Power-Efficient Sparse Matrix Multiplication: Column Combining Techniques with Compute-in-Memory Architectures

ECE 598 Group 5
Meng-Shan Wu, Ping-Huai Tseng, Yen-Cheng Lin, Yu-Ting Huang

Abstract—**This project optimizes Convolutional Neural Networks (CNNs) by combining Column Combining and Compute-In-Memory (CIM) techniques to overcome computational and memory challenges. Column Combining restructures sparse matrices into denser forms, significantly improving hardware efficiency, while CIM embeds computational operations directly within memory, minimizing data movement and achieving a power reduction of 87.4%. Column Expansion techniques restore compressed matrices while maintaining over 80% accuracy. This integrated approach enhances performance and energy efficiency, making CNNs highly suitable for deployment in power-constrained environments such as edge and mobile applications.**

I. Introduction and Motivation

Convolutional Neural Networks (CNNs) are a critical component of modern machine learning, particularly for applications such as image recognition, object detection, and data processing. Despite their high effectiveness, deploying CNNs in resource-constrained environments, such as mobile devices and embedded systems, poses significant challenges due to their substantial computational and memory requirements. These demands result in high power consumption and increased latency, limiting their feasibility for real-time and energy-sensitive applications.

To address these challenges, researchers have utilized the inherent sparsity in CNN models. Sparse Convolutional Neural Networks (SCNNs) are constructed by pruning non-essential weights, effectively reducing the model size while maintaining comparable accuracy. However, the resulting sparse matrices often exhibit irregular structures, which complicates efficient hardware implementation due to challenges in memory access patterns and processing efficiency.

This project proposes two primary strategies to enhance the efficiency of sparse CNNs: Column Combining and Compute-in-Memory (CIM). Column Combining is a technique that restructures sparse matrices into denser, more regular forms by grouping and optimizing columns, thereby improving memory access and computational efficiency. Compute-in-Memory (CIM) integrates computation directly within memory units, substantially reducing data movement overhead between processing units and memory, which is one of the major bottlenecks in conventional architectures. By employing these two strategies, this project aims to enhance energy efficiency, making them more suitable for deployment in environments with limited computational resources.

II. Background Information

A. Column Combining

Weight pruning is an extensively studied method for reducing the computational burden of CNNs while preserving high accuracy. By pruning non-essential weights, the network retains only critical connections, resulting in a sparse representation that requires fewer multiplications. Despite the potential for improvement, unstructured sparsity presents challenges for hardware implementation, especially for architectures like systolic arrays, which benefit from regular data flow.

To make sparse CNNs more suitable for efficient hardware execution, Column Combining is employed. This method restructures sparse matrices into a more regular form by merging columns and retraining the remaining weights to maintain accuracy. This structured format is particularly advantageous for specialized hardware such as systolic arrays, which rely on consistent and predictable data movement.

B. Compute-in-Memory

Compute-in-Memory (CIM) is another crucial technique that directly integrates computational operations within memory units. Traditional architectures often suffer from bottlenecks associated with frequent memory accesses, particularly during data-intensive tasks such as convolution. CIM minimizes this overhead by embedding Multiply-Accumulate (MAC) operations directly in memory, thereby reducing latency and power consumption.

The combination of Column Combining and CIM provides a comprehensive solution to the challenges associated with CNN deployment. By using both sparsity and hardware optimization, this approach not only enhances efficiency but also extends the applicability of CNNs to edge devices, where power efficiency and computational performance are crucial.

III. Architecture Overview

Our architecture efficiently addresses the sparsity in convolutional neural networks (CNNs) using a column combining and compute-in-memory (CIM) methodology. The design begins with the column combining process, which consolidates sparse weight kernels into dense representations. This step

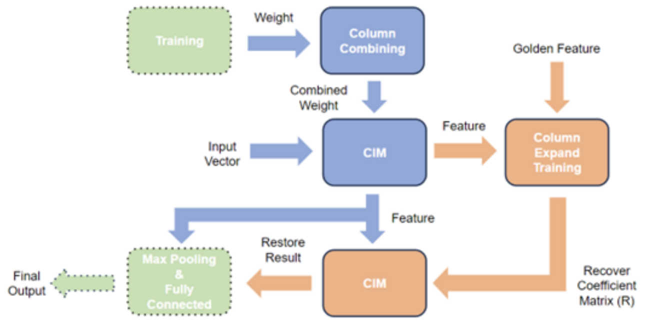

Fig. 1: Architecture

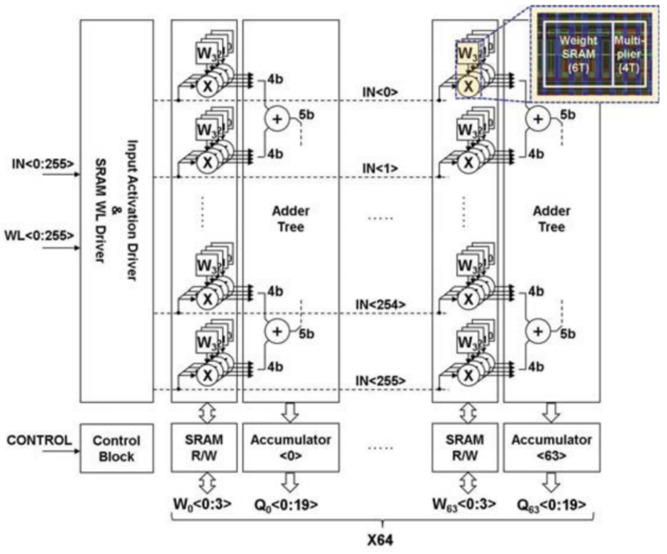

Fig. 2: Architecture of Compute-In Memory

improves computational and memory efficiency by resolving conflicts and pruning smaller weights, while retaining the dominant coefficients. Following the column combination, the weights undergo CIM-based operations. Leveraging a 6T SRAM-based CIM macro, matrix-vector multiplication is executed directly within memory, reducing data movement latency and power consumption. The CIM unit operates in two modes: (1) traditional SRAM storage and (2) multiply-accumulate (MAC) computations. Input vectors are fetched, and partial sums are aggregated within the CIM unit, ensuring high throughput for dense computational workloads. After the CIM stage, the outputs are processed through post-CIM layers, including max-pooling and fully connected layers, to restore features and refine results. The max-pooling layer ensures dimensionality reduction while retaining significant features, and the fully connected layer completes the inference pipeline. To accommodate inaccuracies introduced during compression, the design incorporates a column expansion training process. This iterative step adjusts weights through forward and backward propagation, maintaining model accuracy without compromising sparsity-driven optimizations. Overall, the architecture seamlessly integrates column combining and CIM technologies with modern CNN workflows, delivering a power-efficient solution for sparse matrix computations.

IV. Hardware Implementation

A. High level overview

The Compute-In-Memory (CIM) macro, as shown in Fig. 2, is a 6T SRAM-based all-digital architecture designed for Multiply-Accumulate (MAC) operations within convolutional neural networks (CNNs). The macro operates in two modes: SRAM mode for standard read/write operations, and CIM mode for performing MAC operations.

In Fig. 2, the CIM macro example includes 256 inputs, 64 partial sum outputs, and 256×64 4-bit weights. The macro comprises 64 columns of sub-CIM units. Each sub-CIM unit contains 256 4-bit weights, 256 bit-wise multipliers, a parallel adder tree, a partial-sum accumulator, and an SRAM read/write circuit.

B. Low level details

Figure 3 provides a detailed schematic of the CIM array and the adder tree within a sub-CIM unit. In each cycle of CIM mode, all 256 input activations are simultaneously fed into the CIM array. For 8-bit precision input activations, the sub-CIM unit processes the activations over 8 cycles. During each of these 8 cycles, 256 multiplications are carried out between 1-bit input activations and 8-bit weights. The resulting products are then fed into the adder tree, which generates the partial sum of the 256 multiplication results.

Each CIM cell contains a 6T SRAM cell and a NOR gate, which is used for multiplication. A NOR gate is used instead of an AND gate because it requires only 4 transistors (4T), whereas an AND gate requires 6 transistors (6T). Despite this difference, the NOR gate can effectively perform the same multiplication function.

Figure 5 illustrates the partial-sum accumulator circuit, which is responsible for accumulating the partial sums generated in each cycle in a pipelined manner. The accumulation process requires 8 clock cycles to complete the MAC operation for input activations with 8-bit precision.

V. Column Combing

A. Column Combing Concept

To address sparsity issues in neural networks, we implemented a technique called ”column combining,” as proposed by H.T. Kung et al [1]. This method is designed to enhance the utilization of systolic arrays for sparse Convolutional Neural Networks (CNNs).

Column combining works by merging sparse columns of a CNN filter matrix into denser columns. During this process, if a row contains multiple non-zero weights within the same column group, only the weight with the largest magnitude is retained, while the rest are pruned. To mitigate the accuracy

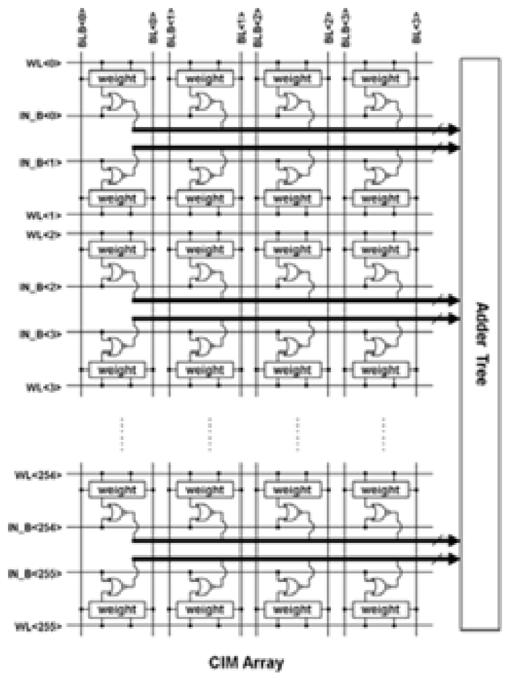

Fig. 3: Schematic view of one sub-CIM unit - CIM Array

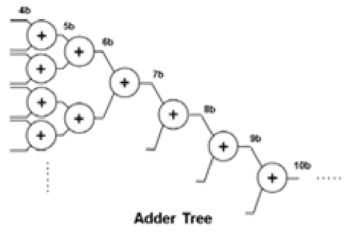

Fig. 4: Schematic view of one sub-CIM unit - Adder tree

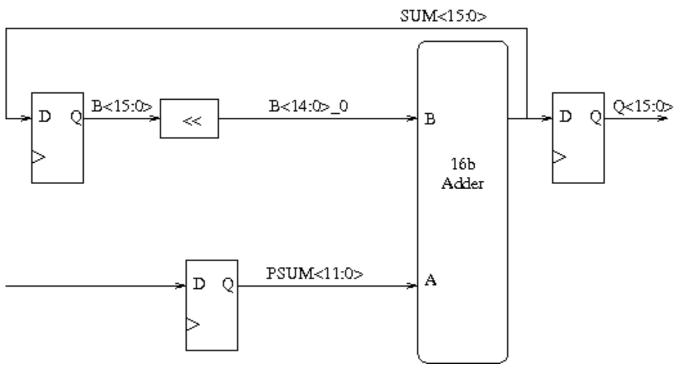

Fig. 5: Schematic view of Accumulator in one sub-CIM unit

loss from pruning, the network is retrained using the remaining non-zero weights.

This technique is guided by parameters such as the maximum number of sparse columns per dense column (α), the pruning schedule (β), and the average number of conflicts allowed per row (γ). To simplify the implementation, we compress an N×N matrix into an N×1 matrix. Higher packing efficiency is achieved by selecting column groups with minimal conflicts and retraining the network to recover accuracy.

Column combining significantly improves hardware utilization efficiency, reduces the number of processing tiles needed for large filter matrices, and results in substantial gains in power efficiency and latency. These optimizations make column combining a highly effective approach for deploying sparse neural networks on hardware accelerators.

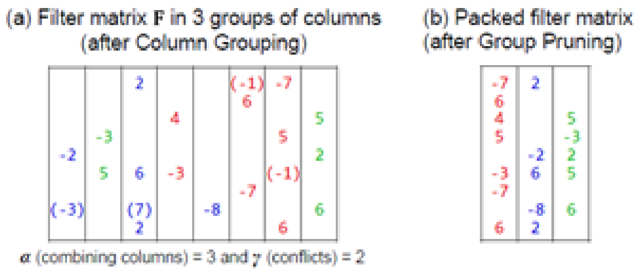
(a) Filter matrix F in 3 groups of columns (after Column Grouping)
(b) Packed filter matrix (after Group Pruning)
α (combining columns) = 3 and γ (conflicts) = 2

Fig. 6: Column Combing proposed by the paper

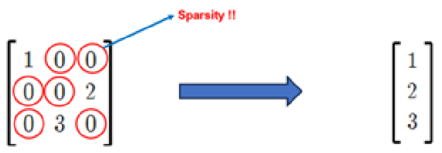

Fig. 7: Example of Column Combing implemented in our design

B. *Algorithm: Column Combing*

- Input:
  - $F \in$ R^{NxN}: Sparse matrix with N rows and N columns.
- Parameters:
  - α: Maximum number of columns to combine per group.
  - γ: Number of conflicts (pruned weights) allowed on per row in each group.
- Output:
  - A single combined column after grouping and pruning.
- Steps:
  1) Initialize:
     - g ←{} : Start with no groups.
     - u ← *1,2,...,M* : Columns not yet grouped.

- *combined_column* : Result column with all rows initialized to 0.

2) Group Columns:
   - Loop:
     - Exit if all columns are grouped (u=ϕ).
     - Calculate pairwise density d (nonzero elements in common).
     - Compute pairwise conflicts o (overlap among nonzero elements).
     - Select the group with the highest density (only one column after combining in our design).
3) Combine Columns:
   - For each row *i* in F:
     - Identify the largest magnitude element max_value from all grouped columns.
     - Set *combined_column[i]* to the maximum value
4) Output:
   - Return the combined dense column.

VI. Column Expansion

A. Introduction: Column Expansion

After applying column combining, the resulting matrix is reduced in size compared to its original dimensions. While this transformation enhances power efficiency by optimizing hardware utilization, it can impact the accuracy of the neural network due to the altered structure. To mitigate this, a balancing approach called **column expand** is introduced.

The objective of column expanding is to restore the compressed result matrix to its original size by multiplying it with a restore matrix. This restore matrix is carefully designed to adjust the combined result back to the dimensions needed for standard matrix multiplication, ensuring compatibility with subsequent layers of the neural network.

The restoration process has two main goals:

1. Maintain the original architecture of the network without requiring modifications to downstream layers.

2. Ensure that the restored matrix elements closely approximate the results produced by a standard matrix multiplication operation.

By achieving these goals, the column expand technique allows for efficient hardware utilization without compromising the integrity of the model's predictions. The restore matrix is constructed through training to minimize the error between the restored and original results, ensuring that the restored values closely match the expected outputs and thereby preserving the accuracy of the neural network.

Furthermore, if the layer following the convolution layer is a max-pooling layer, we can incorporate the max-pooling operation into our training model. Results indicate that doing so enhances the accuracy of the restoration procedure, as the max-pooling effect is directly considered during training.

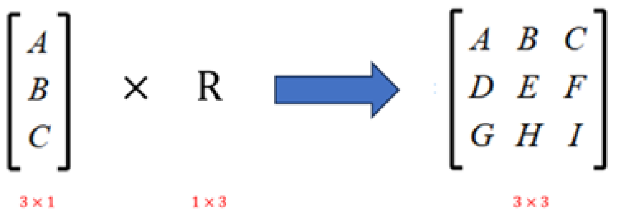

Fig. 8: Column Expansion

B. Algorithm: Combing Expansion

Using 4x4 matrix as example :

- Input:
  1) *no_expand_kernel* : a 4x4 kernel matrix used for reference.
  2) *combining_kernel* : A 4x1 kernel matrix used for compressed operations.
  3) *input_matrices* : A set of 50 random 4x4 input matrices with integer elements between 0 and 256.
  4) *regularization_lambda* : Regularization strength.
  5) *iterations* : Number of iterations for optimization.
- Output:
  - Optimized restore matrix *R*, which restores compressed outputs back to their original form.
  - Comparison between pooled (optimization option) restored results and target results.
- Steps:
  1) Initialize Variables :
     - Generate 50 random 4x4 input matrices (*input_matrices*).
     - Compute the target outputs *Y_target* using matrix multiplication with *no_expand_kernel*.
     - Compute the compressed outputs *Y_combined* using matrix multiplication with *combining_kernel*.
     - Apply 3x3 max pooling on *Y_target* to get *Y_target_pooled*.
  2) Define Optimization Variables:
     - Initialize restore matrix *R* with random values from a normal distribution.
  3) Define Loss Function:
     - During the first **70%** of iterations:
       - Use the full matrix loss. loss = $|Y_{target} - Y_{expanded}|^2$
     - During the last **30%** of iterations:
       - Apply 3x3 max pooling on both $Y_{expanded}$ and Y_{target} before calculating the loss. loss = $|Y_{target} - Y_{expanded}|^2$
  4) Optimization Loop:
     - For each iteration i (up to iterations) :
       - Perform forward propagation to compute the expanded outputs $Y_{expanded} = Y_{combined} \times R$.
       - If i <70% iterations :

Calculate loss using the full matrix outputs.
  * Else :
    Calculate loss using the pooled matrix outputs.
  * Compute gradients of the loss with respect to R.
  * Adjust gradients to prevent large updates.

5) Post-Optimization:
   - Retrieve the optimized restore matrix $R_{optimized}$.
   - Use $R_{optimized}$ to restore the compressed outputs $Y_{expandedoptimized}$.
   - Apply 3x3 max pooling on $Y_{expandedoptimized}$.

6) Validation:
   - Compare the pooled restored results $Y_{expandedoptimizedpooled}$ with the target results $Y_{targetpooled}$

VII. Results and Analysis

A. Verilog Behavior Model

In the waveform (Fig. 9), this CIM system is composed of 4 sub-CIM units, each containing 4 sets of 8 SRAM cells, representing 8-bit weights. Initially, the system starts in SRAM mode (mode = 0), which allows weights to be written into the SRAM cells. During the first cycle, weights are written into the 4 sub-CIM units in the first row, with address A = 0 and weight W = 81 182 245 85. In the following three cycles, weights are written into the 4 sub-CIM units in the second, third, and fourth rows, with addresses A = 1, 2, 3, respectively, and weights for each of the sub-CIM units.

After the weights are written into the CIM, the system switches to CIM mode (MODE = 1). The first bit of the 4 input (in = 1000) is provided, followed by the remaining 7 bits of the input over the next 7 cycles. The product of the input and weight values is processed through the adder tree and accumulator, with a precision equal to the sum of the input precision, weight precision, and the depth of the adder tree (8 + 8 + 2 = 18 in this case). In the waveform, the result can be observed in Q_CIM, where the operation yields 215*81 + 82*205 + 224*14 + 12*219 = 39989 for Q_CIM[0].

B. HSPICE simulation

We simulated a compute-in-memory (CIM) array, as illustrated in Fig. 10, consisting of four rows, each containing eight CIM cells. Each CIM cell integrates a 6T SRAM cell and a NOR gate, serving as both the storage and logic components of the system. Together, these cells represent four 8-bit weights and support the processing of four 8-bit input values for multiplication. The entire configuration, including precharge, weight writing, and multiplication operations, was evaluated using HSPICE simulations. The corresponding SPICE model includes the precharge circuits and write circuits depicted in Fig. 11 and Fig. 12, respectively.

C. Performance & Synthesis results

For performance evaluation, the multiplication of a 4x4 matrix with 8-bit input precision and a 4x1 weight matrix with 8-bit precision is completed in 8 cycles. This latency is determined solely by the input precision rather than the matrix dimensions. Expanding the input or output matrix size will lead to an increase in the area requirements, but it will not impact the overall latency of the operation.

According to the synthesis results, the hardware design achieves a 1.5 ns clock period . The SRAM cell area, referencing values reported in [1], is 0.379 µm² for 1 6T SRAM cell. In the column-combined configuration, the adder tree and accumulator occupy 538.55 µm², whereas without column combining, their area increases to 2136.08 µm².

This table (Table I) presents the power consumption of an 8-bit input matrix multiplication using a 4x1 column-combined 8-bit weight matrix compared to a conventional 4x4 8-bit weight matrix. The results demonstrate a substantial reduction in power when employing the column combining technique, achieving an 87.4% decrease in total power consumption relative to the original configuration without column combining.

TABLE I: Power estimation

	With Column Combine	Without Column Combine
CIM - SRAM cell	210.02	1722.24
Adder tree & Accumulator	12.62	49.10
Total (μW)	222.64	1771.34

D. Column Expand

The column expansion results indicate that training the restore matrix successfully recovers the original matrix size while maintaining accuracy. The mixed training method outperforms the whole matrix training, as demonstrated by a steady decrease in total loss over time (Fig. 13). The restore matrix effectively compensates for the size reduction caused by column combining, highlighting the effectiveness of column expansion in recovering compressed matrices with high accuracy. By using both column combining and expansion, the number of required multiplications is reduced from 64 to 32 compared to traditional 4x4 matrix multiplication.

A random 4x4 weight matrix with a certain level of sparsity was generated, compressed to a 4x1 matrix using the column combining algorithm, and a 1x4 restore matrix was trained through column expansion. The results were then compared to those of normal matrix multiplication, as shown in the table. The table (Table II) illustrates the accuracy achieved after multiplying the input matrix by both the column combined matrix and the restore matrix at different sparsity levels. Despite varying sparsity, the column combining and restoration process effectively maintains high accuracy, demonstrating that this approach can preserve performance even with significant matrix compression.

VIII. Conclusion

In this project, we introduced an efficient hardware architecture utilizing Column Combining and Compute-in-Memory (CIM) techniques to optimize convolutional neural networks (CNNs) for resource-constrained environments. Our design

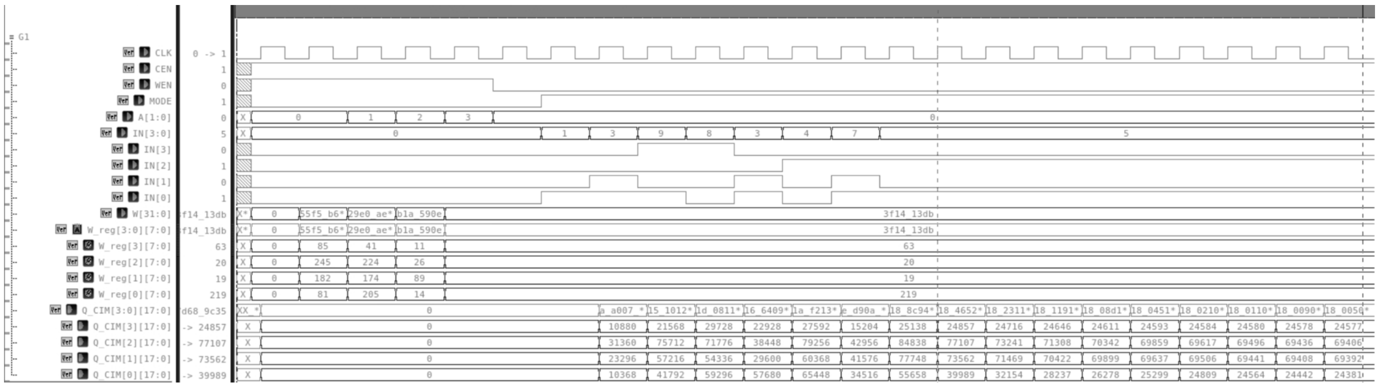

Fig. 9: Behavior Simulation Waveform

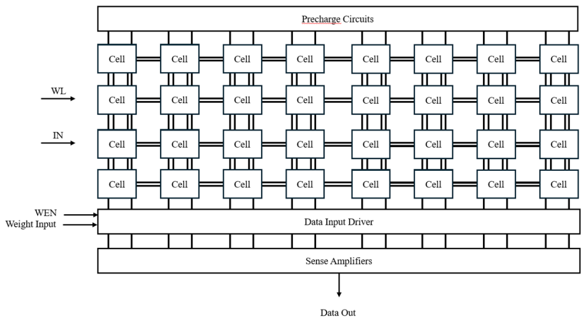

Fig. 10: CIM array

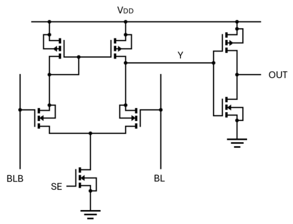

Fig. 12: Sense Amplifier

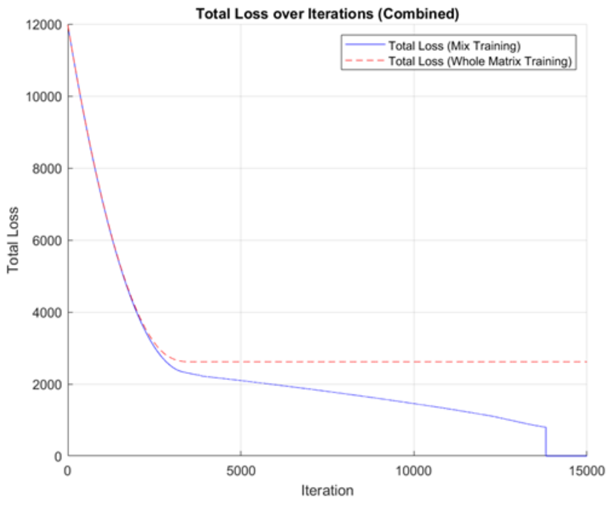

Fig. 13: Column Expansion Training

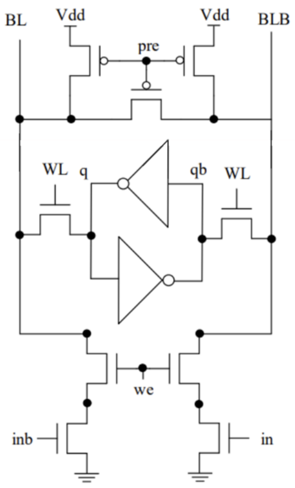

Fig. 11: Cell ciruit with precharge and write circuit

successfully combines sparse matrix representation through column combining and integrated memory computation to reduce energy consumption and improve performance. The experimental results demonstrated significant gains in power efficiency and preserved accuracy with the combined column expansion approach.

By applying Column Combining, we transformed sparse matrices into denser forms that are more suitable for hardware accelerators, such as systolic arrays. Furthermore, Compute-in-Memory operations minimized data transfer enhancing energy efficiency. The column expansion strategy ensured that the accuracy loss from pruning was effectively mitigated by re-training and restoring dimensions, preserving model integrity.

Overall, our implementation illustrates how merging sparsity techniques with specialized hardware can make CNNs feasible for edge devices with strict power budgets, offering a pathway toward scalable and efficient deep learning in mobile

TABLE II: Accuracy under different sparsity

Sparsity	Accuracy
50%	86.264%
60%	82.906%
70%	81.528%
80%	83.358%
90%	88.236%

and embedded systems.

IX. Contribution

- Column Expand - Novel idea
- Column Combine referencing from ”Packing sparse convolutional neural networks for efficient systolic array implementations” [1]
- CIM Architecture referencing from ”16.4 An 89TOPS/W and 16.3TOPS/mm2 All-Digital SRAM-Based Full-Precision Compute-In Memory Macro in 22nm for Machine-Learning Edge Applications” [4]
- All modules are designed from scratched

CIM		
Behavioral Model Implementation & Synthesis	Adder Tree & Accumulator Implementation & Synthesis	SRAM cell HSPICE Simulation
Meng-Shan Wu Yen-Cheng Lin	Ping-Huai Tseng Yu-Ting Huang	Meng-Shan Wu Yu-Ting Huang

Column Combine	Column Expand	
Algorithm	Algorithm	Restore Matrix Training
Ping-Huai Tseng Yen-Cheng Lin	Meng-Shan Wu Yen-Cheng Lin	Ping-Huai Tseng Yu-Ting Huang

References

[1] Kung, H., McDanel, B., and Zhang, S. Q. (2019). Packing sparse convolutional neural networks for efficient systolic array implementations. ASPLOS’19, 821–834. https://doi.org/10.1145/3297858.3304028

[2] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., . . . Yoon, D. H. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. ISCA’17. https://doi.org/10.1145/3079856.3080246

[3] Lin, C., Hong, D., Liu, P., Wu, J. (2023). Accelerate Inference of CNN Models on CPU via Column Combining Based on Simulated Annealing. CANDAR’23, 28, 20–29. https://doi.org/10.1109/candar60563.2023.00011

[4] Y. -D. Chih et al., ”16.4 An 89TOPS/W and 16.3TOPS/mm2 All-Digital SRAM-Based Full-Precision Compute-In Memory Macro in 22nm for Machine-Learning Edge Applications,” 2021 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 2021, pp. 252-254, doi: 10.1109/ISSCC42613.2021.9365766.

[5] Zhang, Z., Chen, J., Chen, X., Guo, A., Wang, B., Xiong, T., Kong, Y., Pu, X., He, S., Si, X., Yang, J. (2023). From macro to microarchitecture: reviews and trends of SRAM-based compute-in-memory circuits. Science China Information Sciences, 66(10). https://doi.org/10.1007/s11432-023-3800-9.

[6] Wang, Y., Tu, F., Liu, L., Wei, S., Xie, Y., Yin, S. (2022b). SPCIM: Sparsity-Balanced Practical CIM accelerator with optimized Spatial-Temporal Multi-Macro utilization. IEEE Transactions on Circuits and Systems I Regular Papers, 70(1), 214–227. https://doi.org/10.1109/tcsi.2022.3216735