

# Efficient In-Memory Acceleration of Sparse Block Diagonal LLMs

João Paulo C. de Lima<sup>1,2</sup>, Marc Dietrich<sup>1</sup>, Jeronimo Castrillon<sup>1,2</sup>, and Asif Ali Khan<sup>1</sup>

<sup>1</sup>Chair for Compiler Construction, TU Dresden, Germany

<sup>2</sup>Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden, Germany  
Corresponding authors: {joao.lima, asif\_ali.khan}@tu-dresden.de

**Abstract**—Structured sparsity enables deploying large language models (LLMs) on resource-constrained systems. Approaches like dense-to-sparse fine-tuning are particularly compelling, achieving remarkable structured sparsity by reducing the model size by over  $6.7\times$ , while still maintaining acceptable accuracy. Despite this reduction, LLM inference, especially the decode stage being inherently memory-bound, is extremely expensive on conventional Von-Neumann architectures. Compute-in-memory (CIM) architectures mitigate this by performing computations directly in memory, and when paired with sparse LLMs, enable storing and computing the entire model in memory – eliminating the data movement on the off-chip bus and improving efficiency. Nonetheless, naively mapping sparse matrices onto CIM arrays leads to poor array utilization and diminished computational efficiency. In this paper, we present an automated framework with novel mapping and scheduling strategies to accelerate sparse LLM inference on CIM accelerators. By exploiting block-diagonal sparsity, our approach improves CIM array utilization by over 50%, achieving more than  $4\times$  reduction in both memory footprint and the number of required floating-point operations.

**Index Terms**—Structured sparsity, Large language models, Computing-in-memory

## I. INTRODUCTION

Pre-trained transformer models, with billions of parameters, demand substantial computational and memory resources, making them impractical for resource-constrained devices [1, 2]. Structured sparsity significantly reduces these requirements, enabling potential deployment on such systems [3, 4]. Dense-to-sparse fine-tuning transforms pre-trained dense models into sparse models with comparable accuracy, achieving up to  $4\text{--}8\times$  compression depending on the structure and task [3, 4]. However, sparsity alone is not sufficient for efficient execution. Hardware-level inefficiencies, especially in exploiting the block-structured sparsity, remain a critical bottleneck to achieving performance gains.

Even with reduced model sizes, transformer inference, particularly memory-bound in the decoding phase, incurs high energy costs due to data movement in conventional von Neumann systems, consuming over 62% of total energy [5]. Analog compute-in-memory (CIM) accelerators address this by performing computations directly within memory, leveraging crossbar arrays where memory cells serve as both storage and computational elements [6, 7, 8]. These arrays enable constant-time matrix-vector multiplication (MVM) in-place, exploiting the analog properties of memory devices to perform operations without data transfer. This eliminates the von Neumann bottleneck and drastically reduces energy and latency [9].

However, on resource-constrained devices, the large GEMM sizes in transformer models still pose challenges, requiring sparsification to make them fit in limited CIM array capacities. Structured approaches such as Monarch matrices [4] replace dense weight matrices in components like attention and feedforward layers with structured, sparse representations. Monarch matrices, which are products of two block-diagonal matrices (up to permutation), enable sub-quadratic multiplication and can represent common operations such as FFTs and convolutions [10]. Compared to unstructured sparsity, this structured form offers both computational advantages and expressiveness. Nonetheless, mapping these block-diagonal matrices efficiently onto CIM arrays is non-trivial. Naive mappings often lead to severe underutilization of array capacity, resulting in performance and energy inefficiencies. Addressing this challenge requires a careful co-design of sparsity patterns, data layout, and memory-aware mapping strategies to fully realize the benefits of sparse models on CIM accelerators.

To this end, we propose an automated framework that transforms the dense layers of transformer networks into structured sparse representations and efficiently deploys them onto analog CIM designs. Specifically, our contributions are:

- **CIM-aware sparse mapping:** We propose latency- and capacity-optimized mapping strategies to densely pack block-diagonal structured matrices onto CIM arrays. These strategies reduce array fragmentation and enable selective activation of crossbar rows and columns, thus reducing execution time and energy consumption by requiring fewer and lower-precision analog-to-digital converters (ADCs) (Sec. III-B).
- **Performance-aware scheduling:** Our scheduling module selectively activates rows within the crossbar arrays, intelligently balancing ADCs sharing and parallelism, leading to significant reductions in energy consumption and inference latency (Sec. III-C).
- **End-to-end automation:** Our framework integrates dense-to-sparse (D2S) transformation, crossbar mapping, and scheduling into an automated toolchain, targeting analog CIM accelerators (Fig. 2).

We conduct design space exploration by analyzing how different mappings requiring varying DA/AD converter precisions and sharing degrees affect performance and energy consumption, and validate our strategies across multiple models. Our evaluations, compared to dense models on the CIM baseline, show that our framework reduces execution time and energy

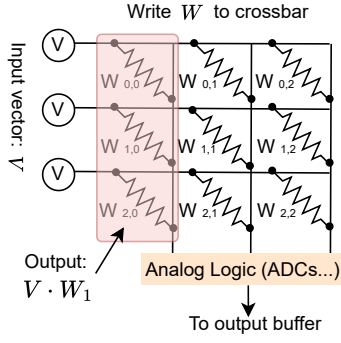


Fig. 1: Dot product on an analog CIM crossbar

consumption by over  $1.7\times$ , while also simultaneously reducing the memory footprint by over  $4\times$ .

## II. BACKGROUND

This section provides background on analog CIM, transformer networks, and structured sparsity.

### A. Analog Computing-in-Memory

Analog CIM leverages the physical properties of memory devices to multiply an input vector  $V$  with a matrix  $W$  directly within memory arrays. This can be implemented using nonvolatile memory (NVM) as well as SRAM technologies. Fig. 1 shows a typical NVM-based CIM crossbar, where the matrix  $W$  is programmed into the memory cells, and the input vector  $V$  is applied as voltages across the rows (wordlines). The MVM is performed via Ohm's and Kirchhoff's laws: the accumulated current on each vertical line (bitline) corresponds to the dot product of  $V$  with the respective column of  $W$  (e.g., column  $W_1$  is highlighted in red).

SRAM-based analog CIM follows similar principles but is implemented in different ways, such as current [11], charge-based [12], or time-based [13]. Regardless of the underlying technology, analog CIM follows a weight-stationary dataflow consisting of the following steps: (1) converting digital inputs into analog signals using digital-to-analog converters (DACs), typically bit-streamed over multiple cycles; (2) performing multiplication in the analog domain ( $V \cdot W$ ); (3) summing the partial products in the analog domain; (4) converting the analog results back to digital using ADCs; and (5) accumulating outputs from multiple crossbars using shift-and-add circuits.

ADC scalability remains a major bottleneck in analog CIM, often accounting for up to 60% of energy and nearly 80% of area in CIM accelerators [14]. Due to their large footprint, ADCs are typically shared across multiple columns, which limits the overall throughput of the crossbar. Successive Approximation Register (SAR) ADCs are commonly used in CIM designs; they allow precision to be dynamically adjusted by changing the number of comparison steps during conversion [15]. Reducing the number of conversion steps lowers resolution and power consumption. By intelligently leveraging model compression techniques, especially block-diagonal sparsity patterns, it becomes possible to aggressively share ADCs across inactive columns and reduce the required

ADC precision. Lower ADC precision directly translates to reduced energy consumption and smaller area footprint, thereby improving energy and area efficiency of CIM accelerators [16].

### B. Transformer Networks

Transformer models are built upon the self-attention mechanism, which enables them to capture long-range dependencies and contextual relationships between tokens (e.g., words) in a sequence (e.g., a sentence). At the heart of this mechanism is the scaled dot-product attention (SPDA), which operates on three sets of vectors: query ( $q$ ), keys ( $\mathbf{K}$ ), and values ( $\mathbf{V}$ ). The attention output is a weighted sum of the value vectors, where the weights are determined by the similarity between the query and each key vector. This similarity is computed as the dot product between  $q$  and each key  $k_j \in \mathbf{K}$ , scaled by the square root of the key dimensionality  $d_k$ :

$$\text{Attention}(q, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{q\mathbf{K}^\top}{\sqrt{d_k}} \right) \mathbf{V}.$$

To improve the model's ability to attend to information from different representation subspaces, Transformer architectures employ multi-head attention (MHA). Instead of performing a single attention operation, MHA projects the input queries, keys, and values into multiple lower-dimensional subspaces using learnable projection matrices  $\mathbf{W}_i^Q$ ,  $\mathbf{W}_i^K$ , and  $\mathbf{W}_i^V$  for each attention head  $i$ . The SPDA operation is then applied independently in each head, allowing the model to jointly attend to information from different perspectives. The outputs from all heads are subsequently concatenated and linearly transformed to form the final output [17]. Following the attention mechanism, each Transformer layer includes a feed-forward network (FFN) composed of two linear transformations with a nonlinear activation in between.

### C. Monarch Factorization

Structured matrix factorizations have been well studied for D2S transformations, with Butterfly factorizations [18, 19] offering compact representations of linear operators through hierarchical and diagonal sparsity. These enable efficient MVM and approximation with reduced time and space complexity. Building upon this foundation, the Monarch framework [4] introduces a structured matrix class that expresses dense weight matrices as a product of two block-diagonal matrices interleaved with fixed permutations. This structure is highly expressive, hardware-efficient, and capable of approximating dense matrices with minimal loss in representational fidelity.

In its most general form, an order- $p$  Monarch matrix is structured as a product  $M = \prod_{i=1}^p (P_i B_i) P_0$ , where  $M \in \mathbb{R}^{n \times n}$ ,  $P_i$  are alternate permutations, and  $B_i$  are block-diagonal matrices [20]. In practice, existing works have focused on factorizations with two block-diagonal matrices (i.e.,  $p=2$ ) composed of blocks of size  $b \times b$ , where  $b = \sqrt{n}$  [4, 20]. This yields a subquadratic matrix multiplication complexity of  $\mathcal{O}(pn^{(p+1)/p})$ , which is highly expressive for common transforms (e.g., it generalizes FFT) and sufficiently expressive for attention and FFN layers.

In addition to algorithmic efficiency, Monarch matrices are hardware-friendly: their structure decomposes larger GEMMs

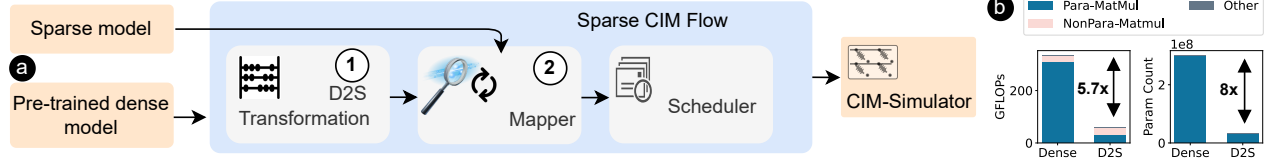


Fig. 2: (a) Overview of the proposed framework; (b) the expected reduction in FLOPs and parameter count for BERT-large.

into smaller GEMMs, which lend themselves well to tensor cores in modern GPUs. This paper explores their execution on CIM accelerators, where the fine-grained block structure of Monarch matrices is leveraged for improved mapping, parallelism, and energy efficiency.

### III. MONARCH MATRICES ACCELERATION USING CIM

Fig. 2a illustrates the overall framework: starting from a pre-trained dense model, the dense-to-sparse (D2S) transformation generates structured sparse matrices, which are then mapped onto CIM arrays. The scheduler orchestrates the execution by issuing memory commands that balance ADC sharing and parallelism, optimizing the trade-off between energy and latency. For models that have already been trained or fine-tuned with Monarch sparsity, the D2S step is not required. On BERT-large with 512-token input sequences, D2S reduces parameters by  $8\times$  and FLOPs by  $5.7\times$  compared to the baseline (Dense), as illustrated in Fig. 2b. The following sections provide detailed explanations of each step in the framework.

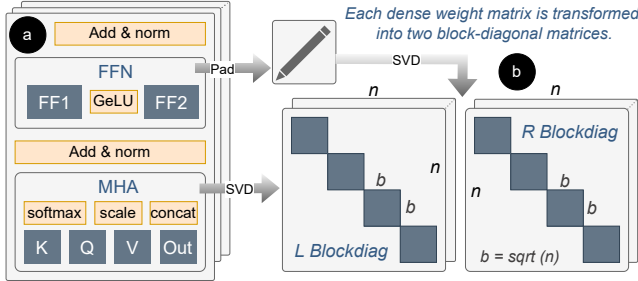


Fig. 3: D2S transformation, (a) dense model (b) sparse.

#### A. Dense-to-sparse (D2S) Transformation

To enable D2S transformation without retraining, authors in [4] propose an analytical method to approximate a dense matrix  $W \in \mathbb{R}^{n \times n}$  with a Monarch-structured matrix  $M \in \mathbb{R}^{n \times n}$  by solving an optimization problem that minimizes the Frobenius norm  $\|W - M\|_F$ . This is achieved by reshaping  $W$  into slices that match the Monarch block structure and applying rank-1 singular value decomposition (SVD) to obtain the factor matrices. The resulting decomposition,

$$M = P \cdot L \cdot P \cdot R \cdot P, \quad (1)$$

use block-diagonal matrices  $L$  and  $R$  derived from SVD, and a fixed permutation matrix  $P$ . This structured approximation preserves the accuracy of the original matrix while

significantly improving memory and compute efficiency. An overview of the factorized matrices  $L$  and  $R$  is shown in Fig. 3.

As shown, we apply the D2S transformation only to parameterized matmuls, such as the linear projections in the MHA and FFN blocks (*Para-Matmul* in Fig. 2b). In contrast, non-parameterized matmul (*NonPara-Matmul*), such as those used to compute attention scores and apply attention weights, operate solely on activation tensors and remain untransformed.

#### B. Mapping Sparse Matrices onto CIM Arrays

As discussed in Sec. II-A, MVM in CIM typically involves pre-storing the weight matrix  $W$  within the memory array, while the input vector  $X$  is encoded as input voltage and applied to all array rows. After applying the D2S transformation described in Sec. III-A, the MVM operation transforms from  $X \cdot W$  to  $X \cdot M$ , where  $M$  is a Monarch-structured matrix defined in Eq. 1. This requires mapping the resulting block-diagonal sparse matrices ( $L, R$ ) onto CIM arrays.

Let the CIM array have dimension  $m \times m$ , and assume that each block in the  $L$  and  $R$  matrices is of size  $b = \sqrt{n}$ . Consider a transformer model with  $N$  layers, each containing  $p$  parameterized matrix multiplications. The mapping problem then reduces to placing  $N \cdot p$  Monarch matrices onto the available CIM arrays of size  $m \times m$ .

In the following, we present two mapping strategies: one optimized for performance, and another optimized for crossbar utilization and energy efficiency. For simplicity, we demonstrate and describe the mapping of a single matrix, though this applies to all matrices.

1) *Performance Optimized Mapping*: In this approach, we naively map each block-diagonal matrix to one or more CIM arrays without considering the memory utilization. Given Monarch matrices with block size  $b$  and CIM arrays of size  $m \times m$ , the  $L$  and  $R$  matrices are partitioned based on  $m$ , and each partition, comprising one or more blocks, is assigned to a separate CIM array. If  $b = m$ , this results in a one-to-one mapping, fully utilizing the array capacity. However, in practice,  $b \ll m$ , which allows packing multiple blocks per array, but still leads to significant underutilization. For instance, in Fig. 4a, when  $m = 2b$ , two blocks can be packed into a single array, with remaining cells zero-padded. This enables parallel execution of blocks since the padded zeros do not impact correctness. Nonetheless, it comes at the cost of memory underutilization, which is especially a critical concern in resource-constrained systems.

In general, the total number of CIM arrays required to store the  $L$  and  $R$  matrices is approximately  $n/m$ . The effective utilization of each array, defined as the ratio of non-zero (valid)

entries to total array capacity, is given by  $(b/m) \times 100\%$ . For example, with  $n = 1024$ ,  $m = 256$ , and  $b = 32$ , the effective array utilization is only  $(32/256) \times 100\% = 12.5\%$ , meaning that 87.5% of the array remains unused due to zero-padding.

This mapping also implicitly assumes that the number of CIM arrays in the system is larger than the required number of arrays, which may not be the case in resource-constrained systems. For systems with a limited number of CIM arrays, this mapping requires rewriting array data (swapping it with new data) dynamically during execution, which incurs significant overhead, especially in NVM-based CIM systems. We next introduce a capacity-optimized mapping strategy that densely packs the sparse block-diagonal matrices in CIM arrays to maximize array utilization and minimize write overhead.

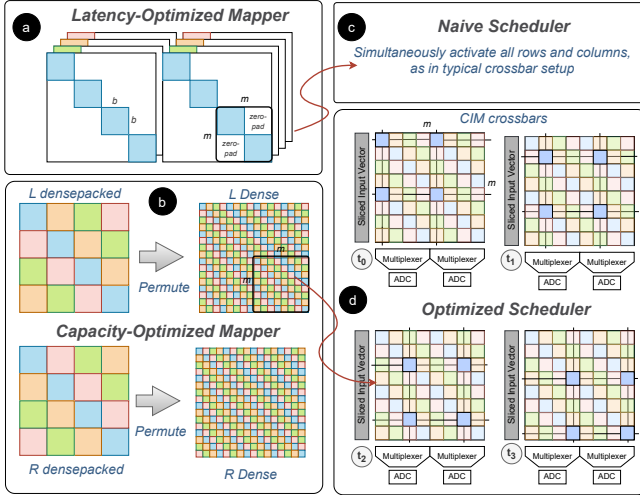


Fig. 4: Proposed mapping and scheduling strategies.

2) *Capacity Optimized Mapping*: This mapping strategy leverages the structured sparsity inherent in Monarch matrices to densely pack multiple block-diagonals into a single CIM array. Given that Monarch-structured matrices reduce parameter count by a factor of  $\sqrt{n}$ , it is possible to store up to  $\sqrt{n}$  block-diagonal matrices in a single  $m \times m$  CIM array. As illustrated in Fig. 4b, the densely packed arrays for  $L$  and  $R$  accommodate four block-diagonals each. These diagonals may correspond to different parameterized operations within a transformer layer or to partitions of a single large matrix that has been partitioned to match array dimensions. This maximizes array utilization, approaching 100% when  $m$  is a multiple of  $b$ , and substantially reduces the number of CIM arrays needed for in-place inference.

a) *Handling Rotations and Shifts*: Each block-diagonal in our densely packed CIM array is assigned a unique diagonal index  $i$ , indicating its relative position within the array. During the MVM operation, this indexing introduces block-wise cyclic rotations in the output vector. Specifically, a diagonal at index  $i$  produces an output that is rotated by  $i$  positions. For instance, in Fig. 5a, the input vector (represented in gray) multiplied by the block-diagonal at index 0 (blue color) produces results that do not require any rotation. However, the

same vector applied to the diagonal at index 1 (brown color) produces results that need to be rotated left by one to produce the correct final result.

Given the two-stage structure of Monarch matrices, first multiplying with  $L$ , then with  $R$ , it is possible to exploit rotational symmetry to cancel out these rotations. By carefully choosing the diagonal index  $i_R$  for each block in  $R$  such that:

$$i_R = -i_L \mod b,$$

the rotation introduced by the first stage ( $L$ ) is effectively neutralized in the second stage ( $R$ ). This reduces the need for costly post-processing or rotation correction operations and enables more efficient scheduling on the CIM hardware.

To exploit this symmetry, the blocks in the  $R$  stage need to be *shifted* to align with the rotational offsets from the  $L$  stage. As shown in Fig. 5c, the output blocks of  $L$  must be routed or reassigned to the appropriate diagonals in  $R$  based on the rotation index  $i_L$ . This reassignment ensures that the composition  $R \cdot L$  yields a correctly aligned output vector.

Special care is required for diagonal indices  $i = 0$  and  $i = b/2$ , as these indices are self-inverses under modulo  $b$  and thus cannot be paired within the same dense matrix without violating the symmetry constraint. These special cases must be distributed across different Monarch matrices or carefully managed to avoid redundant or conflicting assignments.

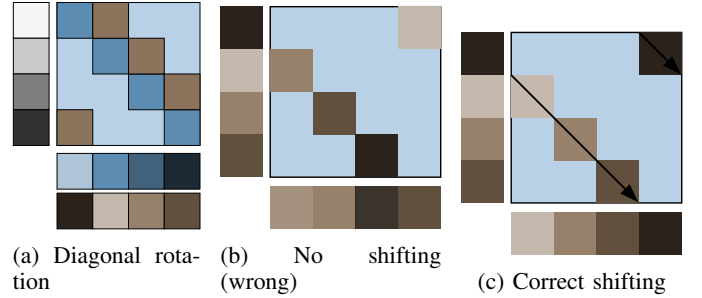


Fig. 5: An example to show why rotations (a) and shifting are needed (b, c)

3) *Folding Permutations into the Matrix Structure*: Recall from Sec. III-A that  $M = P \cdot L \cdot P \cdot R \cdot P$ , i.e., in addition to the  $L, R$  multiplication, the permutation matrix  $P$  must also be applied three times in correct order to produce the correct result. However, these inherent permutations in the Monarch structure can be embedded directly into the matrix structure itself to eliminate the need for explicit permutation operation. The conventional monarch matrix requires three separate permutation operations surrounding the sparse block-diagonal matrices. However, by algebraic rearrangement, this can be rewritten as:

$$M = (PLP) \cdot P \cdot (PRP),$$

which folds the outer permutations directly into the  $L$  and  $R$  matrices. This transformation reduces the number of permutation steps from three to just one, and can be performed offline before mapping the  $L, R$  matrices to CIM arrays.

More importantly, CIM arrays typically have fewer ADCs than the number of bitlines, i.e., ADCs are shared across



groups of bitlines, and an input vector is multiplied in the CIM array over multiple cycles. The transformed  $M$  structure naturally aligns with this multi-cycle behavior by enabling multiplexed ADC access, thereby improving throughput.

### C. The Scheduling Module

For the latency-optimized mapping (Sec. III-B1), all blocks can be computed in parallel by simultaneously activating all rows and columns of the CIM array, as in a typical crossbar operation. In contrast, the capacity-optimized mapping strategy presented in Sec. III-B2 requires the scheduler to issue mapping-aware CIM instructions to ensure correct execution. Naively, activating all rows and columns in this mapping would lead to incorrect results, as each column stores data from multiple block-diagonal matrices.

To illustrate this, consider the highlighted  $m \times m$  partition (where  $m = 8$ ) of the  $L$  matrix in Fig. 4b, which is mapped to a CIM array as shown in Fig. 4d. Note that the figure shows the same array state at four different timestamps. Based on the diagonal indices of the block-diagonal matrix, the computations are temporally scheduled. At time  $(t_0)$ , only two rows and two columns are activated, corresponding to four active cells (highlighted in blue). At  $(t_1)$ , the next four cells along the same diagonal—shifted down by one position—are computed. This pattern continues so that the subsequent elements along the diagonal are processed at  $(t_2)$  and  $(t_3)$ , completing the computation for one diagonal. The same procedure is repeated for the remaining diagonals in the array.

The scheduler, aware of the underlying memory mapping and block-diagonal sparsity, automatically generates the correct memory addresses and issues the appropriate control commands to execute the matrix multiplication accurately. It is important to note that, while computations within a single CIM array are performed sequentially due to temporal scheduling, all CIM arrays operate in parallel. Since the array dimensions are typically much smaller than the matrix dimensions, each array produces partial results that are subsequently combined to compute the final output.

## IV. EXPERIMENTAL SETUP AND EVALUATION

This section describes our experimental setup, benchmarks, and a discussion of the results.

**Benchmarks:** To evaluate our proposed framework, we use the parameterized matmults in three different transformer models: BERT-large [2], BART-large [21], and GPT-2-Medium [1]. BART is an encoder-decoder model, while GPT-2 and BERT are decoder-only and encoder-only transformer architectures, respectively. The context lengths are 512 for BERT, and 1024 for both BART and GPT-2.

**Simulated system:** We use the open-source simulator from [22], which models 2D and 3D analog in-memory computing (AIMC) accelerators with configurable architectural parameters and execution flows. The simulator supports transformer model deployment across weight-stationary CIM arrays for MVM operations, digital processing units (DPUs) for auxiliary operations, and dedicated multi-head attention (MHA) units. It faithfully models data movement and scheduling costs, and enables energy and latency estimation. Since

the monarch factorization only targets parameterized matmults, which constitute over 80% of FLOPs (see Fig. 2b), we specifically focus on the performance of parameterized ones.

The underlying CIM technology used in the simulator is based on IBM’s PCM, with configuration details listed in Table I. However, our proposed mapping and scheduling strategies are CIM technology-agnostic and will deliver similar benefits on other technologies as well. We also compare our results to the NVIDIA RTX 3090 Ti GPU.

TABLE I: Baseline CIM parameters for  $d_{model}=1024$

Specification	Latency (ns)	Energy (nJ)
MVM ( $256 \times 256$ PCM)	100	10
ADCs SAR (8b) [23]	0.833	$13.33e^{-3}$
Communication	48	51.7
LayerNorm	100	42
ReLU / GeLU / Add	1 / 70 / 36	0.06 / 38.5 / 37.7

**Mapping & scheduling strategies:** We evaluate three setups:

- *Linear*: Maps the linear layers of the dense pre-trained model onto the CIM accelerator and serves as the baseline for the proposed mapping strategies.
- *SparseMap*: A latency-optimized sparse mapping, as described in Sec. III-B1.
- *DenseMap*: A capacity-optimized mapping strategy, discussed in Sec. III-B2.

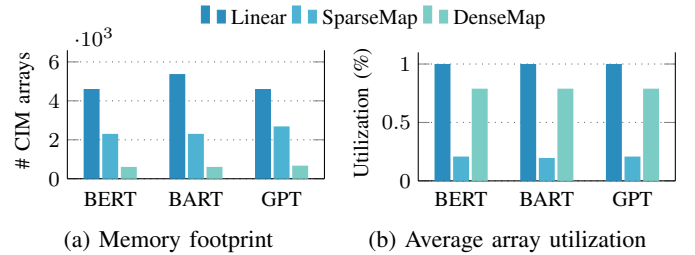


Fig. 6: Comparison of memory requirement and resource utilization across different CIM mapping strategies.

### A. Memory Footprint and CIM Arrays Utilization

In terms of CIM array requirements, Fig. 6a shows that the latency-optimized sparse mapping (*SparseMap*) reduces the number of required arrays by approximately 50% on average across the three models, compared to the *Linear* baseline. In contrast, the capacity-optimized mapping (*DenseMap*) achieves a significantly higher reduction, requiring 87% fewer arrays compared to *Linear*, and over 73% fewer arrays relative to the *SparseMap* configuration.

Fig. 6b compares the array-wise utilization across different configurations. The *Linear* configuration fully utilizes the array capacity. In contrast, the latency-optimized *SparseMap* configuration achieves an average utilization of only 20.4% relative to *Linear*, with 79.6% of the array occupied by padded zeros. The capacity-optimized *DenseMap*, on the other hand, significantly improves utilization, reaching an average of

78.8% of the array capacity. This represents an approximately  $3\times$  improvement over *SparseMap*. While *DenseMap* substantially improves array utilization, it does not achieve full (i.e., 100%) utilization due to a mismatch between the Monarch block and CIM array dimensions, i.e., the block size is not an exact multiple of the array dimension. This residual under-utilization can potentially be further reduced by using smaller block sizes to better align with the CIM arrays' dimensions.

### B. Performance and Energy Comparison

Fig. 7 compares the latency and energy consumption across all configurations. For the BERT model, the *Linear* configuration achieves a speedup of  $16.2\times$  over the GPU and serves as baseline for the other CIM configurations. Averaged (geomean) across all models, the *SparseMap* configuration improves latency by  $1.59\times$  over *Linear* (see Fig. 7a). This improvement stems from three key factors: (i) a reduced number of arithmetic operations due to block-sparse representation, (ii) lower ADC resolution requirements, as only a subset of rows in each array contain non-zero values, and (iii) full parallelism across all CIM arrays, with each block within CIM arrays being processed concurrently (see Sec. III-B1).

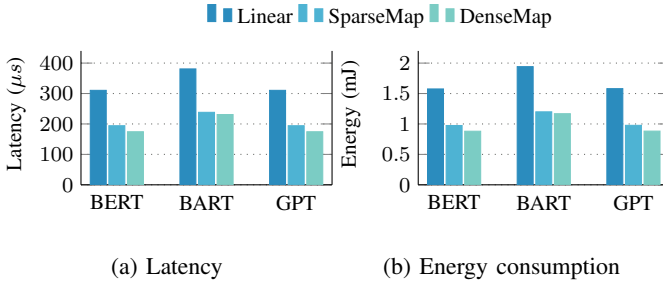


Fig. 7: Latency and energy comparison across configurations.

Although *DenseMap* introduces intra-array sequentiality, its latency is unaffected because (i) all configurations face sequentiality from ADC sharing (assuming one ADC per array; see Fig. 8) and (ii) it uses lower-resolution ADCs (3b vs. 5b in *SparseMap* and 8b in *Linear*). As a result, latency is reduced by  $1.08\times$  over *SparseMap* and  $1.73\times$  over *Linear*.

The energy consumption results (Fig. 7b) show a similar trend. The baseline *Linear* configuration reduces the energy consumption by three orders of magnitude, compared to the GPU. *SparseMap*, on average, reduces energy consumption by  $1.61\times$  compared to *Linear*, while *DenseMap* achieves a reduction of  $1.74\times$ . These gains are primarily attributed to the low-precision ADCs in both configurations.

### C. Design Space Exploration

We previously used a fixed system configuration (Table I). Here, we analyze the sensitivity of latency and energy consumption to the number and precision of ADCs/DACs, which account for 60-80% of the area and energy consumption of MVM in CIM hardware [14]. For the experiments in this section, we use the Accelergy-ADC-plugin [24] to extract the new parameter values and use them with the CIM simulator.

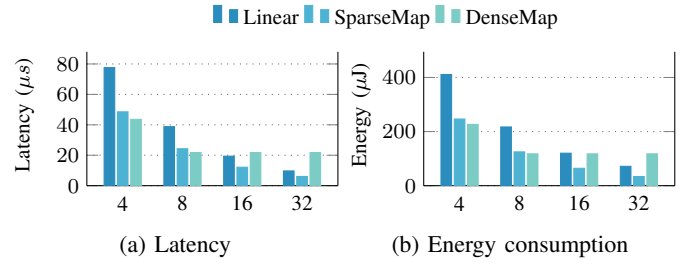


Fig. 8: Latency and energy comparison across varying ADCs per CIM array (X-axis) (BERT model).

Fig. 8 compares latency and energy results for the BERT model under different degrees of ADC sharing, varying number of ADCs per array from 4 (one ADC per 64 column) to 32 (one ADC per 8 columns). Our results indicate that sparse configurations, particularly the capacity/density optimized *DenseMap*, exhibit substantial performance improvements over both the *Linear* and *SparseMap* configurations as the ADC sharing degree increases, i.e., fewer ADCs per array (Fig. 8a). This is due to the inherent sequentiality of this configuration, which better aligns with the sequentiality imposed by ADC sharing. Concretely, *DenseMap* achieves a speedup of  $1.6\times$  and  $1.1\times$  over *Linear* and *SparseMap*, respectively, for the 4 ADCs per array system setup. However, as we increase the number of ADCs per array, *Linear* and *SparseMap* record significant gains while the inherent sequentiality of *DenseMap* does not let the latency improve beyond 8 ADCs/array. For instance, for 32 ADCs per array system setup, it performs worse compared to the other configurations. Our latency optimized *SparseMap* achieves the best result, outperforming *DenseMap* by  $3.57\times$  and *Linear* by  $1.6\times$ .

Energy consumption exhibits a similar trend (Fig. 8b). As the number of ADCs is reduced, the energy savings in *DenseMap* become increasingly significant, reinforcing the benefits of our densely packed configuration that exploits temporal sequencing within arrays and minimizes ADC utilization. **ADC/DAC Resolution:** Lowering converter resolution reduces both latency and energy. Reducing the ADC resolution from 8 bits (required by *Linear* for the given array dimensions) to 3 bits (for *DenseMap*) cuts latency and energy by about  $2.67\times$ .

## V. RELATED WORK

**Structured matrices in digital accelerators:** [25] proposes a HW-friendly model leveraging butterfly sparsity in attention and FFN layers, along with an adaptable, unified butterfly engine. However, their design targets only digital ASICs and is not compatible with analog CIM.

**Sparse attention accelerators in CIM:** ASADI [26] proposes a diagonal compression format and a sparse attention accelerator within the CIM paradigm. TranCIM [27] also targets block-sparse attention in CIM and proposes a scheduler that reduces attention's complexity. While ASADI and TranCIM support block-diagonal sparsity, they overlook the permutations and access patterns from densely packing such structures onto CIM arrays. Moreover, they target only sparse attention with

dynamic data sparsity and no stationary weights, which makes their proposals orthogonal to ours.. In contrast, we are the first to address the mapping and scheduling of parameterized block-diagonal sparsity in both attention and FFN layers for CIM.

## VI. CONCLUSIONS

We present an automated framework for efficiently mapping structured sparse transformer models on analog CIM accelerators. Starting from a dense model, we apply dense-to-sparse transformations to produce Monarch matrices, which are then mapped to CIM crossbars using latency- and capacity-optimized strategies. A mapping-aware scheduler orchestrates execution by generating addresses and issuing CIM commands. Our evaluation of three transformer models shows up to  $1.74\times$  speedup and energy reduction over their baseline dense models. While our simulation framework does not directly support area modeling, the observed reduction in the number of CIM arrays (Fig. 6a) and ADCs (Fig. 8) serve as a reliable proxy, suggesting over  $4\times$  area savings.

## ACKNOWLEDGMENT

The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany and by Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus in the programme Center of Excellence for AI-research „Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig“, project identification number: ScaDS.AI, and funded by the German Research Council (DFG) through the HetCIM project (502388442), and the CRC/TRR 404-Active 3D (528378584).

## REFERENCES

- [1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.
- [3] W. Tan, N. Roberts, T.-H. Huang, J. Zhao, J. Cooper, S. Guo, C. Duan, and F. Sala, “More fine-tuning with 10x fewer parameters,” *arXiv preprint arXiv:2408.17383*, 2024.
- [4] T. Dao, B. Chen, N. S. Sohoni, A. Desai, M. Poli, J. Grogan, A. Liu, A. Rao, A. Rudra, and C. Ré, “Monarch: Expressive structured matrices for efficient and accurate training,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 4690–4721.
- [5] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, “Google workloads for consumer devices: Mitigating data movement bottlenecks,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 316–331. [Online]. Available: <https://doi.org/10.1145/3173162.3173177>
- [6] H. Ahmed, P. C. Santos, J. P. Lima, R. F. Moura, M. A. Alves, A. C. Beck, and L. Carro, “A compiler for automatic selection of suitable processing-in-memory instructions,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 564–569.
- [7] J. P. C. de Lima and L. Carro, “Quantization-aware in-situ training for reliable and accurate edge ai,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1497–1502.
- [8] A. A. Khan, J. P. C. De Lima, H. Farzaneh, and J. Castrillon, “The landscape of compute-near-memory and compute-in-memory: A research and commercial overview,” *arXiv preprint arXiv:2401.14428*, 2024.
- [9] X. S. Hu, M.-Y. Lee, M. Li, J. P. C. De Lima, L. Liu, Z. Zhu, J. Castrillon, M. Niemier, and Y. Wang, “Cross-layer design and design automation for in-memory computing based on non-volatile memory technologies,” *IEEE Design & Test*, 2025.
- [10] T. Dao, B. Chen, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Re, “Pixelated butterfly: Simple and efficient sparse training for neural network models,” *arXiv preprint arXiv:2112.00029*, 2021.
- [11] Q. Dong, M. E. Sinangil, B. Erbagci, D. Sun, W.-S. Khwa, H.-J. Liao, Y. Wang, and J. Chang, “15.3 a 351tops/w and 372.4 gops compute-in-memory sram macro in 7nm finfet cmos for machine-learning applications,” in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020, pp. 242–244.
- [12] J. Lee, H. Valavi, Y. Tang, and N. Verma, “Fully row/column-parallel in-memory computing sram macro employing capacitor-based mixed-signal computation with 5-b inputs,” in *2021 Symposium on VLSI Circuits*. IEEE, 2021, pp. 1–2.
- [13] P.-C. Wu, J.-W. Su, Y.-L. Chung, L.-Y. Hong, J.-S. Ren, F.-C. Chang, Y. Wu, H.-Y. Chen, C.-H. Lin, H.-M. Hsiao et al., “A 28nm 1mb time-domain computing-in-memory 6t-sram macro with a 6.6 ns latency, 1241gops and 37.01 tops/w for 8b-mac operations for edge-ai devices,” in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 1–3.
- [14] S. Negi, I. Chakraborty, A. Ankit, and K. Roy, “Nax: neural architecture and memristive xbar based accelerator co-design,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 451–456.
- [15] A. Nag, R. Balasubramanian, V. Srikumar, R. Walker, A. Shafiee, J. P. Strachan, and N. Muralimanohar, “Newton: Gravitating towards the physical limits of crossbar acceleration,” *IEEE Micro*, vol. 38, no. 5, pp. 41–49, 2018.
- [16] H. Jiang, W. Li, S. Huang, S. Cosemans, F. Catthoor, and S. Yu, “Analog-to-digital converter design exploration for compute-in-memory accelerators,” *IEEE Design & Test*,

vol. 39, no. 2, pp. 48–55, 2022.

- [17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [18] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré, “Learning fast algorithms for linear transforms using butterfly factorizations,” in *International conference on machine learning*. PMLR, 2019, pp. 1517–1527.
- [19] Y. Li, H. Yang, E. R. Martin, K. L. Ho, and L. Ying, “Butterfly factorization,” *Multiscale Modeling & Simulation*, vol. 13, no. 2, pp. 714–732, 2015.
- [20] D. Fu, S. Arora, J. Grogan, I. Johnson, E. S. Eyuboglu, A. Thomas, B. Spector, M. Poli, A. Rudra, and C. Ré, “Monarch mixer: A simple sub-quadratic gemm-based architecture,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 77 546–77 603, 2023.
- [21] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” *arXiv preprint arXiv:1910.13461*, 2019.
- [22] J. Büchel, A. Vasilopoulos, W. A. Simon, I. Boybat, H. Tsai, G. W. Burr, H. Castro, B. Filipiak, M. Le Gallo, A. Rahimi *et al.*, “Efficient scaling of large language models with mixture of experts and 3d analog in-memory computing,” *Nature Computational Science*, pp. 1–14, 2025.
- [23] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.
- [24] “Accelergy ADC Plug-In,” <https://github.com/Accelergy-Project/accelergy-adc-plug-in>, 2025, [Online; accessed 15-June-2025].
- [25] H. Fan, T. Chau, S. I. Venieris, R. Lee, A. Kouris, W. Luk, N. D. Lane, and M. S. Abdelfattah, “Adaptable butterfly accelerator for attention-based nns via hardware and algorithm co-design,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 599–615.
- [26] H. Li, Z. Li, Z. Bai, and T. Mitra, “Asadi: Accelerating sparse attention using diagonal-based in-situ computing,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 774–787.
- [27] F. Tu, Z. Wu, Y. Wang, L. Liang, L. Liu, Y. Ding, L. Liu, S. Wei, Y. Xie, and S. Yin, “Trancim: Full-digital bitline-transpose cim-based sparse transformer accelerator with pipeline/parallel reconfigurable modes,” *IEEE Journal of Solid-State Circuits*, vol. 58, no. 6, pp. 1798–1809, 2022.