MSE: A Matrix Sparsity Extension for RISC-V

Luc Ribas [1], Iago Aquino [1], Isaías Felzmann [1], Lucas Wanner [1], Guido Araújo [1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas, SP – Brazil

{l1247231, i198921}@dac.unicamp.br, {ibf, wanner, guido}@unicamp.br

Abstract. *The increase in size and computing demands of modern Convolutional Neural Networks (CNNs) requires techniques that can reduce model size and accelerate inference execution. A popular method is pruning, where some model weights are zeroed in a controlled way while maintaining model accuracy. This method can create structured sparsity patterns in the weight matrix, allowing storage in compressed formats. This reduces the model's memory footprint while maintaining sequential access to the values. In this paper, we propose the Matrix Sparsity Extension (MSE). The proposed RISC-V instruction set extension takes advantage of the sparsity formats generated by pruning, reducing the amount of memory operations needed for matrix multiplication. MSE was able to provide speedups close to 1.24x over the baseline uncompressed operation, reaching 1.75x when the sparse case offers better alignment in cache.*

1. Introduction

Convolutional Neural Networks (CNNs) have been widely used in computer vision, becoming very relevant in fields such as analysis of complex medical imaging databases [Mienye et al. 2025], development of navigation systems for autonomous vehicles [Tang et al. 2023], and real-time object recognition for machines and robots [Maturana and Scherer 2015]. This is due to the ability of CNNs to extract meaningful patterns from visual data. However, to achieve high accuracy, these models often require millions of parameters, resulting in high computational and storage requirements.

Given the growing size of CNN models and the increasing demand for computational power to train and deploy them, significant implementation challenges arise for their efficient application, especially on resource-constrained devices. To address these challenges, weight pruning has become a widely adopted technique [He and Xiao 2024, Tang et al. 2022]. This process systematically sets a portion of the model's weights to zero, thereby reducing model size, lowering storage constraints, and accelerating inference time.

The generated sparse model can be structured in a N:M format, where for every M elements, N are non-zero. This improves the memory footprint and computation efficiency while preserving high accuracy [Mishra et al. 2021, Cao et al. 2019, Lin et al. 2023]. M and N can have different values depending on the compression accuracy trade-off. Some applications prefer to use a single format configuration, such as 2:4 [Mishra et al. 2021]. Others may allow for customization of M and N values [Lin et al. 2023].

While prior work on vector extensions was able to take advantage of this sparsity with minimal hardware cost [Titopoulos et al. 2023], some matrix extension hardware requires dedicated processing units to handle this type of structure [Jeong et al. 2023]. Others rely on external buffers in cache memory to decompress the data, such as Intel AMX [Intel Corporation 2024].

In this work, we present a RISC-V extension, on top of a Matrix Processing Engine (MPE) architecture composed of a matrix register file and matrix functional units to perform operations such as multiply-accumulate (MAC). Our Matrix Sparse Extension (MSE) of the MPE takes advantage of the N:M compressed format. MSE is composed of three new instructions that leverage a mask register to decompress pruned matrices from a compressed format directly into the engine register file. This enables the execution of matrix multiplication operations $C = A \times B$ where A is a sparse weight matrix and B and C are dense matrices.

To take full advantage of this method, a data reorganization strategy is employed where both the compressed non-zero values and their indices are lined up for sequential access. This approach provides an efficient data fetching mechanism that aims to achieve the highest possible memory bandwidth from the compressed format.

With the added support for sparse matrices, MSE saved up to 47% of the matrix memory footprint while providing consistent speedups in all hardware configurations, due to the reduction in memory accesses. For example, in the case of larger weight matrices, speedups reached up to 1.27x, as the packing time became less of a factor. In a scenario where the compressed format allowed for better cache alignment, the proposed approach achieves a 1.75x speedup. The MSE extension also improved general scaling of the MPE, improving the compute hardware utilization from 70% to 87% of the peak MPE performance on a 512 MAC Units design.

2. Related work

Many different types of hardware are used to perform high-performance General Matrix Multiplication (GEMM) operations, which can sometimes be extended for Sparse-Dense Matrix Multiplication (SpMM). General Purpose Graphics Processing Units (GPGPUs) make use of massive parallelism to perform this operation. The NVIDIA Ampere Architecture [NVIDIA Corporation 2020] takes advantage of this hardware to provide support for sparse 2:4 matrices [Mishra et al. 2021]. This is done through the usage of indices to select for multiplication only the non-zero elements of the B matrix, thus avoiding multiplying zeroes, and hence reducing total arithmetic operations.

Regarding integrated matrix extensions that perform GEMM using the outer product of 1D vectors and a 2D accumulator, sparse extensions are present in the AMD Instinct MI300 Accelerators [Advanced Micro Devices, Inc. 2024]. This hardware is also able to take advantage of the pruned 2:4 format by using special SpMM instructions that allow for the K dimension of the accumulator to be doubled in comparison with the dense version, while requiring the same amount of cycles to complete.

For Central Processing Units (CPUs), matrix and vector extensions are common ways to accelerate SpMM. For the RISC-V architecture, a new instruction has been proposed in the RISC-V Vector Extension [Waterman et al. 2021] that allows for indirect

reads to a preloaded register file [Titopoulos et al. 2023, Titopoulos et al. 2025]. The authors use a Row-wise algorithm that supports multiple sparsity values. For

Other types of matrix extensions have 2D matrix registers, or tiles, to perform GEMM operations. SparseZipper uses these structures for multiplying matrices with unstructured sparsity [Ta et al. 2025]. VEGETA is an extension for tiled acceleration of N:M sparse-dense multiplication [Jeong et al. 2023]. This hardware supports multiple N and M configurations, even N = M = 1, which represents the dense case. This is achieved by adapting both the register file and the multiplication units to handle sparse data. The former is implemented with metadata registers that store the indices of the compressed values, and the latter with multiplexers that use these new registers as inputs, avoiding multiplying values by zero.

Another approach for tiled registers is to decouple the decompression of sparse data from the arithmetic computation. This method leverages existing dense multiplication data paths, avoiding substantial changes in the execution pipeline. Intel AMX [Intel Corporation 2024] does this by using its vector extension AVX to load and decompress the sparse matrix into the cache memory. The masks in this architecture are stored in an activation bits format, which would require an initial conversion from the N:M indices. The generated decompressed buffers can be loaded into the 2D registers of AMX and multiplied by the matrix operators.

However, Intel's approach adds memory operation overhead and occupies part of the cache with the decompressed buffers, leaving less room for the A and B matrices. Contrary to Intel, our approach is based on the N:M format by default and avoids the use of extra cache data by performing decompression directly into the register file. This has the benefit of not relying on the vector extension, lower memory overhead, and better cache utilization, while not requiring hardware modifications in the multiplication datapath.

3. The Matrix Processing Engine (MPE)

The support for matrix operations is usually added with a dedicated Matrix Processing Unit (MPU), which separates matrix instructions from the scalar datapath, reducing complexity and improving performance overall. The Matrix Processing Engine (MPE) used in this paper follows a load-store architecture with 32 matrix registers that hold blocks of the matrices, called *tiles*, which are organized as bi-dimensional square grids[1]. MPE arithmetic operations, such as matrix multiplication, are performed in dedicated functional units, called Matrix Processing Units (MPUs), as shown in Figure 1.

The instructions that operate over matrix registers are described as follows:

- Memory Access: Both load and store instructions are available with a contiguous version (`ml` and `ms`, respectively) and a strided version that fetches rows separated by an offset (`mls` and `mss`).
- Arithmetic: Basic addition (`madd`) and subtraction (`msub`) are implemented, as well as a multiply-accumulate instruction used to calculate an inner product between two matrix registers (`mmacf`, for float).

[1]The draft specification is available at `https://ic.unicamp.br/~isaias/riscv-mpe-spec.html`

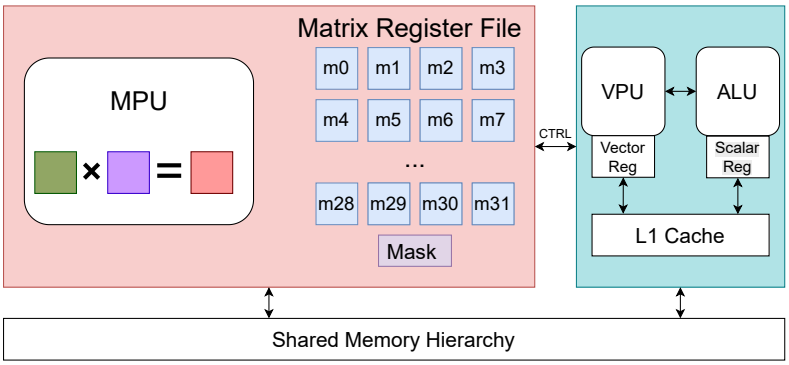

Figure 1. Illustration of the MPE organization. The Matrix Register File with 32 Matrix registers and MPU are separate from the scalar and vector units, communicating through control.

- Data Movement: Scalar to Matrix movement instruction to fill a register with a scalar value (`mmvxmu`). There is also a specialized pseudoinstruction to zero an entire matrix register (`mzero`).
- Mask Operations: A single *Mask Register* enables masked load and store operations, where only matrix elements marked with a '1' are accessed/stored. This dedicated register can be configured with common patterns using specialized instructions such as for diagonal (`mmkd`) and upper/lower triangular (`mmkut`/`mmklt`) structures.

4. Structured N:M Sparsity Format

To store N:M matrices efficiently, all the non-zero elements are stored sequentially in a *Value Matrix*, and an auxiliary *Index Matrix* is created for storing the indices of the values within their respective M window. Figure 2 shows an example of 2:4 compression.

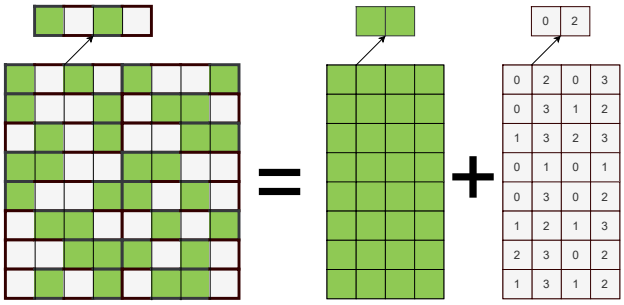

Figure 2. Compressed format for 2:4 matrix. The left-hand side represents a dense matrix that has been pruned, and the right-hand side shows the compressed result, split into a value matrix and an index matrix.

To minimize metadata memory overhead, the indices can be stored using the least possible amount of bits, so that the saving in storage capacity (S_s) can be expressed as a function of the sparsity and of the element datatype width (D_s) as shown in equation 1.

$$S_s = 1 - \frac{N}{M}\left(1 + \frac{\lceil log_2(M) \rceil}{D_s}\right) \tag{1}$$

For a 32-bit operand with 2:4 sparsity, this results in a $\approx 47\%$ storage savings, while for 16-bit operands, $\approx 44\%$ is saved. Changing the sparsity to values such as 1:4 and 1:8 will lead to larger savings in memory.

5. Matrix Sparsity Extension (MSE) Specification

For adding support for N:M matrices, we propose MSE, which consists of three new instructions for decompressing the value matrix inside the register file based on the index. These operations make use of the existing MPE mask register to store and spread the indices within the matrix registers. In the presented form, our extension provides support for 2:4 structured sparsity matrices, and we show how this can be expanded. This design adds minimal hardware overhead to the MPE, and avoids changes to the MAC units.

5.1. Mask Set (`mmkset`): Setting a sparse matrix mask

This instruction loads the data of a regular matrix register to the mask register, changing the format from indices to activation bits. Therefore, bits that match the index of each corresponding M-element window are set to 1 while all the others are set to zero. Figure 3 shows how the instruction works for a 2:4 matrix. The instruction receives as parameters a matrix register loaded with the mask indices and a scalar with the sparsity values.

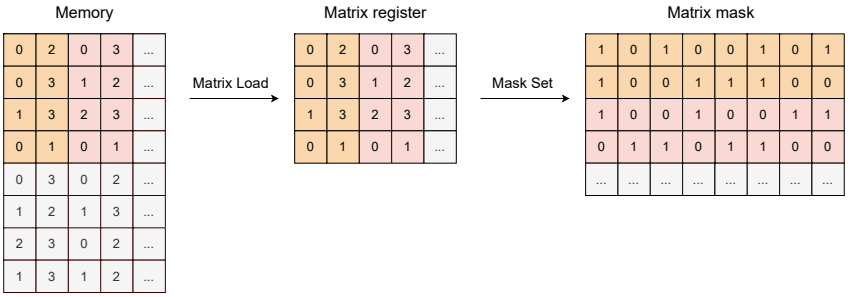

Figure 3. Example of use of `mmkset` instruction for 2:4 sparse matrix. The data is first loaded from memory into a matrix register and then moved to the mask register while being converted to the activation bits format.

5.2. Mask Shift (`mmkshift`): Choosing which mask to apply

To store the indices in the dedicated mask register, one bit per element is needed. Because the mask register is, by design, at least as large as a normal matrix register, the activation bits corresponding to multiple tiles can be stored at once.

The `mmkshift` instruction enables access to the mask values corresponding to each individual tile by moving the data a certain number of bytes towards the beginning of the register and discarding the first elements. This allows the region of interest within the mask register to always be at the start. Figure 4 shows a shift of two bytes (16 bits).

5.3. Mask Spread (`mmksp`) : Decompressing the sparse matrix

The last instruction decompresses the data from one dense matrix register onto itself and one other matrix register using the mask register activation bits. This is done by setting the values only at bits marked as ones in the mask register, similar to what is done with

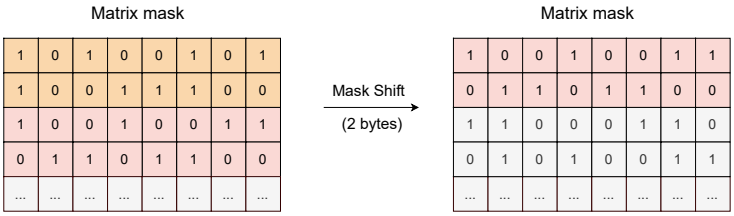

Figure 4. Example of `mmkshift` instruction. The shift removed the first 16 elements of the mask register, moving the others towards the front.

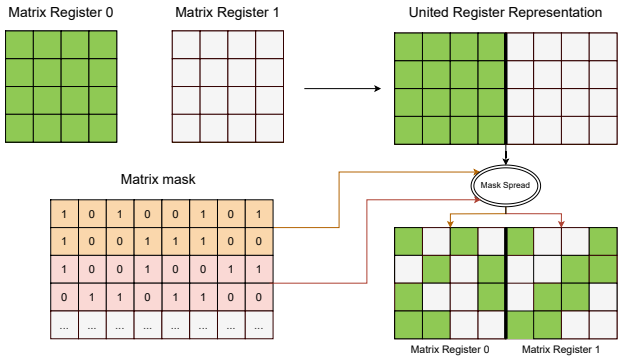

Figure 5. Example of *mmksp* instruction. Matrix Register 0 is initially dense, and Matrix Register 1 has no useful data. The values from Matrix Register 0 are then spread onto both registers, setting only the values that are marked with ones in the mask register.

the AVX extension for Intel [Intel Corporation 2024]. Figure 5 shows how the operation works for 4×4 registers with 2:4 sparsity.

This instruction receives two matrix registers as inputs and spreads the data contained in the first onto both. This implementation somewhat limits the sparsity values, allowing only for a sparsity ratio of 50%. This issue could be overcome by treating the output register as a group, as done by the RISC-V V LMUL feature in the Vector Extension [Waterman et al. 2021]. This allows for the data in the first register to be spread across multiple others, making the sparsity flexible. The RISC-V Matrix Extension is still under specification [2], and some mechanism similar to LMUL is likely to be included. Our MSE instructions can be easily extended to support these register groups and more flexible sparsity structures.

5.4. Sparse-Dense Matrix Multiplication

In a matrix multiply accumulate operation like $C_{m\times n} = A_{m\times k} \times B_{k\times n}$, where A is sparse and B and C are dense, these new instructions allow the application to reduce the amount of loads of the values of matrix A by the sparsity ratio. For the mask, new loads have to be added, but they are less frequent, being needed once every $Mask_l = \frac{M \cdot Register\_size}{N \cdot Elems\_per\_Reg}$ loads of A, which is one every 16 loads for a 4×4, 512-bit register.

Algorithm 1 shows a simple version of a matrix multiplication using the new instructions. For clarity, it is assumed that one matrix register can hold the masks for one

[2]Information about the task group for this specification is available at `https://riscv.atlassian.net/wiki/spaces/AMEX/pages/55083388/`

Algorithm 1 Basic Matrix Multiplication Kernel

for each output block C_{ij} to be computed **do**
    mzero m0
    ml m6, A_mask_i
    mmkset m6, 2:4
    for $k = 0$ **until** $B\_rows - 1$ **step** 2 **do**
        ml m4, A_{ik}
        mmksp m4, m5
        ml m1, B_{kj}
        mmacf m0, m4, m1
        ml m2, $B_{(k+1)j}$
        mmacf m0, m5, m2
        mmkshift 4
    end for
    ms m0, C_{ij}
end for

whole row of A and that the operands are tiles of 4×4 elements, where the corresponding mask for each tile is 16 bits, or 2 bytes. This shows how loading a single element of A allows for the multiplication with two elements of B by decompressing the registers. Figure 6 shows an overview of the micro-architecture of this operation, where A and B are multiplied and accumulated to generate C. Note that B is transposed.

6. Packing and high-performance kernels

To maximize efficiency for matrix multiplication using the MPE, all 32 registers must be used to perform computations. This is done by loading x registers from the A matrix and y registers from the B matrix to generate xy registers for the C matrix. By sliding this window along the k dimension, it is possible to accumulate the C registers and generate the final values. For a sparse-dense matrix multiplication, the amount of loads per kernel relative to the dense implementation (R_l) is given by equation 2.

$$R_l = \frac{x\left(\frac{N}{M} + \frac{1}{Mask_l}\right) + y}{x + y} \tag{2}$$

Because $M > N$ and $Mask_l > 1$, this generally means that maximizing x minimizes the total number of loads needed in the kernel. Another constraint is knowing when to load the index matrix. A conditional check after each data load is too inefficient. Instead, by choosing an x that is an integer divisor of $Mask_l$, the operation becomes periodic, allowing the kernel to load the index matrix at a fixed, predictable interval without the costly checks. Our final window size was chosen to be $\{x, y\} = \{8, 3\}$, as demonstrated in Figure 7, with $y = 3$ being chosen to maximize register use.

To access the values of matrix A in the correct order for using the mask spread operation, a special packing strategy is needed, where the registers used in the same window are sequential in memory. The mask requires a similar packing, which involves ordering the bits in memory such that the indices align with the corresponding values.

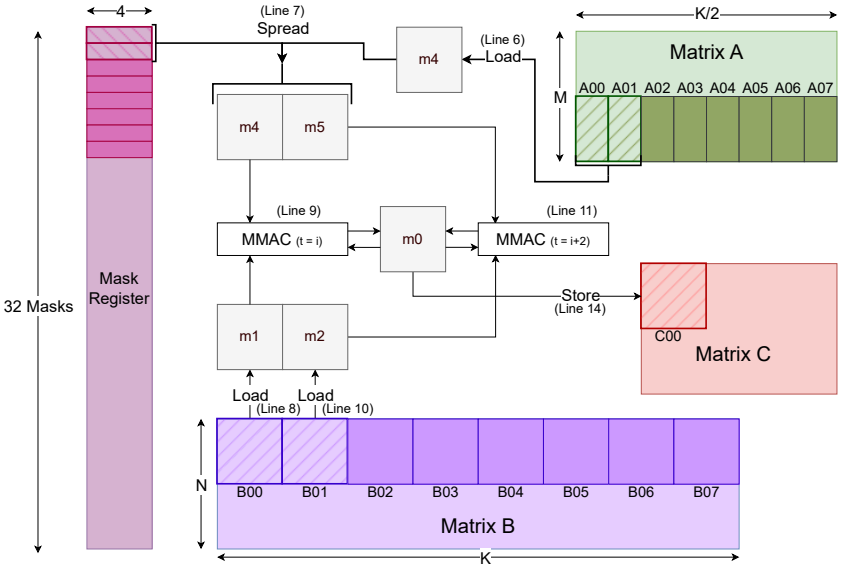

Figure 6. Schematic overview of MSE. The mask register was previously loaded with mask values. m4 is loaded densely, and then its values are spread between itself and m5. There is only one Matrix Multiply-Accumulate (MMAC) unit, so the two `mmacf` operations are done in the same unit sequentially. The `mmacf` operation reads from m0 to perform the accumulation operation.

7. Experimental Evaluation

To evaluate our MSE, we used gem5 [Lowe-Power et al. 2020] to create a MPE implementation, based on prior work on gem5 extensions [C. Aquino et al. 2024], and added the proposed instructions. We instantiated a separate 2 MB L2 cache (20 cycles latency) for the matrix unit and a DDR4 memory model was used (Alveo U250 DDR4).

The matrices were created artificially with random FP32 values, where matrix A was randomly pruned and compressed accordingly. We tested how MSE affected the performance for multiple configurations of MPE parameters, scaling register size, and the number of MAC units. We also evaluate the impact of packing and compare the results based on both the generated speedups and the peak performance of the MPE.

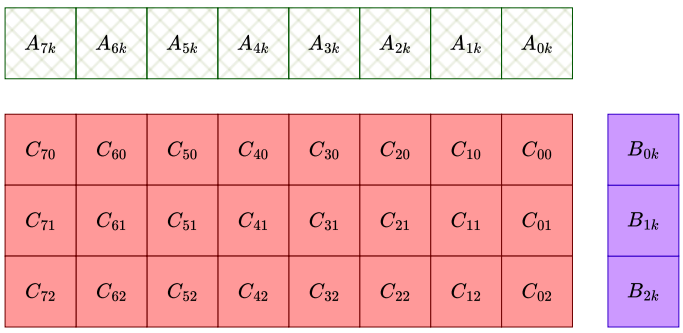

Figure 7. 8×3 window kernel. This kernel is performed by sliding the window through dimension k, in a C-stationary fashion. For one kernel iteration, the registers of B are kept the same, while the remaining registers are cycled to generate all the A matrix values.

As a baseline for comparison, a dense kernel with a 5x5 shape was used, in which 10 loaded registers can perform 25 `mmacf` instructions. This results in the most significant possible computational intensity value for a dense kernel in the 32 register architecture, 5 FLOPs/byte. This computational intensity is approximately 26% lower than the 6.8 FLOPs/byte of our sparse 8x3 kernel. We tested without any tiling strategies to verify how cache and RAM usage affect the results. This allowed direct measurement of slowdowns caused by accessing the slower main memory, by preventing packed buffers from being consistently held in the faster cache.

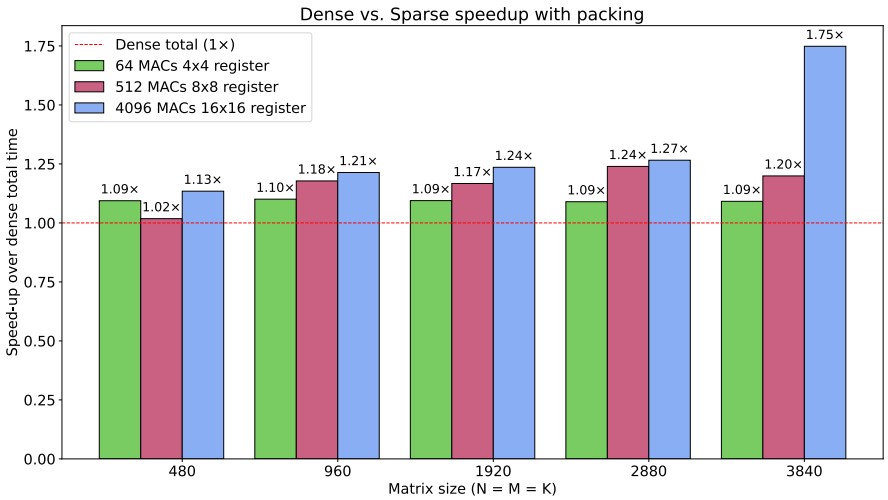

Figure 8. Speedup of different configurations of sparse implementation vs dense baseline. In general, larger hardware leads to greater speedups. The speed of larger configurations increases with matrix size as packing becomes less of a factor.

The results shown in Figure 8 illustrate the impact of varying register size and number of Multiply-Accumulate Units of the MPE on the speedup of MSE over the dense implementation. In most cases, larger hardware represents a higher speedup because the improvements in compute power make the problem more memory-bound, which favors the compressed format of the sparse matrices.

The speedup trend varies significantly with the number of MACs. While the 64 MACs configuration maintains a relatively consistent speedup across all matrix sizes, the 512 MACs and 4096 MACs cases show an initial increase in speedup with matrix size. Still, the former eventually reaches a plateau, and the latter has a spike at $N = 3840$. This sudden increment for the last data point at 4096 MACs represents the size at which cache thrashing starts to occur for the dense implementation, but not for the sparse one, resulting in a higher time difference. In the other instances, the overall growth pattern is directly linked to the diminishing relative impact of the packing phase as the computational workload increases.

Figure 9 demonstrates that for smaller matrix dimensions with larger hardware, packing dominates the total GEMM/SpMM execution time, especially for the higher MAC counts. This explains why, in the $N = 480$ test case, the 512 MACs configuration performs slower than 64 MACs, as its substantial packing overhead at this size negates the benefits of its faster compute kernel. This is not the case for the 4096 MACs configuration,

which still achieves a speedup over 64 MACs at $N = 480$, because of the higher speed advantage of its computing kernel. In particular, the 4096 MACs sparse scenario also exhibits the largest percentage of time spent packing, representing at least a third of the computation, and the largest difference between its dense counterpart; however, the speed of the computation offsets this impact, still providing the highest speedups.

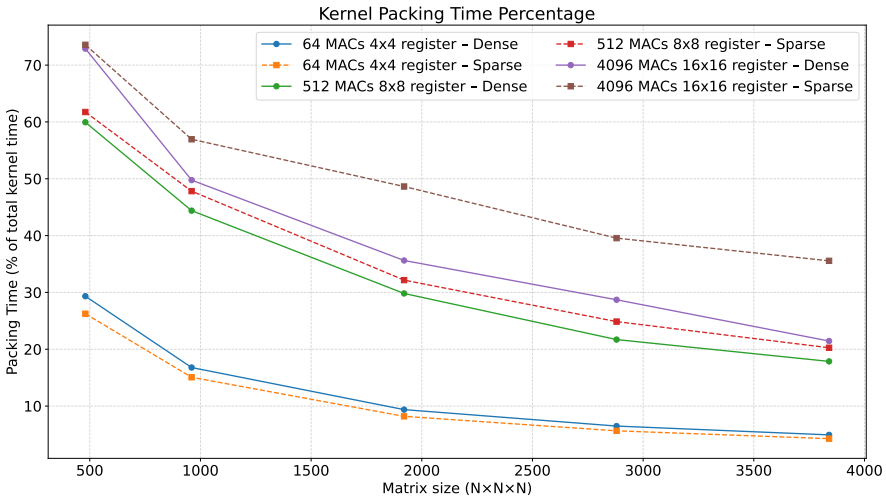

Figure 9. Percentage time for packing in GEMM/SpMM. In all configurations, the percentage of packing in the total time decreases. Packing time is a larger factor in bigger configurations.

Figure 10 shows a roofline representation of the highest performance for each pair of dense and sparse kernels in all three register lengths used, without considering packing. Mlen is the size of the register in bits. This shows the reason for the small speedups of the 64 MACs kernel, as the dense version is already close to peak performance, and adding sparsity just takes a value of 86% of the peak to around 93%. For the 512 MACs case, the increase is more pronounced, going from 70% to 87%. This is closely related to the computational intensity being closer to the DRAM and L2 cache limits. In the 4096 MACs, the limitation is no longer the computations, and it becomes the cache bandwidth. This results in the kernels not being able to achieve performance close to the peak and in a lower percentage increase compared to the 512 MACs case, going from 17% to 25%.

MSE was able to increase performance for the 64, 512, and 4096 MACs configurations, adding minimal hardware to the MPE. While the speedup for the first case was small because the dense version was already able to get close to the peak of computation, the improvements for the other cases show the potential to provide better scaling for the architecture. Another benefit is the capability to work with larger matrix sizes before reaching the limits of the cache, which could allow for smaller packing overhead in the SpMM operation.

8. Conclusions

Hardware to take advantage of structured sparsity is an important component towards improving the memory footprint and inference time of large pruned CNNs. This paper introduces MSE to add support for matrices with 2:4 structure in RISC-V with minimal

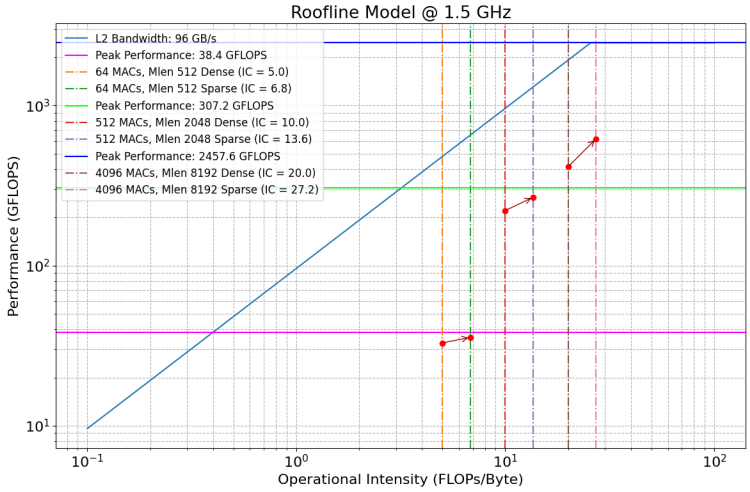

Figure 10. Roofline model for different hardware setups. As MAC units increase, configurations become more memory-bound, decreasing performance relative to the peak and yielding better results for the compressed format.

hardware changes. This is done by using a mask register to store the index matrix and decompressing the data directly into the register file. The evaluation of this extension used a comparison of dense and sparse optimized window kernels that were able to get close to the peak hardware performance (in GFLOPs) in some cases.

While the performance gains were small and steady for smaller hardware configurations, larger matrices demonstrated greater speedups as packing operations started to have a lesser impact on the execution time. This led to better size and compute scaling for the MPE, as the sparse implementations were able to approach maximum efficiency. Future work includes adding support for grouping registers with LMUL-like instructions, enabling the addition of variable sparsity values like 1:4 that would further compress the A matrix and provide an even lower memory footprint.

9. Acknowledgements

This work was supported by the São Paulo State Foundation for Research Support (FAPESP) grant 2024/17401-2.

References

Advanced Micro Devices, Inc. (2024). *"AMD Instinct MI300" Instruction Set Architecture: Reference Guide*. Advanced Micro Devices, Inc., Santa Clara, CA, USA. Version published July 15, 2024.

C. Aquino, I., Wanner, L., and Rigo, S. (2024). *Architectural Simulation with gem5*, chapter 4, pages 92–118. Sociedade Brasileira de Computação, São Carlos, SP.

Cao, S., Zhang, C., Yao, Z., Xiao, W., Nie, L., Zhan, D., Liu, Y., Wu, M., and Zhang, L. (2019). Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 63–72, New York, NY, USA. Association for Computing Machinery.

He, Y. and Xiao, L. (2024). Structured pruning for deep convolutional neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(5):2900–2919.

Intel Corporation (2024). *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Documentation Changes*. Document Number: 355308-003US, Chapter 20: Intel® Advanced Matrix Extensions (Intel® AMX).

Jeong, G., Damani, S., Bambhaniya, A. R., Qin, E., Hughes, C. J., Subramoney, S., Kim, H., and Krishna, T. (2023). Vegeta: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus.

Lin, B., Zheng, N., Wang, L., Cao, S., Ma, L., Zhang, Q., Zhu, Y., Cao, T., Xue, J., Yang, Y., and Yang, F. (2023). Efficient gpu kernels for n:m-sparse weights in deep learning. In *Sixth Conference on Machine Learning and Systems (MLSys'23)*.

Lowe-Power, J., Akram, A., Amin, R., Hill, M. D., Wood, D. A., Chen, D. H., Hsu, L., Krishna, T., Agarwal, N., Wright, A. R., et al. (2020). The gem5 simulator: Version 20.0+. arXiv preprint arXiv:2007.03152.

Maturana, D. and Scherer, S. (2015). Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928.

Mienye, I. D., Swart, T. G., Obaido, G., Jordan, M., and Ilono, P. (2025). Deep convolutional neural networks in medical image analysis: A review. *Information*, 16(3).

Mishra, A. K., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. (2021). Accelerating sparse deep neural networks. *CoRR*, abs/2104.08378.

NVIDIA Corporation (2020). Nvidia a100 tensor core gpu architecture. Technical Report V1.0, NVIDIA Corporation. Whitepaper.

Ta, T., Randall, J., and Batten, C. (2025). Sparsezipper: Enhancing matrix extensions to accelerate spgemm on cpus.

Tang, A., Quan, P., Niu, L., and Shi, Y. (2022). A survey for sparse regularization based compression methods. *Annals of Data Science*, 9(4):695–722.

Tang, Y., Zhao, C., Wang, J., Zhang, C., Sun, Q., Zheng, W. X., Du, W., Qian, F., and Kurths, J. (2023). Perception and navigation in autonomous systems in the era of learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):9604–9624.

Titopoulos, V., Alexandridis, K., Peltekis, C., Nicopoulos, C., and Dimitrakopoulos, G. (2023). Indexmac: A custom risc-v vector instruction to accelerate structured-sparse matrix multiplications.

Titopoulos, V., Alexandridis, K., Peltekis, C., Nicopoulos, C., and Dimitrakopoulos, G. (2025). Optimizing structured-sparse matrix multiplication in risc-v vector processors.

Waterman, A., Asanović, K., and Foundation, R.-V. (2021). The risc-v instruction set manual, volume i: Unprivileged isa. Technical report, RISC-V Foundation. Document Version 20191213.