.). ACM, 67–77. https://doi.org/10.1145/3626202.3637568

[24] Kartik Hedge, Hadi Asghari, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Flectcher. 2019. ExTensor: An Accelerator for Sparse Tensor ALgebra. In 2019 International Symposium on Microarchitecture (MICRO-52). IEEE/ACM, 319–333.

[25] Vasant G. Honavar, Mark D. Hill, and Katherine Yelick. 2016. Accelerating Science: A Computing Research Agenda. arXiv:1604.02006 [cs.CY] https://arxiv.org/abs/1604.02006

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

[27] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Donald Yeung, and Bruce Jacob. 2019. Design for ReRAM-based main-memory architectures. In Proceedings of the International Symposium on Memory Systems (Washington, District of Columbia, USA) (MEMSYS '19). Association for Computing Machinery, New York, NY, USA, 342–350. https://doi.org/10.1145/3357526.3357561

[28] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In 2012 Symposium on VLSI Technology (VLSIT). 87–88. https://doi.org/10.1109/VLSIT.2012.6242474

[29] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In 2017 IEEE International Memory Workshop (IMW). 1–4. https://doi.org/10.1109/IMW.2017.7939084

[30] Nachiket Kapre. 2015. Custom FPGA-based soft-processors for sparse graph acceleration. In 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP). 9–16. https://doi.org/10.1109/ASAP.2015.7245698

[31] Joonyoung Kim and Younsu Kim. 2014. HBM: Memory solution for bandwidth-hungry processors. In 2014 IEEE Hot Chips 26 Symposium (HCS). 1–24. https://doi.org/10.1109/HOTCHIPS.2014.7478812

[32] Seongwook Kim, Gwangeun Byeon, Sihyung Kim, Hyungjin Kim, and Seokin Hong. 2023. Conveyor: Towards Asynchronous Dataflow in Systolic Array to Exploit Unstructured Sparsity. In 2023 IEEE 41st International Conference on Computer Design (ICCD). IEEE, 423–431.

[33] David R Kincaid, Thomas C Oppe, and David M Young. 1989. ITPACKV 2D user's guide. Technical Report. Texas Univ., Austin, TX (USA). Center for Numerical Analysis.

[34] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O

test methods using 29nm process and TSV. In 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). 432–433. https://doi.org/10.1109/ISSCC.2014.6757501

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

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

[37] Jiajun Li, Yuxuan Zhang, Hao Zheng, and Ke Wang. 2023. FDMAX: An elastic accelerator architecture for solving partial differential equations. In Proceedings of the 50th Annual International Symposium on Computer Architecture. 1–12.

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

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

[40] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. Matrix 108 (2023), 1012.

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

[42] Yajing Liu, Ruiqi Chen, Shuyang Li, Jing Yang, Shun Li, and Bruno da Silva. 2024. FPGA-Based Sparse Matrix Multiplication Accelerators: From State-of-the-Art to Future Opportunities. ACM Trans. Reconfigurable Technol. Syst., Article 59 (Nov. 2024), 37 pages.

[43] Zirui Liu, Chen Shengyuan, Kaixiong Zhou, Daochen Zha, Xiao Huang, and Xia Hu. 2023. RSC: Accelerate Graph Neural Networks Training via Randomized Sparse Computations. In Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202), Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 21951–21968. https://proceedings.mlr.press/v202/liu23ad.html

[44] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08). IEEE Computer Society, USA, 453–464.

[45] 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

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

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

[48] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 1–14.

[49] Junjie Mu, Chengshuo Yu, Tony Tae-Hyoung Kim, and Bongjin Kim. 2022. A scalable bit-serial computing hardware accelerator for solving 2D/3D partial differential equations using finite difference method. In ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC). IEEE, 353–356.

[50] Francisco Muñoz-Martínez, Raveesh Garg, José L Abellán, Michael Pellauer, Manuel E Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. arXiv preprint arXiv:2301.10852 (2023).

[51] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 90–106.

[52] NVIDIA. [n. d.]. NVIDIA cuSPARSE library samples. https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuSPARSE/spmv_csr [Accessed: June-24th-2024].

[53] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok kim, Davic Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In 2018 IEEE Internation Sumposium on High Performance Computer Architecture. IEEE, 724–736.

[54] Alberto Parravicini, Francesco Sgherzi, and Marco D Santambrogio. 2021. A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA. In Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC). 378–383.

[55] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In 2011 IEEE Hot Chips 23 Symposium (HCS). 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[56] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishana. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In 2020 IEEE Internation Sumposium on High Performance Computer Architecture. IEEE, 58–70.

[57] 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. https://doi.org/10.1145/3626202.3637557

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

[59] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. 2013. Efficient trajectory optimization using a sparse model. In 2013 European Conference on Mobile Robots. 138–143. https://doi.org/10.1109/ECMR.2013.6698833

[60] Yousef Saad. 2003. Iterative methods for sparse linear systems. Vol. 82. siam.

[61] Fazle Sadi, Larry Pileggi, and Franz Franchetti. 2017. Algorithm and hardware co-optimized solution for large SpMV problems. In 2017 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 1–7.

[62] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 347–358.

[63] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling low-overhead and scalable near-data pattern matching acceleration. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. 311–323.

[64] SciPy. 2020. List-of-list sparse matrix. [Online; accessed April-2020].

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

[66] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In 2020 30th International Conference on Field-Programmable Logic and Applications (FPL). IEEE, 9–17.

[67] 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

[68] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Virtual Event, USA) (FPGA '22). Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3490422.3502357

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

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

[71] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In Proceedings of the 26th ACM International Conference on Supercomputing (San Servolo Island, Venice, Italy) (ICS '12). Association for Computing Machinery, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

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

[73] Richard W Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In International Conference on High Performance Computing and Communications. Springer, 807–816.

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

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

[76] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (Reno, Nevada) (SC '07). Association for Computing Machinery, New York, NY, USA, Article 38, 12 pages. https://doi.org/10.1145/1362622.1362674

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

[78] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[79] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). 931–945. https://doi.org/10.1109/ISCA59077.2024.00072

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

[81] 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. https://doi.org/10.23919/DATE58400.2024.10546672

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

[83] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. arXiv preprint arXiv:2002.08947 (2020).