Conjunctive Merge Instruction to Accelerate Sparse Matrix - Dense Vector Multiplication

Manuel Osterno*, César Marcon*, Jarbas Silveira†, Fernando Moraes*, Jardel Silveira†
*School of Technology – Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
†UFC - Universidade Federal do Ceará
adahilmuniz@hotmail.com, jardel@ufc.br, jarbas@lesc.ufc.br, fernando.moraes@pucrs.br, cesar.marcon@pucrs.br

***Abstract*—Sparse linear algebra is essential in many domains due to reduced computation and efficient memory usage. However, the irregularity of sparse data poses challenges for conventional software and hardware. While specialized accelerators offer performance gains, they lack general-purpose flexibility and rely on processor communication, creating bottlenecks. This work addresses these issues by proposing a tiling strategy to improve vector register usage and extending the RISC-V Vector (RVV) ISA with a custom merge instruction. Experiments using the gem5 simulator show that the tiled vector version achieved speedups of up to 1.30x (95% sparsity) and 1.72× (65%). In contrast, the version with merge instructions reached up to 1.81× and 6.04×, respectively, over a baseline implementation.**

***Index Terms*—Sparse Algebra, Risc-V, Vector ISA, Acceleration, MAC.**

I. Introduction

Sparse algebra operations are essential in applications such as linear solvers [1], graph mining, and machine learning [2]. In neural networks, sparsity arises from zero activations and pruned weights, often reaching over 70% [3]. To reduce memory usage, sparse matrices are typically stored in formats like Compressed Sparse Row (CSR) or Compressed Sparse Fiber (CSF).

By skipping zero computations, sparse operations reduce execution time, energy, and memory bandwidth. These formats avoid unnecessary activations of arithmetic units and minimize data flow, improving performance and lowering power consumption. However, their irregular structure leads to scattered memory access and frequent cache misses, which remain challenging for general-purpose processors. While some accelerators target sparse operations [4], they lack generality and rely on external cores. In contrast, Instruction Set Architecture (ISA) level support for sparse data remains limited [5], [6].

This work addresses this gap by extending the RISC-V Vector (RVV) ISA with a custom instruction, `vconv`, to perform conjunctive merge operations over integer data. We integrate this extension into the gem5 simulator and demonstrate its application in Sparse Matrix–Dense Vector Multiplication (SpMxDV), leveraging a tiling strategy to improve vector register utilization.

II. Related Works and Motivation

To boost the performance of applications such as linear solvers [1], GPM [6], and Deep Learning [2], some efforts focus on designing Application-Specific Integrated Circuits (ASICs) with multiple Processing Elements (PEs) and sparse acceleration units [7], [8], or using FPGAs [9]. While ASICs deliver high performance, they lack flexibility and depend on external general-purpose processors. To address this, other projects explore Application-Specific Instruction Processors (ASIPs) [10].

This chapter reviews representative ASIC and ASIP solutions to assess their strengths and limitations, and to distinguish this work's contribution.

A. ASIC Approach

The Tensor Marshaling Unit (TMU) [7] integrates with the Central Processing Unit (CPU) to handle sparse matrix index operations, enabling acceleration of various algorithms such as Gustavson, SpAdd, and SpMxDV.

NeCTAr [8] is a System-on-chip (SoC) that places sparse accelerators near the cache, handling sparse data loading, decoding, and computation. These units are tightly coupled to caches and scratchpads to optimize data flow.

Both projects design dedicated hardware for sparsity, requiring CPU integration for general-purpose tasks. In contrast, our work proposes extending the processor ISA itself.

B. ASIP Approach

Scheffler et al. [11] to optimize indirect memory accesses for sparse linear algebra within a specialized SoC [12].

IndexMAC [13], [14] extends RVV with a vindexmac instruction for SpMxDV acceleration. However, its focus on structured sparsity limits general applicability.

VIA [15] introduces a co-designed vector architecture with Smart Scratchpad Memory (SSPM), a Content Addressable Memory (CAM) that stores fiber values and indices to improve data handling. While effective, this adds area overhead.

SparseCore [6] defines a custom ISA for "streams"—structured key-value data—supported by 14 specialized instructions. Though innovative, it results in a highly domain-specific core.

To conclude, some of the works chose the skip strategy, that implements hardware to avoid computation by skipping the zero operands like EIE SAVE and SparCE, this strategy, however, seems to be going out of use, since it consumes a considerable amount of area only to do the zero identification on fetch stage and the current formats used to represent sparse data already omit the zero values. Other promising approach is the indirection access optimization architectures (UVE and ISSR); these architectures still add an area and time overhead to the processor datapath. In the end, we have the merge optimizati onstrategy, the merge operations (conjunctive

and disjunctive) are presented on the basic sparse algorithms (SpMxDV, SpMxDM, SpMxSpM, and SpAdd), as shown in Chapter III, so, optimizing this process may result in good performance results [4] [16] [7] [10].

To conclude, our work contrasts from the cited ones in some points, aiming to accelerate SpMxDV by extending the RVV ISA to add a conjunctive merge instruction, maintaining CPU generality, reusing existing hardware, and minimizing compiler and SoC changes.

III. Background

Sparse matrices, vectors, and tensors are characterized by many zero elements, enabling performance and memory optimizations. The sparsity level, defined in Equation 1, quantifies this characteristic.

$$Sparsity = ZeroElements/TotalElements \quad (1)$$

Higher sparsity favors formats that omit zero values, such as Compressed Sparse Row (CSR) and Compressed Sparse Fiber (CSF), the latter being an extension of CSR [17].

The CSR format compresses row information to reduce memory usage, avoiding the redundancy found in the CO-Ordinate (COO) format. It uses three arrays: `IA`, `JA`, and `A`. The `IA` array stores the number of non-zero (NZ) elements until that row, therefore, to find the number of NZ elements in the i-th row, use Equation 2.

$$NZ[i] = IA[i+1] - IA[i] \quad (2)$$

The *JA* array contains the column indices where to find NZ elements and the *A* array contains the actual NZ value.

Figure 1 illustrates a CSR-encoded sparse matrix.

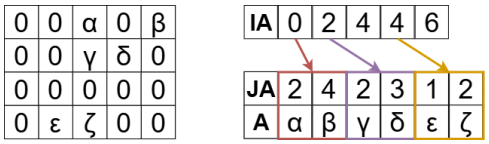

Fig. 1. CSR Sparse Matrix Representation.

In that example, the 0-th row contains two NZ elements (2-0, see Equation 2) and those are α and β placed within columns 2 and 4.

A key sparse operation is Sparse Matrix–Dense Vector Multiplication (SpMxDV), widely used in deep learning [2]. Code 1 shows its algorithm, which iterates over each matrix row and performs element-wise multiplication for non-zero entries. The algorithm performs a conjunctive merge of indices, followed by multiplication and accumulation into a dense output vector.

```
for i in 0 to N_ROWS:
    for j in IA[i] to IA[i+1]-1:
        y[JA[j]] += A[j]*x[JA[j]]
```

Listing 1. SpMxDV algorithm.

IV. Methodology

To evaluate our approach, we developed two versions of the SpMxDV algorithm: one based on the RVV ISA and another incorporating a custom conjunctive merge instruction. We also implemented a tiling strategy to optimize vector register usage and memory access.

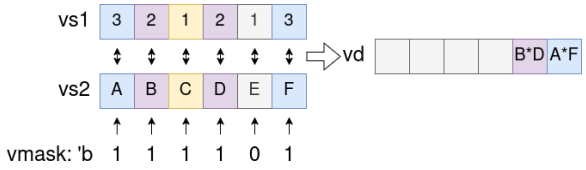

Fig. 2. vconv usage example.

A. Conjunctive Merge Instruction

The SpMxDV operation involves matching column indices and multiplying the corresponding values, followed by a reduction. We extended the RVV ISA in gem5 to accelerate this with the `vconv` instruction. This instruction accepts index and data vectors (`vs1`, `vs2`), a mask (`vmask`), and stores the result in `vd`.

```
# Syntax pattern for conjunctive merge inst.
vconv.vv vd, vs1, vs2, vmask.t
```

Listing 2. Proposed conjunctive merge instruction pattern.

The vconv instruction compares the values present in vs1 (index vector) and, when any of these values match, the values in vs2 (data vector) relative to the values in vs1 are multiplied and the result is stacked at the bottom of vd (result vector). Figure 2 shows two input vectors (vs1 and vs2), where each pair of elements of each vector is colored the same color and the result elements are colored the same color as the match that resulted in these values. So, reading from right to left (0 to N) the first index to match is three (for values A and F), so, the result A*F is stored at the position zero of vd, then the second index which match is two (for values B and D) and the result B*D is stored at position one of vd. Finally, the third index that might have matched is one, however, position four (where the second 1 is found) of the vectors is masked, and then this result is ignored.

B. vconv Hardware Instruction

We used the existing comparators employed by `vmseq` implementation in a multicycle scheme to minimize hardware overhead, avoiding dedicated hardware. The number of comparisons and cycles required for a vector of size n is given by:

After comparisons, the results are passed to existing multipliers used by the `vmul` instruction.

$$nb\_comparisons = \frac{n^2 - n}{2}, \quad nb\_rounds = \left\lceil \frac{n-1}{2} \right\rceil \quad (3)$$

C. SpMxDV Vectorized and Merge Versions

The classic C-based SpMxDV code was not vectorized automatically due to indirect memory access. Thus, we created:

- **SpMxDV Vectorized**: operates with IA, JA, A, and x in vector registers and explicitly performs scalar operations within loops (Code 3).

- **SpMxDV Merge**: replaces the inner loop with the `vconv` instruction and performs reduction via `vredsum` (Code 4).

```
vload IA, JA, A, x
for i in 0 to N_ROWS:
    # BLOCK 0: Get IA[i] and IA[i+1]
    j_init = IA[i] # Move IA[i] to scalar
    j_final = IA[i+1] # Move IA[i+1] to scalar

    for j in j_init to j_final:
        # BLOCK 1: Get A[j], x[JA[j]] and
        # A[j] * x[JA[j]]
        A_j = A[j] # Move A[j] to scalar reg
        JA_j = JA[j] # Move JA[j] to scalar reg
        x_JA_j = x[JA[j]] # x[JA[j]] to scalar
        res = A_j * x_JA_j # Mult. the scalars

        # BLOCK 2: Positioning Result
        v_aux = 0 # Zero the entire vector
        v_aux[i] = res # Set mult. result

        # BLOCK 3: Storing result into y vector
        y += v_aux # Vector accum. the result
vstore y
```

Listing 3. SpMxDV Vectorized

```
vload IA, JA, A, x
create idx_x # Create the dense index vector
# BLOCK 0: Vectors Initialization
v_val_concat = {x, A} # Concatenation of values
v_idx_concat = {idx_x, JA} # Concat. index vect.
# END OF BLOCK 0
for i in 0 to N_ROWS:
    # BLOCK 1: Get number of non-zeroes on row i
    j_init = IA[i] # Move IA[i] to scalar
    j_final = IA[i+1] # Move IA[i+1] to scalar
    nnz = j_final - j_init # Number of non-zeroes

    # BLOCK 2: Build the mask
    v0 = build_mask(nnz)

    # BLOCK 3: Calculate A[:], x[JA[:]] for all
    JA and A elements of the row and sum them up
    #Conjunction
    "vmconv.vv v_merge_res, v_val_concat,
    v_idx_concat, v0.t \n"
    #Sum Reduction
    res = redsum(v_merge_res)

    # BLOCK 4: Store the value in the result
    vector
    y[i] = res
```

Listing 4. SpMxDV Merge - using vconv instruction.

The merged version improves performance by simplifying the inner loop using a masked conjunctive merge.

The SpMxDV Vectorized algorithm is divided into four blocks, considering that the vectors are already instantiated. In "BLOCK 0" the IA vector is manipulated to get the IA[i] and IA[j] values, the boundaries of the row, in "BLOCK 1" these values are used to manipulate A, JA and x vector to get the elements to be multiplied and then multiply them (A[j] * x[JA[j]]). In "BLOCK 2" and "BLOCK 3", the result is put in the correct place and accumulated into the result vector; creating and manipulating the masks omitted in the pseudo-code is necessary.

The SpMxDV Merge version erases the inner loop of the SpMxDV Vectorized, since that loop was used to "merge" the vector, which is played by the instruction vconv. As in the prior code, this one also considers that the vectors have already been initialized.

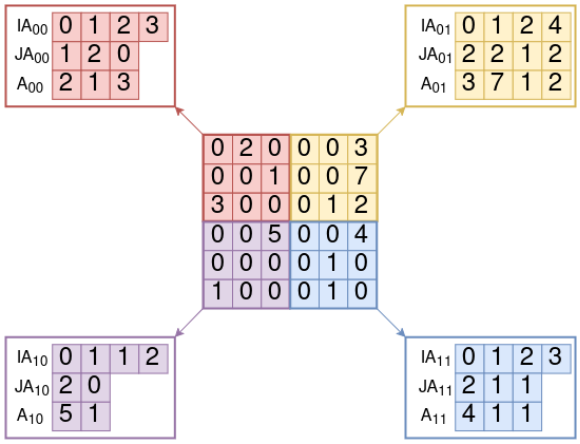

Fig. 3. Proposed tiling strategy.

The BLOCK 0 is responsible for concatenating the sparse and dense data vectors (A and x) and the sparse and dense index vectors (JA and idx_x) because only one vector register is shared by data and only one is shared by indices. The code BLOCK 1 gets the number of non-zeroes elements from row i (A[i+1]-A[i]) and BLOCK 2 builds the mask that will be used by vconv instruction, this mask enables all the elements from the dense part and enables only the elements of the row being used by the iteration. For example, if we have a vector of sixteen elements (eight for dense and eight for sparse), but in the row zero, only two elements are valid, the mask of this iteration should assume the following binary value: 'b1111_1111_0000_0011.

In BLOCK 3, the conjunction merge instruction uses the concatenated registers and the mask built previously. The results (considering the example, A[0]*x[JA[0]] and A[1]*x[JA[1]]) are then reduced by the sum reduction instruction (vredsum). The final result is placed in the correct position of the final vector (v20) in BLOCK 4.

D. Tiling Sparse Matrices

Since CSR arrays often exceed vector register capacity, we proposed a tiling strategy that splits the original dense matrix into uniform tiles, ensuring the number of rows and non-zero elements per tile satisfy $NZ \leq VL$, where NZ is the number of Non-Zero elements and VL is the vector length. Figure 3 illustrates this strategy.

We implemented two methods:

- **Optimized**: iteratively refines tile sizes to meet $NZ \leq VL$.
- **Fast**: uses fixed $\sqrt{VL} \times \sqrt{VL}$ tiles for guaranteed compliance.

The optimized method was adopted for evaluation. After tiling, matrix-vector multiplications are performed tile-wise using the proposed SpMxDV versions, and results are accumulated using the `vadd` instruction.

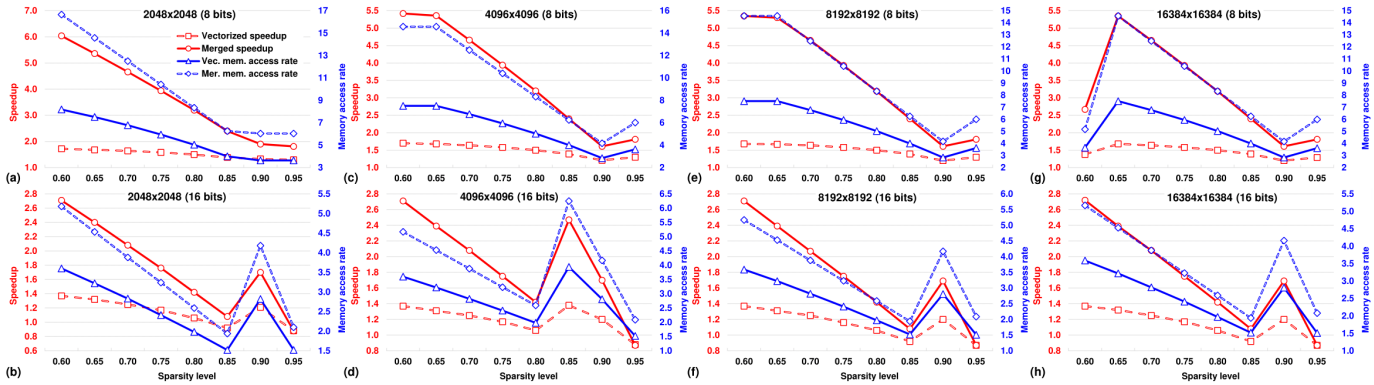

Fig. 4. Speedup (left axis, red lines) and Memory Access Rate (right axis, blue lines) for vectorized and merged SpMxDV versions |(a), (c), (e), (g) - 8 bit data type |(b), (d), (f), (h) - 16 bit data type.

V. Results

A. Experimental Setup

We developed a Python-based infrastructure to generate sparse matrices (via *Scipy Sparse*), tile them using the optimized strategy, and convert them into CSR format. Parameters include matrix size (from 2048×2048 to 16384×16384), sparsity level (60%–95%), and data type (8-/16-bit integers). The tiled and classic CSR matrices are compiled into ELF binaries for execution on gem5.

The target architecture is a RISC-V RV64GCV system (Table I), featuring a 256-bit vector engine, DDR4 memory, and L1/L2 caches. Emulation results include execution time (ticks), memory accesses, and cache statistics. Theoretical latency per vector operation was 16 cycles (8-bit) and 8 cycles (16-bit), based on Equation 3.

TABLE I
Emulation System Properties.

Processor core	RISC-V 64GCV 256-bit vector engine L1I: 16kB, 2 cycle latency L1D: 64kB, 2 cycle latency
L2 cache	512kB 8 cycle latency
Main memory	DDR4-2400

B. Experimental Results

Figure 4 shows speedups and memory access rates across eight sparsity levels (60% to 95%). Performance improves with lower sparsity (higher density) due to greater row occupancy, which enhances vector register utilization. An exception occurs in Figure 4 (d) for a 4096x4096 matrix, where at 85% sparsity the tile size is larger than at 80%, yielding better performance despite lower density.

Maximum speedups for 8-bit data at 60% sparsity (2048×2048) were 1.72× (Vectorized) and 6.04× (Merged), tied to a tile size of 16. For 16-bit data, both versions had the lowest speedups (0.88×) due to tile management overhead that outweighed performance gains.

Both approaches enhance data locality via vector registers. Memory access is reduced by over 8× (Vectorized) and 16× (Merged) compared to the classic method for 8-bit data.

SpMxDV Merged outperforms the Vectorized version due to fewer memory accesses, especially at lower sparsity and 8-bit data. At 65% sparsity, it requires half the memory accesses and executes 1.88× fewer instructions, leading to faster execution. Merged version also spends more time (46.18%) on arithmetic operations than Vectorized (26.66%), indicating better computational efficiency.

Performance improves with lower sparsity due to better use of vector registers. However, this trend may reverse if tile sizes are smaller for denser matrices, as seen in Figure 4(b), comparing 85% vs. 90% sparsity.

TABLE II
Algorithm Execution Profile (mean values)

	SpMxDV Vectorized	SpMxDV Merged
Number of executed instructions	825,367,204.23	438,035,436.31
Arithmetic instruction rate	26.66%	46.18%
Memory accesses	115,141,942.35	79,653,370.09
Speedup over classic	1.31	2.60

VI. Conclusion

This work proposed a tiling strategy and a conjunctive merge instruction to accelerate Sparse Matrix–Dense Vector Multiplication (SpMxDV). Experimental results show that, depending on data type and sparsity, the Vectorized and Merged versions achieved speedups of up to 1.68× and 5.90×, respectively, with average gains of 1.30× and 2.47×. Significant reductions in memory accesses support the improvements, up to 8× (Vectorized) and 16× (Merged), compared to the classic approach. Additionally, the Merged version executed 1.81× fewer instructions than the Vectorized version, contributing to its superior performance. The proposed strategy effectively boosts SpMxDV performance, offering promising benefits for machine learning workloads, particularly in deep neural networks and graph processing.

ACKNOWLEDGMENTS

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Finance Code 001; Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), 308182/2023-5, 305621/2024-6; National Institute of Science and Technology (INCT-Signals) under grant 406517/2022-3; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), 23/2551-0002200-1; HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA.

REFERENCES

[1] H. Anzt, S. Tomov, and J. Dongarra, "Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product," in *Symposium on High Performance Computing (HPC)*, 2015, pp. 75–82.

[2] 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," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.

[3] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. Keckler, and W. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," 2017. [Online]. Available: https://arxiv.org/abs/1708.04485

[4] G. Zhang, N. Attaluri, J. Emer, and D. Sanchez, "Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 687–701.

[5] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Indirection stream semantic register architecture for efficient sparse-dense linear algebra," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 1787–1792.

[6] G. Rao, J. Chen, J. Yik, and X. Qian, "SparseCore: stream ISA and processor specialization for sparse computation," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 186–199.

[7] M. Siracusa, V. Soria-Pardos, F. Sgherzi, J. Randall, D. J. Joseph, M. Moretó Planas, and A. Armejach, "A tensor marshaling unit for sparse tensor algebra on general-purpose processors," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. Association for Computing Machinery, pp. 1332–1346. [Online]. Available: https://doi.org/10.1145/3613424.3614284

[8] V. Schmulbach, J. Kim, E. Gao, L. Revina, N. Jha, E. Wu, and B. Nikolic, "NeCTAr: A Heterogeneous RISC-V SoC for Language Model Inference in Intel 16," Mar. 2025, arXiv:2503.14708 [cs]. [Online]. Available: http://arxiv.org/abs/2503.14708

[9] J. Oliver, C. Álvarez, T. Cervero, X. Martorell, J. D. Davis, and E. Ayguadé, "b8c: SpMV accelerator implementation leveraging high memory bandwidth," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2023, pp. 216–216, iSSN: 2576-2621. [Online]. Available: https://ieeexplore.ieee.org/document/10171501

[10] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Sparse stream semantic registers: A lightweight ISA extension accelerating general sparse linear algebra," vol. 34, no. 12, pp. 3147–3161. [Online]. Available: http://arxiv.org/abs/2305.05559

[11] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores," Apr. 2020.

[12] G. Paulin, P. Scheffler, T. Benz, M. Cavalcante, T. Fischer, M. Eggimann, Y. Zhang, N. Wistoff, L. Bertaccini, L. Colagrande, G. Ottavi, F. K. Gürkaynak, D. Rossi, and L. Benini, "Occamy: A 432-Core 28.1 DP-GFLOP/s/W 83% FPU Utilization Dual-Chiplet, Dual-HBM2E RISC-V-Based Accelerator for Stencil and Sparse Linear Algebra Computations with 8-to-64-bit Floating-Point Support in 12nm FinFET," in *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, Jun. 2024, pp. 1–2, iSSN: 2158-9682. [Online]. Available: https://ieeexplore.ieee.org/document/10631529

[13] V. Titopoulos, K. Alexandridis, C. Peltekis, C. Nicopoulos, and G. Dimitrakopoulos, "IndexMAC: A custom RISC-v vector instruction to accelerate structured-sparse matrix multiplications." [Online]. Available: http://arxiv.org/abs/2311.07241

[14] ——, "Optimizing Structured-Sparse Matrix Multiplication in RISC-V Vector Processors," Jan. 2025, arXiv:2501.10189 [cs]. [Online]. Available: http://arxiv.org/abs/2501.10189

[15] J. Pavon, I. V. Valdivieso, A. Barredo, J. Marimon, M. Moreto, F. Moll, O. Unsal, M. Valero, and A. Cristal, "VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations." IEEE Computer Society, Feb. 2021, pp. 921–934. [Online]. Available: https://www.computer.org/csdl/proceedings-article/hpca/2021/223500a921/1t0HUpXQTQs

[16] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 766–780. [Online]. Available: https://ieeexplore.ieee.org/document/9251978

[17] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: efficient vectorization of SpMV on x86 processors," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2018, pp. 149–162.