Near-Memory Parallel Indexing and Coalescing: Enabling Highly Efficient Indirect Access for SpMV

Chi Zhang*, Paul Scheffler*, Thomas Benz*, Matteo Perotti*, Luca Benini*†
* *Integrated Systems Laboratory, ETH Zurich*, Switzerland
† *Department of Electrical, Electronic, and Information Engineering, University of Bologna*, Italy
{chizhang,paulsc,tbenz,mperotti,lbenini}@iis.ee.ethz.ch

***Abstract*—Sparse matrix-vector multiplication (SpMV) is central to numerous data-intensive applications, but requires streaming indirect memory accesses that severely degrade both processing and memory throughput in state-of-the-art architectures. Near-memory hardware units, decoupling indirect streams from processing elements, partially alleviate the bottleneck, but rely on low DRAM access granularity, which is highly inefficient for modern DRAM standards like HBM and LPDDR. To fully address the end-to-end challenge, we propose a low-overhead data coalescer combined with a near-memory indirect streaming unit for AXI-Pack, an extension to the widespread AXI4 protocol packing narrow irregular stream elements onto wide memory buses. Our combined solution leverages the memory-level parallelism and coalescence of streaming indirect accesses in irregular applications like SpMV to maximize the performance and bandwidth efficiency attained on wide memory interfaces. Our solution delivers an average speedup of 8x in effective indirect access, often reaching the full memory bandwidth. As a result, we achieve an average end-to-end speedup on SpMV of 3x. Moreover, our approach demonstrates remarkable on-chip efficiency, requiring merely 27kB of on-chip storage and a very compact implementation area of 0.2-0.3mm² in a 12nm node.**

***Index Terms*—Sparse Computing, Memory Systems, Coalescing, DRAM, HBM, AXI4**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is pivotal in data-intensive application domains including machine learning [1], fluid dynamics [2], and graph analytics [3]. Sparse matrices in these domains often exhibit vast scales and low nonzero densities. To reduce memory footprint, they are stored in compressed formats like compressed sparse row (CSR) or sliced ELLPACK (SELL) [4].

Efficient SpMV poses a challenge to general-purpose architectures like central processing units (CPUs), graphics processing units (GPUs), and vector processors. The challenge arises primarily from the indirect addressing and irregular, noncontiguous access of vector elements. Indirect addressing complicates program access flow as addresses depend on the values of indices in memory. Irregular memory access patterns disrupt the efficient utilization of the memory hierarchy in contemporary processors, resulting in low bandwidth efficiency, cache trashing, and elevated access latencies.

Modern processors circumvent these issues by increasing core counts to saturate bandwidth, improving non-blocking cache miss handling, and increasing cache sizes. While these measures improve SpMV performance, they also incur significant additional on-chip area and further aggravate the cache pollution inherent to SpMV's low data reuse [5].

To address these inefficiencies, recent hardware proposals handle indirect accesses *near memory* [6]–[9]: as SpMV's indirection can be fully decoupled from computation, indirected elements may be streamed directly from the main memory without the need for large conventional caches. This insight has led to memory controllers with gather-scatter functionality [6], [7], near-memory data layout transformers [8], and pattern-aware caches [9]. However, many of these solutions rely on *narrow* memory channels with *low access granularity* (≤ 64 b) commonly provided by older DRAM standards, which enable efficient access to individual vector values in SpMV. In contrast, contemporary DRAM interfaces usually have larger access granularities (~ 512 b); modern standards such as high bandwidth memory (HBM) and LPDDR do not provide narrow channels, instead prioritizing high bandwidth or low power. To achieve truly portable efficiency in SpMV, narrow elements must be efficiently accessed through the wide interfaces of these modern DRAM subsystems.

AXI-Pack [6] is a recently proposed extension to Arm's widespread Advanced eXtensible Interface 4 (AXI4) [10] protocol which maximizes *on-chip* bus efficiency on SpMV and other irregular workloads. It packs multiple narrow elements onto a wide on-chip bus and enables *bursts* of strided and indirect accesses, providing efficient on-chip transport of narrow data. Unfortunately, it was only demonstrated with low-granularity (32 b) on-chip scratchpads and does not address or accommodate the high granularity of modern off-chip DRAM.

In this work, we present an low-overhead access coalescer for AXI-Pack enabling highly efficient indirect accesses to modern, high-granularity DRAM interfaces. To this end, we leverage the memory-level parallelism and coalescence of decoupled streaming indirect accesses, enabling cache-less data reuse and significantly improving bandwidth efficiency. With its aid, SpMV can be efficiently handled on general-purpose processors without the need for extensive on-chip resources. Our contributions are as follows:

- We extend the existing AXI-Pack adapter's indirect stream unit with a low-overhead data coalescer to accommodate a 512 b granularity DRAM controller. The new adapter design leverages both the memory-level parallelism and coalescence of indirect streams.
- Connecting our adapter to a standard 512 b granularity HBM channel controller, we achieve an 8× increase in

979-8-3503-4859-0/DATE24/© 2024 EDAA

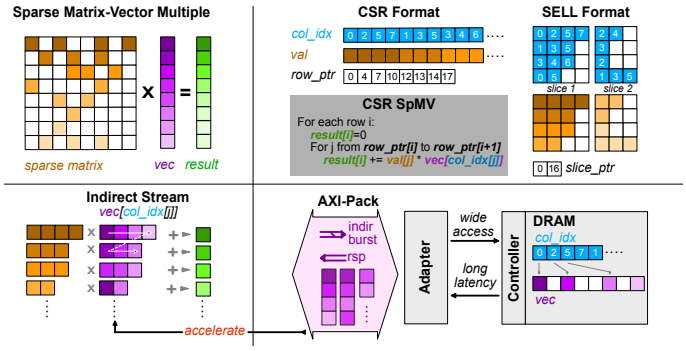


Figure 1. (TL) shows a graphical representation of SpMV, a sparse matrix multiplied by a dense vector equals a dense vector. (TR) represents the two file formats used in this work: CSR and SELL. (BL) is the indirect stream of CSR SpMV accelerated by (BR) our coalescing *adapter* using AXI-PACK.

effective indirect access bandwidth for real-world sparse matrices.

- We integrate our DRAM coalescer into an open-source RISC-V vector processor system, where we achieve an average SpMV speedup of 10×compared to a baseline with a 1 MiB last-level cache (LLC), and 3×over an adapter sans coalescer.
- Our solution shows 1.4×and 2.6×superior on-chip efficiency, while retaining 1×and 0.9×SpMV performance efficiency compared to state-of-the-art vector processors.

II. ARCHITECTURE

We will first present our *indirect stream unit* translating AXI-PACK indirect bursts into wide DRAM requests in Section II-A. We will then detail the contained *request coalescer* in Section II-B. Finally, we will describe the *integration* of our hardware into a vector processor system in Section II-C.

A. AXI-PACK *Indirect Stream Unit*

Figure 2a depicts the AXI-PACK adapter containing our *indirect stream unit*, which is based on the previously proposed AXI-PACK indirect read converter [6]. It translates AXI-PACK indirect burst requests into bandwidth-efficient sequences of wide DRAM accesses in a non-blocking, index-parallel manner. Among its five components, the *element request generator* and *element packer* are reused from the original indirect read converter without modification. The *index fetcher* was modified and the *index splitter* and *request coalescer* were newly added to efficiently interface wide DRAM controllers downstream.

Upon receiving an indirect burst request from an upstream AXI-PACK manager, the *index fetcher* determines the index stream's start address and length to issue efficient wide DRAM reads retreiving the indices. It monitors the usage of downstream index queues to prevent their overflow. For every received wide block of indices, the *index splitter* splits it into N parallel segments, which are subsequently pushed into the aforementioned N parallel index queues. The *element request generator* then produces N parallel narrow element requests by extracting indices from the index queues and adding them to the requested element base address.

The generated N parallel narrow element requests are passed to the *request coalescer* which forms wide DRAM accesses, reducing access redundancy and enabling data reuse in the process. We explain the inner workings of the coalescer in Section II-B. Ultimately, it returns the N parallel requested elements in the order of the original narrow requests. To comply with the AXI-PACK protocol, the retrieved narrow elements are densely packed onto the upstream wide bus by the *element packer*, maximizing on-chip bus utilization.

B. Request Coalescer

Figure 2b shows the architecture of the request coalescer. The *upsizer* receives and pushes the N incoming parallel narrow requests into W decoupled request queues. Both N and W must be powers of two and $W \geq N$. For each narrow request port, requests are evenly distributed among W/N request queues, collecting *windows* of W requests. A *regulater* presents a complete window of the W oldest requests to the *request watcher*. In case there aren't enough requests to form a complete window within a time limit, it forwards the remaining requests.

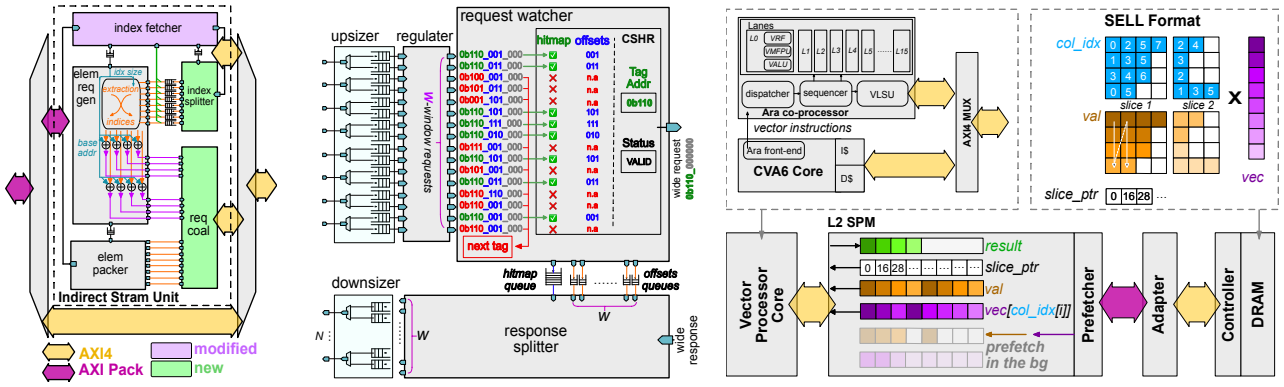
The *request watcher* identifies requests in the current window accessing the same wide DRAM block, together forming a request *warp*. A *coalescer status holding register (CSHR)* stores the following information on the current request warp:

- **Tag**: The address of the wide block in DRAM for which requests are currently being coalesced. If a narrow request in the current window can be served from this block, we call it a *hit* request; otherwise, it is a *miss* request.
- **Status**: 0 (*IDLE*) denotes that a coalescing is in progress and 1 (*VALID*) that a block has been coalesced.
- **Hitmap**: A W-bit array denoting which window requests have been merged into the current request warp.
- **Offsets**: The address offsets for each coalesced request.

The request watcher only contains one active CSHR. In each cycle, the request watcher checks all valid requests in the window in parallel to see if they can hit the current CSHR. It then accepts hit requests, updates the CSHR's *Status*, *Hitmap*, and *Offsets* information accordingly, and tells the regulator to invalidate the corresponding window entries. With pending miss requests, the request watcher issues the wide request of the current CSHR downstream, updating the *Hitmap* and *Offsets* information to *metadata queues*. Meanwhile, miss requests are used to determine the next CSHR's *Tag Address*. Once all requests in the window have been coalesced, the CSHR tells the regulator to forward the next window. Additionally, a watchdog will issue the current CSHR's wide request should no new request window arrive within a configurable time limit.

The *metadata queues* consist of a deep first in, first out (FIFO) holding the *Hitmap* and W shallow FIFOs for the *Offsets*. Their behaviors differ slightly: every time the request watcher issues a wide access, it loads the entire *Hitmap* array into the deep queue while only pushing *offsets* of the hit window entries to the corresponding shallow FIFOs.

In the return path, a response splitter retrieves the parallel narrow elements from the wide downstream data response. Upon receiving a response, it checks the *Hitmap* queue to identify the window entries receiving the element. Subsequently, it



(a) (b) (c)

Figure 2. (a) shows the extended AXI-PACK adapter unit; modified blocks are marked in green and light purple. (b) presents a detailed architectural view of the *request coalescer (req coal)*. (c) outlines the integration of our extended AXI-PACK in a system based around *CVA6* [11] and *ARA* [12].

pops the related *Offsets* queues to retrieve the word offsets. It then extracts the elements of the window entries in parallel and forwards them to the *W element queues*. Finally, a downsizer mapps the *W element queues* onto the *N* output ports. To summarize, the downsized is structurally similar to the upsizer performing the inverse function while enforcing the ordering.

C. Integration into a Vector Processing System

To evaluate its performance benefits, we integrate our AXI-PACK adapter into a RISC-V vector processor system as shown in Figure 2c. Ara [12] is an open-source vector coprocessor supporting version *1.0* of the RISC-V vector extension. Together with the 64 b Linux-capable CVA6 [11] core, it forms a vector processor core (VPC) wherein CVA6 offloads vector instructions to Ara. Both share a common AXI4-based memory interface.

The VPC's 512 b-wide AXI4 interface is connected through a prefetching-enabled level two (L2) scratch pad memory (SPM) to DRAM main memory. The L2's *prefetcher*'s manager interface implements a 512 b-wide AXI-PACK bus connected to main memory through our coalescing-enhanced AXI-PACK *Adapter*. This *prefetcher* implements both contiguous and indirect prefetching through AXI-PACK requests. This unit is configured through CVA6 and supports up to two outstanding prefetching requests.

We execute tiled SpMV workloads on the VPC on data stores in DRAM using the SELL format. Six equally-sized arrays are allocated in the L2 SPM: one for the *slice pointers* and the *results*, two for holding the *nonzero elements* and two for the *indexed vectors*. Initially, the first data tile, consisting of the slice pointers, nonzero elements, and indexed vectors, is prefetched into L2 through AXI-PACK interface. The VPC then prefetches the subsequent data tiles during the computation of the first tile. Thanks to the indirect accesses of AXI-PACK, the VPC mainly executes vector-multiply-add (VMAC) operations. Slice pointers are consumed more slowly; thus, it is not required to prefetch them during the VPC's computation in every iteration. The VPC interrupts execution should the slice pointer array deplete or if the result array is full. The VPC then signals the prefetcher to refresh the stored data in L2 SPM.

III. Evaluation Methodology

Table I
ADAPTER AND VECTOR PROCESSOR SYSTEM PARAMETERS

Model	Parameter
AXI-PACK Adapter	Queue depth = 256(*index*), 2(*up/downsizer*), 128(*hitmap*), 2048/*W*(*offsets*) On-chip storage = 27KB (*W*=256)
Vector Processor System	16 lanes, 1GHz, 384KB L2
DRAM and Controller	One HBM2 chan, 1GHz, 32GB/s (ideal) Schedule policy: open adaptive, FR-FCFS [13]

We built our AXI-PACK adapter and vector processor system using register transfer level (RTL) models, except for the DRAM controller and models, which are modeled by *DRAMSys* [13]. We compiled DRAMSys into a dynamically linkable library and used Siemens' Questa advanced simulator to connect it, enabling co-simulation of our RTL model with DRAM models in a cycle-accurate manner. All our model parameters are listed in the Table I. The adapter also has three variant types for comparison: without a coalescer (*MLPnc*), with an *x-window* parallel coalescer (*MLPx*), and with an *x-window* coalescer but operating sequentially (*SEQx*). The latter configuration is achieved by serializing the parallel element requests and reducing the input ports of the coalescer to one. For indirect stream and SpMV benchmarking, we selected twenty real-world sparse matrices from the *SuiteSparse* [14] collection and *HPCG* benchmark [2], with columns ranging from 1.4k to 6.8M and nonzeros from 23k to 37M. When converting these matrices to CSR or SELL formats, we utilized 32 b indices and 64 b for nonzeros and metadata, with 32 rows per slice in SELL format. We also introduced a baseline for SpMV comparison on vector processor systems. This baseline uses a 1 MiB LLC between the vector processor and memory controller, both connected by a 512b-wide AXI4 bus. Since the baseline lacks an indirect prefetching function, it executes the naive SpMV code with coupled indirect access and arithmetic operation.

IV. Experimental Results

A. Indirect Stream Analysis

We first evaluate our adapter's ability to stream indirect accesses for the whole SpMV dataflow. For testing, an ideal requestor issued continuous AXI-Pack indirect read requests from upstream, and our matrices, prepared in either SELL or CSR format, were preloaded into the HBM model.

As depicted in Figure 3, without coalescence, our design's indirect stream bandwidth averaged just 2.9 GB/s out of the possible 32 GB/s, a limitation stemming from the under-utilization of wide accesses. However, introducing coalescence drastically boosted this bandwidth, especially as the coalescence window size increased. Impressively, a 256-window parallel coalescer amplified the indirect stream bandwidth by factors of 8.4× and 8.6× for SELL and CSR formats, respectively. This enhancement enabled twelve of the twenty matrices to achieve an indirect access bandwidth surpassing 70 % of the full memory bandwidth. Some CSR formats even exceeded this limit, showcasing the 256-window parallel coalescer's prowess in harnessing data reuse. In contrast, a 256-window sequential coalescer cannot enhance performance as significantly as the parallel coalescer. Although it can still speed up the indirect stream by an average of 2.9×compared to a non-coalescer design, the indirect stream bandwidth is capped under 8 GB/s, averaging a 3.0×slowdown compared to a parallel coalescer with the same window size.

To further comprehend the interplay of memory-level parallelism and coalescence, we delved deeper with six representative matrices (SELL format). Our focus was on the downstream bandwidth utilization and the adapter's coalesce rate. The bandwidth was categorized into three segments: fetching elements, fetching indices, and the loss from the ideal HBM channel bandwidth. We defined the coalesce rate as the ratio of effective indirect access elements to the data amount requested by the coalescer from downstream.

Figure 4 sheds light on several nuances. Without a coalescer, each narrow element request led to one wide access, monopolizing a large chunk of the downstream bandwidth for element fetching. This dominance compromised bandwidth reserved for fetching indices, while the effective indirect stream bandwidth remains low. By contrast, the parallel coalescer's deeper window boosted the coalesce rate. It coalesced more narrow requests under a larger window size while simultaneously reducing wide accesses for elements. This in turn allocated more downstream bandwidth for index fetching, boosting overall throughput. For instance, the matrix *af-shell10* with the 256-window parallel coalescer, showcased an indices fetching bandwidth of 13.2 GB/s. This suggests our adapter was concurrently generating and coalescing *3.3* requests per cycle. Conversely, while the sequential 256-window coalescer might reach the same coalesce rate, the sequential processing becomes a significant bottleneck. One request per cycle limits the indices fetching bandwidth to around 4 GB/s and reduces the bandwidth for element fetching, leading to a relatively low HBM bandwidth utilization.

From our observations, memory-level parallelism and coalescence emerge as complementary forces for indirect stream:

1) Without coalescence, the bandwidth dedicated to index fetching is overshadowed by that for element fetching. Thus, having the hardware to generate parallel element requests from indices becomes redundant.
2) Absent memory-level parallelism, a sequential process can't capitalize on coalescence's potential, culminating in limited bandwidth utilization and reduced effective indirect access bandwidth.

In conclusion, to maximize indirect access bandwidth, a symbiotic approach, harnessing both memory-level parallelism and coalescence, is indispensable.

B. SpMV Performance

To assess the impact of our adapter on SpMV performance, we benchmarked SpMV on four RISC-V vector processor systems. These include three AXI-Pack enabled vector processor systems (as described in Section II-C) with configurations of no coalescer, and parallel coalescers with windows of 64 and 256 (labeled *pack0*, *pack64*, *pack256*). Additionally, we used one *base* system detailed in the Section III.

In Figure 5a, we show their SpMV speedup compared to the *base*, normalized SpMV runtime, and the time spent on indirect accesses. The latter is counted from the prefetcher as the time to transfer the indirect stream in the *pack* system. However, for the *base* system, since it doesn't have a prefetcher, we count the indirect access time as the time the Vector-Load-Store-Unit spends on indices fetching and the gather operation.

The *pack0* system demonstrates an average 2.7×speedup compared to the *base* system. We observed that even a naive prefetching scheme is better than using a conventional cache for SpMV to hide the access latency from DRAM. It saves a significant amount of runtime in tasks like accessing non-zero elements, where the cache suffers from its lack of data reuse nature while the prefetcher leverages long data stream transfers in the background of SpMV arithmetic computation. However, it doesn't significantly reduce the time for indirect accesses since its AXI-Pack adapter isn't equipped with a coalescer, and the indirect stream bandwidth remains low.

In the *pack64* and *pack256* systems, we noticed a substantial improvement in SpMV. The increase of the coalescer window significantly shrinks the time for indirect access, which in turn reduces the runtime of SpMV. The *pack256* system shows an average 3×speedup compared to the *pack0* system and a 10×speedup compared to the *base* system.

Figure 5b displays SpMV's off-chip traffic and HBM bandwidth utilization for all these systems compared to the ideal, which is no redundant off-chip traffic and full bandwidth utilization. Although the *base* system demonstrated low off-chip traffic overhead due to a large LLC, its bandwidth utilization, as low as 5.9 %, indicates its poor performance. While the *pack0* system has the best average bandwidth utilization of 65.8 %, it exhibits an average 5.6 ×off-chip traffic compared to the ideal case, significantly increasing the energy waste on off-chip data movement while the performance is suboptimal. With the help of increasing the coalescer window size to 256, we largely

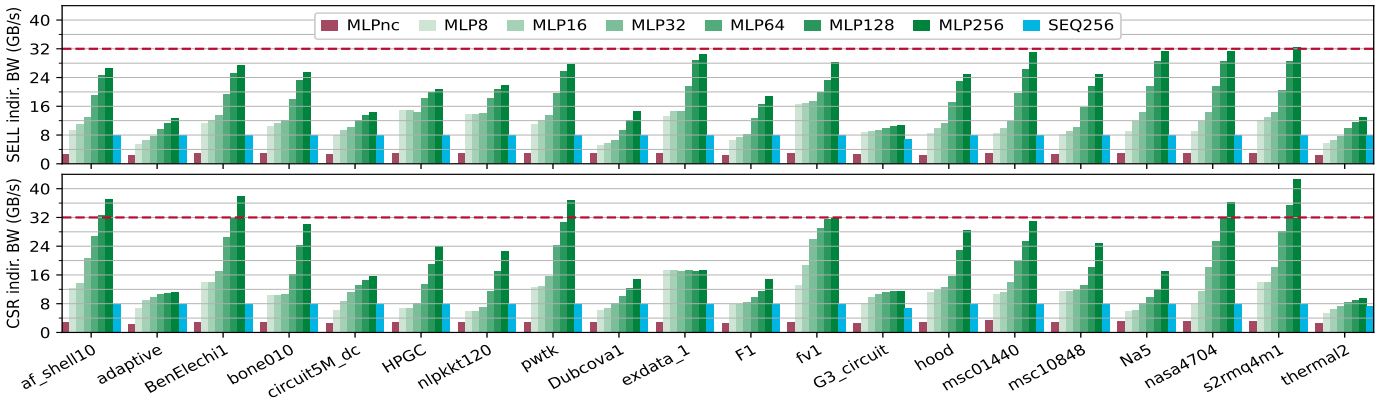


Figure 3. indirect stream bandwidth

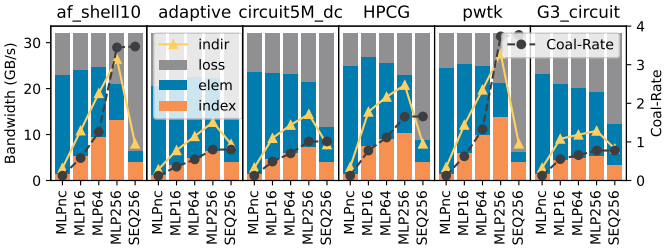
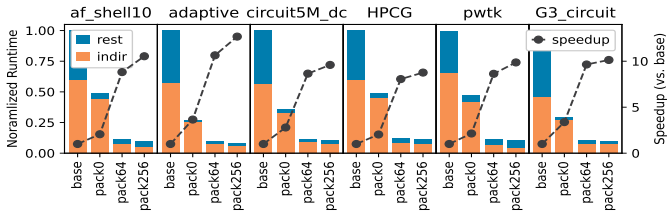
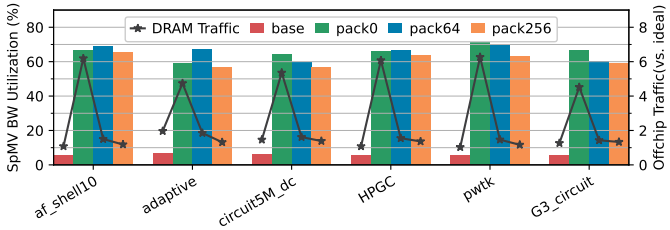


Figure 4. bandwidth breakdown and coalesce rate



(a) Ara SpMV performance



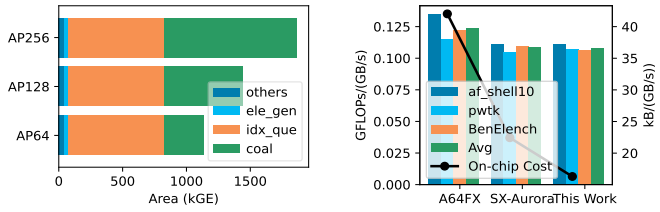
(b) SpMV off-chip traffic and bandwidth utilization

Figure 5. SpMV characterization

reduced the off-chip traffic to only 29 % more than the ideal, while maintaining an average bandwidth utilization of 61 %. Compared to the *base* system with a 1 MiB LLC, our 256-window coalescer even shows 2 % less in off-chip traffic on average.

C. Area And On-chip Efficiency

We implemented our AXI-PACK adapter with Synopsys' *Fusion Compiler* for GlobalFoundries' 12 nm FinFet technology at



(a) AXI-PACK adapter breakdown (b) SpMV efficiency comparison

Figure 6. Area And On-chip Efficiency

1 GHz, worst-case conditions. We mapped the *index queues* of the indirect stream unit and the *hitmap queue* in the coalescer to dual-port static random-access memory (SRAM) macros, while other queues were mapped to standard cells. The logic area breakdown of the AXI-PACK adapter is shown in Figure 6a. The index queues take a large portion of the area in the adapter design, up to 754 kGE. The area of the coalescer increases linearly with the coalesce window, resulting in 307, 617, and 1035 kGE for 64, 128, and 256 windows, respectively. We then successfully implemented our 64, 128, 256 coalesce window adapter in 0.19 , 0.26 , 0.34 mm^2 design area with 60.5 , 56.5 , 56.4 % standard cell utilization, respectively.

Our adapter's minimal on-chip storage of 27 kB and compact design highlights its potential for on-chip efficient general-purpose processors handling SpMV. Using a moderately-sized L2 SPM and the AXI-PACK indirect stream unit, our vector processor approach proves more on-chip efficient than extensive and dedicated multi-level memory system hierarchies. Figure 6b compares our method with leading HBM-based vector processors, *SX-Aurora* [15] and *A64FX* [16], in terms of on-chip storage per memory bandwidth and SpMV performance per memory bandwidth. To ensure fairness, on-chip storage includes all memory components (register file, L1, L2 caches, and LLC). Memory bandwidth, evaluated by the STREAM benchmark's copy test, defines the peak main memory bandwidth. SpMV performance uses a consistent set of sparse matrices in SELL formats, with SXAurora and A64FX results sourced from corresponding references. Our solution offers up to 1.4 ×and 2.6 ×better on-chip efficiency and retains 1 ×and

0.9 ×SpMV performance efficiency when compared to *SX-Aurora* and *A64FX*, respectively.

V. Related Work

Hardware for Memory Coalescing: Memory coalescing, a mechanism that optimizes memory accesses by consolidating multiple requests into fewer, broader memory transactions, has seen various implementations. In GPUs, a coalescer near the L1 cache manages warp accesses, enhancing bandwidth efficiency [17]. Caches also serve as inherent hardware coalescers by reusing cache lines. A distinct approach involves dynamic memory coalescing for hybrid memory cube (HMC) accesses [18], which are processed sequentially. Unlike these, our method integrates a hardware coalescer with near-memory indirect streams, reducing core-side index processing overhead and offering a more area-efficient parallel coalescing scheme.

Near-memory Extensions: To accelerate indirect access for SpMV, many studies have proposed handling it near the main memory with specific hardware units [3], [6]–[9], [19], [20]. Apart from works relied on narrow channels, as mentioned in Section I, some recent works have shifted to broader interfaces. For example, the *Stream Compact Unit (SCU)* [19] compacts sparse vector elements into an array in consecutive memory using an interface with cache-line granularity of 512 b. While they also explore the coalescence of vector elements, these methods only perform sequential coalescing within a 32-element window, achieving only 20 to 40 % memory bandwidth utilization [19]. Similar works, like *TDGraph* [3] and *MeNDA* [20], both employ coalescence when fetching sparse elements from a wide interface to L2 memory and dual inline memory module (DIMM) Rank, respectively. However, they too rely on sequential coalescing within a limited window of fewer than 40 elements [3], [20]. To the best of our knowledge, this work is the first to leverage both parallel indexing and parallel coalescing for accelerating indirect access near main memory and to delve deeply into their effects.

VI. Conclusion

We proposed a low-overhead coalescer aid for AXI-Pack to efficiently stream indirect accesses from a high-granularity DRAM interface. By leveraging memory-level parallelism and coalescence, it improves indirect stream bandwidth by 8x and offers an end-to-end speedup for SPMV by 3x. Remarkably, its solution for SPMV also demonstrates 1.4x and 2.6x on-chip efficiency compared to SoA vector processors, while retaining 1x and 0.9x SPMV efficiency.

VII. Acknowledgment

This work has been supported in part by 'The European Pilot' project under grant agreement No 101034126 that receives funding from EuroHPC-JU as part of the EU Horizon 2020 research and innovation programme.

References

[1] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, pp. 241:1–241:124, 2021.

[2] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[3] J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang *et al.*, "Tdgraph: a topology-driven accelerator for high-performance streaming graph processing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 116–129.

[4] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer, 2010, pp. 111–125.

[5] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 46–56.

[6] C. Zhang, P. Scheffler, T. Benz, M. Perotti, and L. Benini, "Axi-pack: Near-memory bus packing for bandwidth-efficient irregular workloads," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[7] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama, "Impulse: building a smarter memory controller," *Proc. Fifth Int. Symp. High-Performance Comput. Architecture*, pp. 70–79, 1999.

[8] G. S. Lloyd and M. B. Gokhale, "In-memory data rearrangement for irregular, data-intensive computing," *Computer*, vol. 48, pp. 18–25, 2015.

[9] S. Li and W. Liu, "Accelerating gustavson-based spmm on embedded fpgas with element-wise parallelism and access pattern-aware caches," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[10] Arm, "AMBA AXI and ACE Protocol Specification," https://developer.arm.com/documentation/ihi0022/hc.

[11] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 27, pp. 2629–2640, 2019.

[12] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A "new ara" for vector computing: An open source highly efficient risc-v v 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022, pp. 43–51.

[13] M. Jung, C. Weis, and N. Wehn, "Dramsys: a flexible dram subsystem design space exploration framework," *IPSJ Transactions on System LSI Design Methodology*, vol. 8, pp. 63–74, 2015.

[14] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, 2011.

[15] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running spmv on long vector architectures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 292–303.

[16] C. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2020, pp. 1–7.

[17] K. Choo, W. Panlener, and B. Jang, "Understanding and optimizing gpu cache memory performance for compute workloads," in *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*. IEEE, 2014, pp. 189–196.

[18] X. Wang, A. Tumeo, J. D. Leidel, J. Li, and Y. Chen, "Mac: Memory access coalescer for 3d-stacked memory," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[19] A. Segura, J.-M. Arnau, and A. González, "Scu: a gpu stream compaction unit for graph processing," in *Proceedings of the 46th international symposium on computer architecture*, 2019, pp. 424–435.

[20] S. Feng, X. He, K.-Y. Chen, L. Ke, X. Zhang, D. Blaauw, T. Mudge, and R. Dreslinski, "Menda: a near-memory multi-way merge solution for sparse transposition and dataflows," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 245–258.