Hardware and Software Co-design for Optimized Decoding Schemes and Application Mapping in NVM Compute-in-Memory Architectures

Shanmukha Mangadahalli Siddaramu, Ali Nezhadi, Mahta Mayahinia, Seyedehmaryam Ghasemi, and Mehdi B. Tahoori
Department of Computer Science, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
E-mail: {shanmukha.siddaramu, ali.nezhadi, mahta.mayahinia, seyedehmaryam.ghasemi, mehdi.tahoori}@kit.edu

***Abstract*—The Computation-in Non-volatile Memory (NVM-CiM) approach addresses the growing computational demands and the memory-wall problem faced by traditional processor-centric architectures. CiM capitalizes on the parallel nature of memory arrays enabling effective computation through multi-row memristor reading and sensing. In this context, the conventional design of memory decoders needs to be accordingly modified for efficient multi-row activation and parallel data processing.**

This paper presents the design and optimization of address decoders for NVM-CiM system architectures, employing a cross-layer co-optimization approach that integrates circuit and architecture design with application requirements. Our methodology starts at the circuit level, examining various decoder designs, including cascaded, hierarchical, latched, and hybrid models. An in-depth application-level characterization follows, utilizing an extended NVM-CiM-capable gem5 simulator to assess the impact of these decoders on the mapping of CiM-friendly applications and the resulting system performance, particularly in facilitating rapid and efficient activation of multi-row memory configurations. This holistic analysis allows us to identify the bottlenecks and requirements from the application side and adjust the design of the decoder accordingly.

Our analysis reveals that Hybrid Decoders significantly decrease latency and power consumption compared to other decoder designs within NVM-CiM systems. This highlights the crucial role of the decoder's row selection flexibility, reducing additional system-level data movement even at the expense of its performance, can substantially improve the overall efficiency of NVM-CiM systems.

***Index Terms*—CiM, Decoder, Latch, gem5, Binary Tree Data Structure**

I. Introduction

In today's data-driven computational landscape, the memory-centric computation paradigm is emerging as a promising solution to tackle the memory-wall problem posed by traditional von Neumann computing architectures. Conventionally, computation has always been processor-centric: data is retrieved from the memory, computed by the processor, and then results are stored back into the memory. However, this conventional concept of computation has become less efficient with the rising demands of modern-day computing, particularly Artificial Intelligence (AI) applications [1].

The concept of memory-centric computation takes shape in various forms, considering different methodologies for integrating computation with memory. A distinct variation is Process-in-Memory (PiM) as illustrated by UPMEM, a realistic fabricated architecture that embeds lightweight processors directly within dynamic random access memory (DRAM) chips to facilitate computation near memory [2]. On the other hand, in the Computation-in-Memory (CiM) approach, computation is performed within the memory cores, using the parallel capabilities of memory arrays through multi-row memory activation and sensing techniques, hence, further reducing unnecessary data movement [3]. CiM can utilize a range of memory technologies including charge-based static and dynamic RAM (SRAM and DRAM) [4], [5]. However, the use of memristive technologies as memory devices, with their non-volatile resistive switching properties makes it feasible to perform analog computation and eliminates static power required for data retention, reducing energy consumption [6] [7].

Computation-in Non-volatile Memory (NVM-CiM) paradigm not only promises to increase processing efficiency but also demands a re-evaluation of existing memory peripheral component designs like decoder to support multi-row activation and parallel data handling. Despite the critical role of decoders in the CiM circuitry, architecture, and system, there remains a significant gap in the literature concerning their optimized design and the comprehensive evaluation of their impact on system performance.

The main focus of this paper is the development of address decoder design within the NVM-CiM framework, facilitated by a comprehensive hardware-software co-design methodology. We investigate various decoder designs for multi-row access, including widely reported Cascaded and Latched Decoders, and introduce new hierarchical group activation decoders assessing their efficiency and impact on system performance. A significant contribution of our work is the introduction of a latch-based Hybrid Decoder, designed to provide flexible support for both pre-defined pattern multi-row access and the adaptability afforded by latched outputs, as needed in many applications.

To effectively incorporate these decoders into NVM-CiM systems, we have developed a cross-layer circuit-to-application NVM-CiM framework on the gem5 simulator [8]–[10]. This framework enables a comprehensive evaluation of the circuit design to application performance. It is within this framework that we analyze the efficiency of various decoding schemes

and their impact on overall system performance by considering the implication on CiM application mapping, pinpointing bottlenecks, and application-specific requirements. We simulate six real-world workloads within this NVM-CiM framework, a crucial step for understanding the real-world demands placed on decoders and facilitating the integration of application-driven insights into the circuit design process.

The insights from this analysis are crucial in fine-tuning the design of CiM decoders, especially Hybrid Decoder, to ensure optimal system performance. Highlighting the benefits of our approach, the proposed Hybrid Decoder performs better than widely reported Cascaded Decoders, reducing runtime by over 35%. Through meticulous co-optimization, our goal is to ensure that decoder functionality is perfectly aligned with the operational needs and efficiency requirements of the intended applications, showcasing the potential for substantial improvements in NVM-CiM system performance.

II. Background & Related Work

As mentioned in the introduction, the CiM architecture effectively integrates processing capabilities within the memory units [11]. This section explores the foundational technologies and techniques and an overview of CiM architecture that supports this progress.

A. NVM technologies

The core of CiM combines a densely packed non-volatile memory (NVM) crossbar array with CMOS peripheral circuitry. An example of the core CiM system is shown in Fig. 1(a). There are various NVM technologies, and each is governed by distinct physical phenomena [7] [12]. The promising NVM technologies are spin transfer torque magnetic RAM (STT-MRAM), redox-based RAM (ReRAM), and phase change memory (PCM). These NVM technologies primarily differ in their writing or switching mechanisms. However, from a reading perspective, all NVMs share a common feature: they can exhibit at least two distinct resistive states—low and high resistance states (LRS and HRS), respectively. Therefore, resistance sensing is the reading process in these devices. Fig. 1(b) presents the typical current-voltage (IV) characteristics of ReRAM, highlighting the HRS and LRS states. Moreover, due to non-volatility and analog CiM utilization, NVM-CiM has received a significant amount of interest from the frontier industries in recent years [13]–[15] and show their superiority against charge-based counterparts [4], [5].

B. NVM-CiM techniques

In the NVM-CiM concept, operations can be distinguished by whether they employ stateful or non-stateful logic. Stateful logic, like the Memristor-Aided Logic (MAGIC) method, executes operations within the memory array using the non-volatile resistive states and changing the resistive state of the device to store computational results. Further, non-stateful logic, utilizes resistive devices mainly as input operands, with the peripheral devices performing computations. A promising non-stateful example is the Scouting logic [16]. This method involves concurrently applying an input voltage across multiple rows of resistive devices, corresponding to the number of operands in a Boolean operation. By comparing the aggregate current output from all activated devices against a reference current, one can determine the outcome of the logic function. Notably, Scouting logic relies purely on sense operations, preserving the input data. Fig. 1(c) illustrates the principle for realizing various bitwise logic operations (OR, AND, and XOR) by Scouting logic. Fig. 1(a) shows the typical example of a Scouting-based 2-bit Boolean operation with a modified Pre-charge Sense Amplifier (SA) as comparator [17]. Here, input voltages are applied to the specific number of rows corresponding to the number of operands in a Boolean operation. To determine the result of this Boolean operation, the combined resistance from the activated rows is compared with a reference resistance using a comparator as shown in Fig. 1(d).

In the pre-charge phase, the output nodes (OUT and OUTBAR) are charged to VDD, and in the evaluation phase, both the output nodes are discharged with different rates (RC time constant) which are determined based on the resistive values of the data and reference sides.

In addition, more common arithmetic tasks used in NVM-based hardware acceleration such as vector-matrix multiplication are also non-stateful [3]. Hence, by employing non-stateful logic, NVM-CiM architectures can perform a wide range of computational functions without compromising the endurance of memory devices, positioning it as a versatile solution for advanced computing applications.

C. CiM Architecture

Numerous research papers have been published in the fields of CiM, examining various technologies such as SRAM, DRAM, and NVM as basic infrastructure [18]–[21]. Each of them examines existing issues and offers appropriate solutions. In our case, a narrower focus on NVM-based CiM architectures [22]–[25] reveals a strong emphasis on certain applications, such as neural network acceleration tasks or solutions to enable simple Boolean operations. However, this specialization limits our ability to implement a diverse range of applications. In this situation, even the decoder design can be customized for that particular application, eliminating the need to consider general use-case decoders. The second problem with these architectures was the necessity for an architecture that could be correctly modeled at the circuit level, allowing us to provide precise parameters (such as power, latency, etc.) for our high-level simulations. As a result, we developed a straightforward architecture to address these problems and model the impacts of the decoder with respect to applications.

D. Related Work on Multi-Row Selection Decoder

The design of the decoder plays a crucial role in leveraging the full potential of CiM. It is worth noting that the type of Boolean operations or number of operands can change in runtime, i.e., based on the application's needs and the particular executed NVM-CiM operation. So, the decoders are required

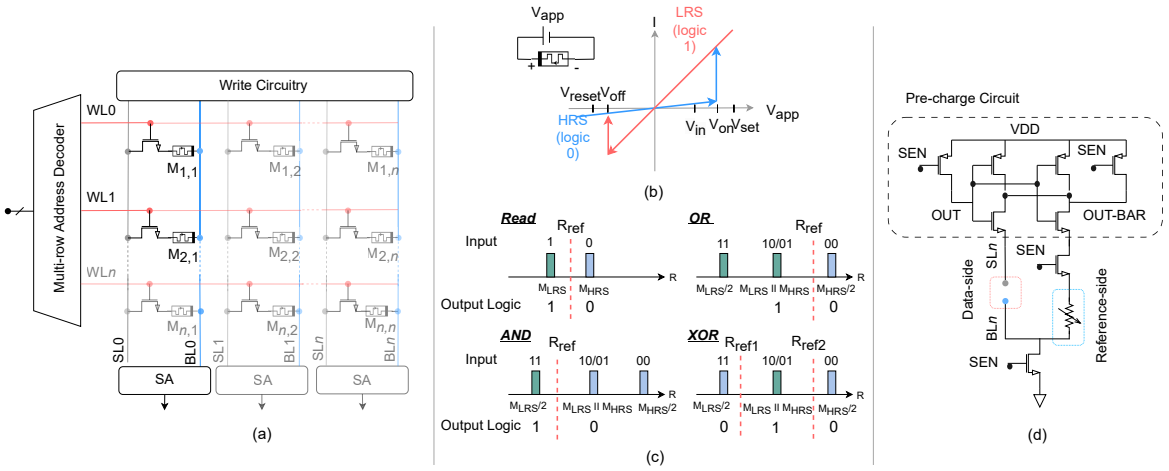

Fig. 1. (a) CiM core architecture with multi-row access decoder. The figure also highlights the scouting-logic operation by activating multiple rows, here $WL0$ and $WL1$, allowing to compute the parallel resistance of $M_{1,1}$ and $M_{2,1}$ for the execution of 2-bit Boolean operations like OR, AND, or XOR directly at the SA. This illustrates how the decoder facilitates memory operations by activating multiple rows for parallel processing. (b) Typical current-voltage (I-V) characteristics of the ReRAM, showing LRS and HRS. (c) Scouting Logic Principle: Reference resistance selection for basic 2-bit logic operations. (d) Pre-charge Sense Amplifier (SA) circuit with reference resistance.

to have the flexibility to select a different combination of rows. After the row selection procedure, the subsequent data must be sensed and interpreted correctly based on the type of computation. Despite its fundamental role, there is limited detailed research specifically on designing the decoders to adapt to the NVM-CiM framework. A frequent trend, from the literature [11], [26] and [27], revolves around parallel cascading of multiple decoders, typically augmented with OR gates. The design presented in [26] employs dual decoders to facilitate the concurrent selection of two rows. This is achieved by linking corresponding output lines to the input of an OR gate, enabling the simultaneous reading of two memory rows. While this design satisfies the functional requirements of 2-bit Boolean computation, it comes at significant costs: a two-fold increase in input lines, area, and power consumption. Additionally, it can select only two rows simultaneously, and it lacks the flexibility to adapt to more number of operands, as required in many CiM-friendly applications.

Another popular multi-row selection decoder design style is Latched Decoders like Texas Instruments' SN74HCS259-Q1, which serves as an 8-bit addressable latch designed for a variety of storage applications in digital systems [28]. This design integrates a latch at every output line of the traditional decoder. The latch enable pin is connected to the corresponding output line of the decoder while all the data pins of the latches are interconnected, enabling simultaneous activation or deactivation of all output lines. To activate three output lines, the corresponding three input combinations must be transmitted sequentially over three successive clock cycles. Hence it requires κ cycles to select κ output lines—selecting each output line in one cycle. Despite the decoder's unparalleled flexibility in the selection of the rows in the crossbar, it has drawbacks in performance metrics, especially latency as a standalone decoder component.

III. Cross-layer Circuit-to-Application CiM Analysis Framework

The foundational step in evaluating the effect of different decoder designs involves developing a robust NVM-CiM architecture framework capable of supporting various applications and accurately modeling the circuit-level backend, as mentioned in the previous section II-C. For this purpose, we utilize gem5, a well-established event-driven architecture simulator, enabling us to conduct our full-system simulations. Additionally, we employ other tools such as SPICE, NVSim [29], and Synopsis to extract circuit-level parameters (such as power, area, and latency) for our gem5 simulations. In the subsequent paragraphs, we elaborate on our implementation of the CiM architecture within gem5.

Fig. 2(a) illustrates a conventional main memory interface, consisting of a memory controller and memory storage. Given our focus on NVM technologies rather than conventional DRAM technologies, we adopt the naming conventions utilized in NVSim for memory storage organization. The memory storage is partitioned into multiple banks, each featuring several MATs (Memory Array Tile) connected to the internal data bus to facilitate data exchange with the memory controller. Additionally, each MAT is composed of a row decoder, column multiplexer, and multiple side-by-side sub-arrays. A sub-array includes a crossbar of NVM cells, a local SA for analog-to-digital conversion, and write circuitry. We assume 64 columns for the sub-array crossbar, aligning with the width of the internal data bus.

To enable CiM, we introduced modifications to the conventional main memory interface, as depicted in Fig. 2(b).

- CiM Controller: A dedicated CiM controller has been added to oversee CiM operations and manage signal assignments effectively.
- Additional logic has been integrated into the memory controller. This logic is responsible for filtering CiM commands and routing them appropriately to the CiM

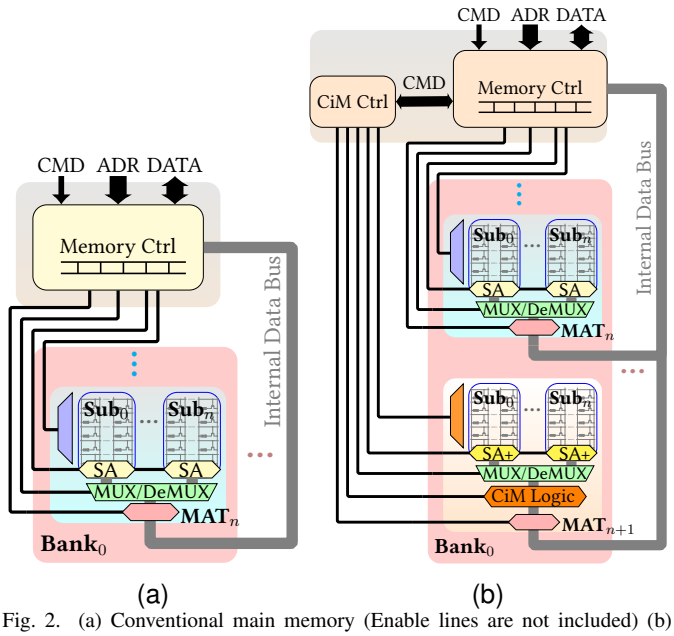

Fig. 2. (a) Conventional main memory (Enable lines are not included) (b) Enabling Compute-in-Memory capability in main memory by adding CiM-capable MATs

controller for processing, which will be discussed more in section III-B.

- Modification of MAT: The last MAT in each bank has been specifically modified to execute CiM operations. These CiM-capable MATs feature a multi-row activation decoder, a buffer to store intermediate results at the output of SA, and a CiM logic, incorporating a 64-bit register with left/right rotate operation, bitwise NOT, and bytewise comparison by 0x00 capabilities.
- By implementing a single CiM logic at the output of each CiM-capable MAT, we can achieve bank-level parallelism. Alternatively, copying the CiM logic into the SA circuitry allows for sub-array-level (or word-level) parallelism.

A. CiM Operations

After analyzing various workloads, we determined that data-intensive tasks, which can be parallelized through simple Boolean-based single instruction multiple data (SIMD) operations such as AND, OR, and XOR, represent the most suitable applications for harnessing CiM capabilities. However, relying solely on Boolean operations is insufficient for effectively leveraging CiM. The continuous offloading of data to CiM units and retrieving results back to the central processing unit (CPU) prove to be more time- and energy-consuming than conducting these operations internally within the CPU. Nevertheless, certain workloads, some falling under the category of Embarrassingly Parallel [30], allow for the majority of their execution to be offloaded to CiM units. Subsequently, the CPU can utilize the prepared results to carry out the final post-processing tasks.

To execute the primary portion of these workloads within the CiM, in addition to basic Boolean operation support, a COPY operation with the capability of data rotation and bytewise comparison by 0x00 is required to implement basic

Listing 1. An example of bitwise NOR operation on two vectors in our CiM API

```
cimModule.OR({ 1, 3});
cimModule.NOT_COND(2, true, false); // bitwiseNOT
```

if statements. Bytewise comparison by 0x00 involves performing an OR operation on all 8 bits in a byte. Based on the result, the corresponding output byte will be either 0xff or 0x00. Bytewise comparison with 0x00 is almost equivalent to the **PCMPEQ** command in the MMX extension for Intel architectures [31].

B. Interaction with CiM from Application-layer

While the CiM controller is inactive, the memory controller can execute regular read/write operations in the CiM region, similar to other memory regions. However, when the CiM controller is active, the memory controller must wait for the CiM controller to complete its tasks before accessing the corresponding regions. To facilitate communication from the application layer to the CiM controller, we have considered a fixed physical shadow address, i.e., a virtual address without a specific physical location. Commands issued from the application layer are written to this address, and upon detection by the memory controller, the content is directly forwarded to the CiM controller.

It is essential to note that we considered this memory mapping as non-cacheable and non-shareable. Non-cacheability is necessary as we directly write data into the main memory. Non-sharability is crucial to prevent a process from acquiring this hardware resource while another process has not released it, ensuring data consistency. To enforce these requirements, an operating system driver is provided. Additionally, an application programming interface (API) allows programmers to interact with this hardware resource. Listing 1 presents a simple code snippet of this API for bitwise NOR operation on two vectors, while Fig. 3 illustrates the actual operations within the CiM units.

In Listing 1, it is assumed that certain content is already stored in row #1 and row #3 within the CiM units (shown as idle state in Fig. 3(0)), the OR operation is executed simultaneously inside the sub-arrays, by activating the two lines (row #1 and #3 in Fig. 3(1)). Subsequently, the NOT operation is performed on the intermediate results using the CiM logic (Fig. 3(2) and Fig. 3(3)), and finally, the outcomes are written back to row #2 (Fig. 3(4)). Since bitwise NOT and bytewise comparison are combined in a single operation (**NOT_COND** in Listing 1), two argument flags are utilized to distinguish between them: one for bitwise NOT and another for bytewise comparison with 0x00. Moreover, **NOT_COND** can accept an optional *source row* argument to specify the row to operate on, instead of operating on the output of buffers in CiM units (see lines 15 and 17 in Listing 3).

To demonstrate the implementation of an **if** statement in the CiM API, we can consider the ternary operator in C++ (since it closely resembles what executed inside the CiM units.), as illustrated in Listing 2. The difference between sequential code for the CPU and code for CiM is that, in the case of the CPU, when a conditional statement is not satisfied during an iteration, the **if** statement is simply skipped.

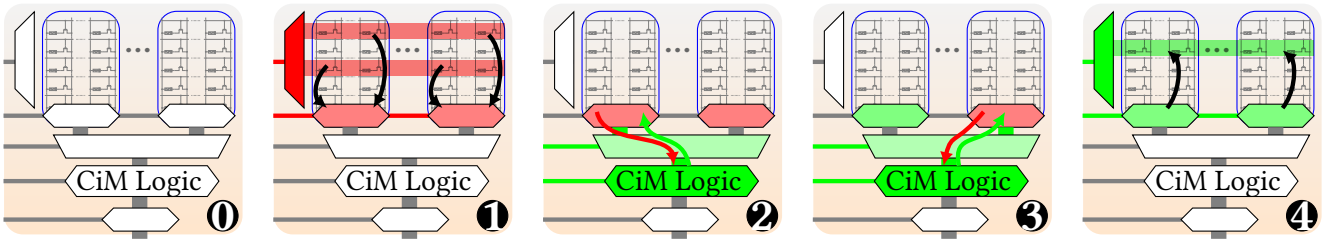

Fig. 3. Operation execution example in CiM units

Listing 2. An example of a ternary operator in C++ language

```
for(size_t i=0; i< MAX_ROW_SIZE; i++)
    a[i] = (b[i]==0x12) ? c[i] : d[i];
```

However, in the case of CiM, there is no mechanism to skip commands; every command is executed on all elements. To avoid changing the content of some elements when a condition is not met, it is necessary to copy their content back in some way. This is achieved using condition masks generated by **NOT_COND** in combination with other CiM commands to ensure correct functionality. To translate this sequential C++ code into CiM commands, we assume that each vector has a maximum size equal to the CiM row size. In our simulations, we consider 16 banks, each equipped with one CiM-capable MAT. Each CiM-capable MAT includes 64 sub-arrays with a column size of 8 bytes. Therefore, in this example, the **MAX_ROW_SIZE** would be $16 \times 64 \times 8 = 8\,\text{KiB}$. Of course, larger vectors can be divided into aligned sizes.

Listing 3 presents the equivalent code using the CiM API. In this scenario, the programmer copies each vector to a dedicated row. Consequently, operations on each byte are executed in parallel from the programmer's perspective, eliminating the need for the **for** loop. If our application includes other types of loops, they will be unrolled during the CiM code generation, considering the absence of loop control commands in our CiM Controller. Similar to someone using SIMD instructions in assembly language to convert a sequential code into a parallel version, the responsibility for this conversion lies with the programmer in the CiM context as well.

Each CiM application operates with a fixed number of commands and is designed to function independently of CPU interaction during CiM operations. Consequently, issuing CiM commands individually from the application level, loading files into the cache, and subsequently offloading them to the CiM region, is found to be inefficient, proves inefficient, and goes against the CiM concept. Instead, a more efficient approach involves storing all these commands in the main memory, triggering the CiM controller to read and execute these commands, and performing intra-main memory data loading if necessary. To achieve this, we must integrate internal direct memory access (DMA) and share the internal memory bus between the DMA and the memory controller, and the details are outside the scope of this paper.

C. Application Implementation for Different Decoders

In this section, we discuss our approach to implementing the same application across different decoders, detailing the crucial steps and considerations involved.

Listing 3. Converting sequential ternary operators to parallel CiM commands

```
// row0 <- b...
cimModule.copy_to_cim(0, (void *)&b[0]);
// row1 <- 0x12...
cimModule.copy_to_cim(1, (void *)&CONSTx12[0]);
// row2 <- c...
cimModule.copy_to_cim(2, (void *)&c[0]);
// row3 <- d...
cimModule.copy_to_cim(3, (void *)&d[0]);
// row4 <- a...
cimModule.copy_to_cim(4, (void *)&a[0]);

// (b[i]==0x12)
cimModule.XOR({0,1})
// row5 <- 0xff if result is 0, otherwise 0x00
cimModule.NOT_COND(5, false, true); // Condition
// row6 <- ~row5
cimModule.NOT_COND(6, 5, true, false); //Bitwise
// row7 <- row2 & row5
cimModule.AND({2,5})
cimModule.COPY(7)
// row8 <- row3 & row6
cimModule.AND({3,6})
cimModule.COPY(8)
// a <- row7 | row8
cimModule.OR({7,8})
cimModule.COPY(4)
```

Considering the limitations of the decoder, allocating rows for different vectors becomes extremely challenging during application implementation. Implementing an application for a latch-based decoder is relatively straightforward, as the programmer can easily assign a vector to any desired row, allowing all rows to be freely activated in combination with others. However, this simplicity is not reflected in tree-based decoders. For instance, in a tree-based decoder, row #1 can only be activated simultaneously with row #0. To address this challenge, we tried to automate the row allocation process. In other words, the programmer can initially write the application for a latch-based decoder, and later, using our CiM API, it will be converted to the desired decoder.

Register Allocation is a well-known problem in compiler optimization. While high-level language programmers can use numerous variables in their applications, the limited number of physical CPU registers necessitates efficient register allocation by compilers. This problem can be mapped to the graph coloring problem, a known NP-complete problem. In our case, each row can be considered analogous to a physical CPU register. However, the decoder limitations introduce additional constraints to this graph-coloring problem since multiple rows need to be activated together. In other words, each node of the register allocation graph becomes a sub-graph in our case,

leading to an exponential runtime to solve.

To address this challenge, we utilized a heuristic method that scans through the intermediate codes generated by the CiM API for the latch-based implementation of a particular application. It identifies all unique combinations of rows and counts their occurrences in the intermediate code. Subsequently, it maps these rows to CiM rows while considering the decoder limitations. This approach ensures that high-demand rows are efficiently allocated to the CiM region, maximizing performance and minimizing unnecessary data relocation. Hence, we are not optimizing application implementation rather we are optimizing row allocation.

As discussed in Sec. III-B, data-dependent control flow is not present in our CiM applications at the application layer. In other words, there are no 'if/else' statements to selectively execute CiM commands based on individual vector or element values. Instead, CPU-version code must be manually converted to CiM-version code, and mask flags generated by the **`NOT_COND`** command are used to achieve the desired functionality, similar to the **`PCMPEQ`** instruction in Intel MMX. Since all CiM commands are constant expressions, they can be evaluated and optimized at compile time. However, as conventional compilers are not capable of optimizing these type of commands, a heuristic method is employed to optimize row allocation based on the target decoder.

In our heuristic method, the focus is solely on multi-row commands (AND, OR, and XOR), as each decoder can easily activate any single row, with no consideration given to the operation type, as they are uniformly treated by the decoders. Consequently, a unique collection of sets is obtained based on the input row arguments (e.g., $\{1,2\}$, $\{0,5\}$, $\{7,8,9,2\}$). In the context of sets, the order of elements does not matter, and the number of occurrences of each set is considered in an unrolled manner. By 'unrolled manner,' it is meant that all loops in our applications are unrolled by simply executing them. The goal is to map these unique sets as closely as possible to the unique sets of the target decoder, giving priority to the sets that occur most frequently. For example, the set $\{2,7,9,12\}$ might be mapped to the set $\{0,1,2,3\}$. All the row labels 2, 7, 9, and 12 in the application are then renamed to 0, 1, 2, and 3, respectively, as if they were chosen by the programmer in the first place.

IV. CiM Decoder Designs

In this section, we propose three novel decoder designs, each with its distinct architectural or algorithmic underpinning and the core objective of enabling multi-row selection. They are namely, in this paper, κ-Grouped Hierarchical Decoder which is based on the traditional decoder, Tree Decoders which are inspired by binary tree data structure, and Hybrid Decoder. Importantly, Tree and κ-Grouped Decoders form hierarchical group pattern activation decoders as they are engineered to activate $\kappa = 2^i$ output lines simultaneously, where i varies from 0 to $\log_2(WL)$ (WL being the number of output lines of the decoder or number of word-line in the memory crossbar). We then compare these designs against the cascaded and latch-based decoder from the literature. Each design proposed offers unique advantages and when implemented, promises to overcome the noted shortcomings of its predecessors and also adapt to CiM requirements.

TABLE I
TRUTH TABLE FOR $(n+1):2^n$ κ-GROUPED HIERARCHICAL DECODER FOR n = 2

	Input			Output			
	A	B	C	WL3	WL2	WL1	WL0
κ=1	1	0	0	0	0	0	1
	1	0	1	0	0	1	0
	1	1	0	0	1	0	0
	1	1	1	1	0	0	0
κ=2	0	1	0	0	0	1	1
	0	1	1	1	1	0	0
κ=4	0	0	1	1	1	1	1
	0	0	0	0	0	0	0

A. κ-Grouped Hierarchical Decoder

The κ-Grouped Hierarchical Decoder presents a nuanced enhancement to the traditional decoder paradigm. This design integrates an extra input line, a minor change that gives rise to an increase in decoding versatility by twice. Consequently, the architecture of this decoder transitions from the usual $n:2^n$ to $(n+1):2^n$ configuration, where n is the number of input lines to the decoder. To illustrate, consider a $(2+1)$ to 2^2 decoder. The logic behind its operation is detailed in the Table I. From the table,

- to individually activate WLs, set the most significant bit (MSB) of input lines to logic 1. The next 2 input bits then determine which one of the 4 output lines gets activated.
- to activate two consecutive (2-Grouped) WLs, set the top two MSB bits to "01". The remaining last input bit is then used to decode 2 out of 4 consecutive output lines simultaneously.
- to activate the four WLs, set the top three MSB bits to "001".

This decoder design is extended to higher order up to (7+1):2^7 configuration in this work.

B. Tree Decoders

The hierarchical nature of binary tree data structure provides a template that can be flexibly adapted for decoding purposes, making it an ideal choice for multi-row access. The implementation method may vary depending on the specific application requisites. Within the scope of this paper, our emphasis is on the activation of 2^i output lines, proposing two distinct possible algorithmic approaches to achieve this.

1) Tree Decoder-1: In essence, a binary tree comprises nodes where each node, excluding the leaf nodes, bifurcates into two distinct child nodes: the left and the right child. In the context of the Tree Decoder-1 (refer Fig. 4),

- *Node Representation*: Each node represents a specific set of output lines. The root node represents all output lines, whereas the leaf nodes stand for individual output lines.
- *Propagation Logic*: The propagation mechanism in the Tree Decoder-1 takes hints from binary tree traversal.

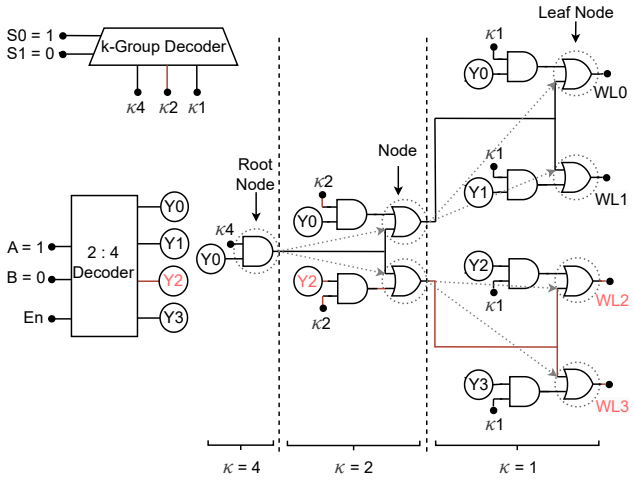

Fig. 4. Binary tree structure of Tree Decoder-1 showing the activation propagation logic for $WL2-3$.

For activating a specific set of outputs, specific nodes are activated. Activation of a parent node will inherently propagate the signal to its child nodes, mirroring the nature of binary tree traversal.

For instance, a 2 to 4 decoder with a 3-layer (or depth) binary tree structure is shown in Fig. 4 and its corresponding truth table is shown in Table II:

- *Full Activation*: Activate the root node ($\kappa 4$=1), by setting the signals $S0$ and $S1$ to logic 0 and 1 respectively. This will inherently activate all underlying nodes. Thus, if the inputs are A =0 and B =0, then the $Y0$ line carries a logic 1, this is inherently propagated through the entire tree, resulting in all output lines (WL) having logic 1.
- *Partial Activation*: For targeted activation, specific branches or depths of the tree must be activated. For example, to activate the last two consecutive output lines ($WL2$ and $WL3$), activate the layer containing two nodes ($\kappa 2$=1) by setting the signals $S0$ and $S1$ to logic 1 and 0 respectively. Now, the logic value of $Y2$ will be propagated to the desired output lines. This mechanism benefits from the inherent hierarchical structure of binary trees, allowing for efficient and selective propagation. This example is clearly illustrated with the red color highlights in Fig. 4.
- *Individual Activation*: For individual WL activation, set both signals $S0$ and $S1$ to logic 0. This action selects the leaf nodes layer (as $\kappa 1$ will be set to logic 1). With these settings, the decoder operates in a traditional 2:4 configuration, activating a single output line based on the values of the two input signals, A and B.

In summary, the depth of the tree offers the granularity of the activation. A deeper tree enables more specific output line targeting, whereas the breadth (κ) at any given depth provides the range of outputs that can be concurrently activated.

2) Tree Decoder-2: In this tree decoder, we have $2n$ input lines where the first set of n bits are referred to as the *Both-branch Selection vector* and the second set of n bits as the *Side-branch Selection vector*. The desired output line (WL) is activated using the Side-branch Selection vector when

TABLE II
TRUTH TABLE FOR $n + \lceil \log_2(n+1) \rceil : 2^n$ TREE DECODER-1 FOR $n = 2$.

	Input				Output			
	S1	S0	A	B	WL3	WL2	WL1	WL0
κ=1	0	0	0	0	0	0	0	1
	0	0	0	1	0	0	1	0
	0	0	1	0	0	1	0	0
	0	0	1	1	1	0	0	0
κ=2	0	1	0	0	0	0	1	1
	0	1	1	0	1	1	0	0
κ=4	1	0	0	0	1	1	1	1
En=0	x	x	x	x	0	0	0	0

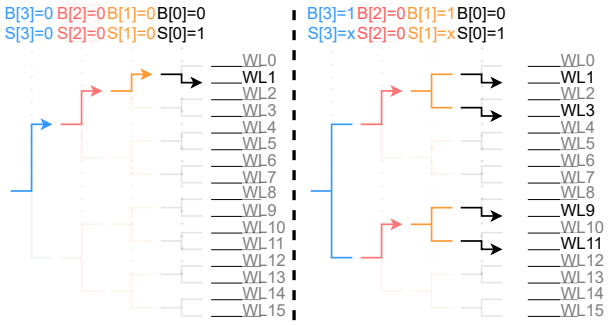

Fig. 5. Two different examples for input configuration in Tree Decoder-2, where B is Both-branch Selection vector and S is Side-branch Selection vector.

the Both-branch Selection vector is zero (functioning as a traditional decoder). Whenever any of the bits in the Both-branch Selection vector are set to logic 1, the corresponding bits (with the same indexes) in the Side-branch Selection vector will be ignored, and both branches of each node in the corresponding layer of the tree hierarchy will be activated. Likewise, whenever any of the bits in the Both-branch Selection vector are set to logic 0, the corresponding bits in the Side-branch Selection vector (based on their values) will activate the corresponding left or right branch of each node.

For a better demonstration, Fig. 5 shows two different examples. For 16 output lines, we have 8 input bits (4 bits for the Both-branch Selection vector and 4 bits for the Side-branch Selection vector). In Fig. 5, on the left, all the bits in the Both-branch Selection vector are set to logic 0, and the value of the Side-branch Selection vector (`0001`) activates output row #1. On the right, rows #1, #3, #9, and #11 are activated by setting the values of the Both-branch Selection vector to `1010` and the Side-branch Selection vector to `x0x1` (`x` indicates don't care).

In general, by having $2n$ input signals, we can activate 3^n different combinations of WLs. Table III demonstrates the truth table for a $2n = 4$ Tree Decoder-2. Furthermore, Fig. 6 comparison study shows the circuit diagram of a Tree Decoder-2 and how it can be combined in a hierarchical fashion to scale. Moreover, multiple instances of Tree Decoder-2 can be combined with a regular decoder to scale.

C. Hybrid Decoder

The κ-Grouped and Tree decoders mentioned above follow a hierarchical group pattern for output line activation. While being innovative, these approaches do not always meet the

TABLE III
TRUTH TABLE FOR $2n : 2^n$ TREE DECODER-2 ($n = 2$), WHERE B IS BOTH-BRANCH SELECTION VECTOR AND S IS SIDE-BRANCH SELECTION VECTOR.

	Input				Output			
	B1	B0	S1	S0	WL3	WL2	WL1	WL0
κ=1	0	0	0	0	0	0	0	1
	0	0	0	1	0	0	1	0
	0	0	1	0	0	1	0	0
	0	0	1	1	1	0	0	0
κ=2	0	1	0	x	0	0	1	1
	0	1	1	x	1	1	0	0
	1	0	x	0	0	1	0	1
	1	0	x	1	1	0	1	0
κ=4	1	1	x	x	1	1	1	1
En=0	x	x	x	x	0	0	0	0

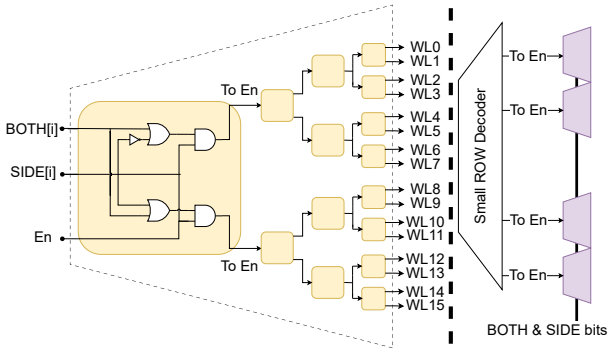

Fig. 6. Circuit diagram of a Tree Decoder-2 with 16 WL (left), and the combination of a regular small decoder with multiple instances of Tree Decoder-2 (right). ($BOTH$=Both-branch Selection vector, $SIDE$=Side-branch Selection vector)

dynamic needs of various applications. It often requires additional COPY operations to activate non-predefined hierarchical output lines at the system level, creating a significant bottleneck in system performance due to the high latency and energy consumption associated with write operations in NVMs. So making the decoder more flexible by eliminating the COPY operation and additional data mapping, even at the expense of its performance, could significantly benefit overall system performance in terms of speed and energy efficiency.

To this end, we introduce a ***latch-based hybrid design*** that maximizes flexibility by allowing the selection of specific combinations of inputs, while still trying to make it efficient. By minimizing the total number of cycles needed for decoding across all cases, merging the flexibility of latch-based designs with the efficiency of combinational designs. Unlike conventional latch-based decoders that require κ cycles to select κ out of WL output lines—selecting each output line in one cycle—the Hybrid Decoder selects κ_1 outputs in the first cycle, κ_2 outputs in the second cycle, and so on, until all κ outputs are activated, where $\sum_{i=1}^{m} \kappa_i = \kappa$. The aim is to minimize m, the total number of cycles necessary to activate all required decoder outputs/rows in the Hybrid Decoder.

This design is achieved by adapting the κ-Grouped decoder to integrate the most common output line combinations as determined by a thorough analysis of application behaviors and by adding latch functionality for individual line selection. Thus Hybrid Decoder provides customized, application-specific activation patterns. Its dual-mode functionality ensures adaptability to adjust the changing computational demands, allowing for any combination of output lines to be activated.

The development of this hybrid design was made possible through our proposed cross-layer CiM framework. This framework is essential for bridging the gap between application-level requirements and circuit-level design constraints.

1) Design and Functionality of the Hybrid Decoder: The design process for the Hybrid Decoder begins with a detailed analysis to identify the most frequently utilized multi-row output line combinations across a variety of applications. This step is crucial for making informed decisions about the specific patterns that need to be readily activatable by the decoder.

For illustrative purposes, consider a 4-output line decoder scenario. Analysis of application requirements might identify the four most popular output line combinations as $\{WL0, WL1\}$, $\{WL2, WL3\}$, $\{WL1, WL2\}$, and $\{WL0, WL2, WL3\}$. Given these combinations, a total of 8 distinct activation scenarios emerge, necessitating the use of 3 input lines for comprehensive coverage. The accompanying Table IV provides a truth table for the Hybrid Decoder, illustrating its ability to activate both single output lines and pre-defined patterns with example configurations. This table serves as a practical reference for understanding the Hybrid Decoder's operation and its application in enabling efficient and flexible CiM computations. So with a single clock cycle, we can activate the desired combination if it's part of a design like activating $\{WL1, WL2\}$ together. In the case of the activation of output combinations like $\{WL3, WL1, WL0\}$, the combination for activation is not readily available. In this case, we can make use of the latched feature in design with sequential input signals $\{A, B, C\}$ = {0, 1, 0} followed by {1, 1, 1} in two clock cycles can trigger the desired output, with the latched state persisting until a reset as shown in Table IV. Hence, for the activation of uncommon combinations, we can also have latch functionality to the decoder, giving complete flexibility in activating any combination. The Hybrid Decoder combines the high-efficiency multi-row activation capabilities of pattern decoders with the flexibility of latch-based decoders.

2) Implementation and Coverage: In this work, we have designed 16 and 32 output line Hybrid Decoder with 4 and 5 input lines respectively to achieve a $(n+1) : 2^n$ configuration, in general. In the 16 output line decoders, the first 8 output combinations are designated to activate a single output line, with the subsequent 8 output combinations designed for popular pattern activation. These selected patterns, determined by application analysis, covers up-to 96% of application-required output line combinations within a single clock cycle. Similarly, the 32-output line decoder achieves up-to 97% coverage in just one cycle, with the flexibility to activate any remaining combinations within an additional 2-3 cycles, thanks to the latch functionality.

D. Design of Sense Amplifier with Decoders

For Boolean-based computations, the reference side of the pre-charge SA needs to be adjusted to account for the number of inputs and the type of the Boolean operation. For this aim, we reuse the *reference trimming* concept [32]. Reference trimming has been originally introduced to mitigate the effect of

TABLE IV
TRUTH TABLE FOR HYBRID DECODER WITH 4 OUTPUT LINES WITH EXAMPLE 4 PRE-DEFINE PATTERN IN ADDITION TO SINGLE OUTPUT LINE ACTIVATION.

	Input				Output			
	Clock	A	B	C	WL3	WL2	WL1	WL0
Single output line activation	↑	1	0	0	0	0	0	1
	↑	1	0	1	0	0	1	0
	↑	1	1	0	0	1	0	0
	↑	1	1	1	1	0	0	0
Pre-defined pattern activation	↑	0	1	0	0	0	1	1
	↑	0	1	1	1	1	0	0
	↑	0	0	1	0	1	1	0
	↑	0	0	0	1	1	0	1
Reset=0	↑	x	x	x	0	0	0	0

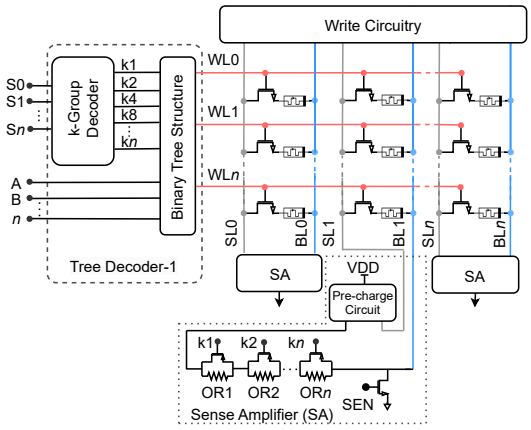

Fig. 7. Schematic representation of the pre-charge SA adjustments concerning the Tree Decoder-1 structure for Scouting logic. This showcases the multiplexed reference resistance modulation based on Boolean OR (NOR) operation and operand count.

the process and thermal variation in the normal resistive NVM read operation [33]. The structure of the trimming circuitry consists of a series connections of multiple resistances that can be individually bypassed. This trimming circuitry can be re-purposed to implement Scouting logic. Hence the periphery circuitry is added with a trim-controlling mechanism to select the resistance to be bypassed. This trim-controlling mechanism requires specific signals and, consequently dedicated circuitry that is co-optimized with the CiM controller to generate them.

As shown in Fig. 7, the structure of Tree Decoder-1 can inherently generate the required signals $(\kappa 1, \kappa 2, \kappa 4, ..\kappa n)$ for controlling the trimming circuitry. Therefore, enabling us to modulate the balancing resistances based on the Boolean operation and the number of input operands. In contrast, for the Hybrid Decoder, Tree Decoder-2, and κ-Grouped Hierarchical Decoder, an additional multi-bit signal circuitry is introduced to manage reference side control. This circuitry, designed with minimal multiplexing overhead, is co-optimized with the CiM controller for precise trim adjustment. This is done by embedding the number of operands directly into the command bits from the application layer, using them as enable inputs for the trim circuitry. Each proposed decoder design introduces distinct challenges to the multiplexing framework for adjusting the sensing reference of the SA. This highlights the necessity for an integrated design approach combining both the decoder and SA.

V. SIMULATION SETUP AND RESULTS

Following the exploration of various decoder design styles, this section outlines the simulation setup and presents the corresponding results. Initially, each decoder is analyzed at the circuit level, followed by a detailed evaluation within an NVM-CiM environment to demonstrate their performance under practical operational conditions. This approach allows us to present a comprehensive analysis that highlights how these decoders interact with and impact the dynamics of an NVM-CiM system.

A. Simulation Setup

The implementation of the decoder designs was carried out using Verilog Hardware Description Language (HDL), and these designs were synthesized using the Synopsys Design Compiler, adhering to the specifications of the Global Foundry 22FDX library. The synthesized results provided the performance metrics for incorporating the decoders into a CiM system, facilitating an in-depth analysis of their impact on the system's overall performance in running the application. To calculate the relative area of the decoders in the CiM system, we developed NVSim++, a modified version of NVSim [29], to approximate the latency, energy, and area of the NVM-CiM functionality. To realize the CiM functionality for NVSim++, in the memory hierarchy of bank → MAT → sub-array, we assumed that only a few sub-arrays are CiM-capable. Therefore, the modifications are at the sub-array level, focusing on the decoder and the sense amplifier. The synthesized results for the CiM-capable decoders, including area, energy, and latency, were incorporated into NVSim. For the sense amplifier which accounts for the number of the activated rows and the intended logical operation, we conducted electrical-level simulations using the SPICE tool to gather the necessary information for updating NVSim. The results of the relative area that each decoder occupies within a sub-array of size 64×64, as obtained from NVSim++, are shown in Table VI.

For this paper, we implement six CiM-friendly applications in our framework. We define CiM-friendly applications as prevalent applications that primarily rely on numerous Boolean operations for computation, and they typically can be parallelized for execution on vector and/or array processors. While utilizing CiM for some of these applications might not seem ideal, the intention was to demonstrate CiM capabilities. These applications are as follows:

1) **BitIndexing [34] (BIT):** Bitmap-formatted database queries are being ORed with each other, and then the results are being ANDed with another query. At the end, the number of activated bits is counted and reported as the final result.
2) **BLASTN algorithm [35] (BLS):** It is a well-known algorithm for searching short DNA sequences inside large DNA protein sequences.
3) **Morphological Image Processing [36] (MIP):** A variety of procedures can be applied to images during image processing. For this study, we implemented Dilation and Erosion algorithms with binary-coded input images with

TABLE V
MAIN ASSUMPTIONS FOR THE SYSTEM SETUP IN GEM5

NVM models	STT-MRAM / ReRAM / PCM
CPU type / ISA	OoO / x86-64
Compiler & Optimization Flag	`g++ -O3`
Clock Frequency	1 GHz
# of Load/Store Queue Entries	32 / 32
# of Reorder Buffer Entries	128
# of Instruction Queue Entries	64
# of Physical Int/FP/Vec Registers	128 / 128 / 128
L1 Inst/Data Cache Size	32 KiB
L1 Tag/Data/Response Latency	2 / 2 / 2 cycles
L2 Cache Size	256 KiB
L2 Tag/Data/Response Latency	20 / 20 / 20 cycles
System Bus Latency	10 ns
Average Memory Controller Latency	15 ns
Read Latency	1 / 1 / 1 cycle
Write Latency	4 / 45 / 40 cycles

various filter sizes 3x3 and 6x6 (MIP-3 and MIP-6 respectively).

4) **Marching Squares [37] (MSQ):** It is an algorithm used for extracting contour lines from 2D images. For this paper, we implement this algorithm for binary-coded input images.
5) **Shifted Hamming Distance [38] (SHD):** It is a famous algorithm used for calculating edit distances in short DNA sequences. It accepts a minimum-distance number and checks two sequences with each other to see if they satisfy this minimum-distance condition or if the number of insertions, deletions, or mutations is larger than the minimum-distance number. Here, we set minimum distance at 3 and 6, giving SHD-3 and SHD-6 respectively.
6) **BitWeaving [39] (BWV):** A technique presented to scan database queries in memory and return the results that satisfy the input conditions (e.g., $a \leq x < b$).

Our comprehensive simulation setup, including key assumptions, is detailed in Table V. To focus on the decoder impacts, we limited the number of CiM rows to 16 and 32, while it was possible to use more rows. Due to this decision, in this work, we present circuit-level decoder results for 32-bit output lines, despite having designed and simulated decoders capable of supporting up to 128-bit output lines. While we investigated different non-volatile memory (NVM) technologies, we only report results for STT-MRAM here. Because other technologies have significant write delay, their results are proportional to STT-MRAM technology, hence we have excluded them from this research. To compute the off-chip power and latency of the memory, we used NVSim, built originally on top of CACTI [40] with NVM extensions. Moreover, to accurately model multirow activation property, we performed circuit-level (SPICE) simulations for NVM components. An NVM crossbar is simulated with pre-charge SA as shown in Fig. 7. These circuit simulations were executed in Cadence Virtuoso with the Global Foundry 22FDX library. All simulations were carried out at a standard VDD of 0.8 V at the temperature of 27°C, ensuring a conducive and stable environment for realistic results.

B. Circuit-level Results

A comprehensive simulation was conducted to evaluate key performance metrics such as area, power, latency, and the power-delay product (PDP) across nine distinctive decoder designs, as detailed in Table VI. Along with decoders mentioned until now, this also included an assessment of a 4-Cascaded Decoder capable of selecting up to four output lines simultaneously and a Serial-input-parallel-output register (SIPO), which operates as a decoder by inputting a serialized binary representation of the desired active output line.

Among the designs, the Hybrid and Sequential decoders excel in flexibility, but they are not the most efficient in terms of area, power, or latency. This inefficiency, for instance, in the Latched Decoder, arises from the additional latching layer added to the traditional decoder circuitry. In the Hybrid Decoder, inefficiency stems from merging complex combinational logic needed for activating predefined output combinations with latching mechanisms for dynamic selection.

Tree Decoder-2 offers the best area efficiency. It achieves this by using more input lines, which simplifies the overall logic required to implement the decoding functionality, hence reducing the logic area. It slightly outperforms the 2-Cascaded Decoder. Yet, it is important to note that, specifically for dual-row active applications, 2-Cascaded Decoder offers a more flexible selection. It also exhibits the lowest circuit-level latency among all tested designs, a result of having fewer stages from input to output ports, closely followed by Tree Decoder-2. However, cascaded decoder's design strategy shows scalability concerns with power-delay product (PDP) rise unfavorably with increasing size. The efficiency of Tree Decoder-2 in varying group selection while maintaining a minimal area footprint renders it an excellent choice when considering both area and latency.

Despite its larger logic area compared to Tree-Decoder-2 and 2-Cascaded Decoder, Tree Decoder-1 and κ-Grouped Decoder show a significant reduction in power consumption with a reduced number of input lines. Unlike other decoder schemes, specifically, the Tree Decoder-1 approach ensures that only the necessary paths are activated to reach the desired outputs, thereby reducing power consumption. However, Tree Decoder-1 and κ-Grouped Decoder are less versatile in output line selection compared to Tree Decoder-2, as observed in Fig. 4 and Fig. 5.

In scenarios where the number of input lines is a concern, the κ-Grouped Hierarchical Decoder emerges as an optimal choice. It requires only one additional input line compared to traditional decoders, yet maintains performance metrics that are substantially lower or on par with other decoders. Consequently, this decoder serves as the basis for the design of the Hybrid Decoder, as detailed in Section IV-C, due to its balanced performance attributes.

C. System-level Results

After establishing our simulation setup and analyzing standalone decoder performance, we next assess these decoders within the NVM-CiM framework using the gem5 simulator. This framework offers a detailed model that evaluates the

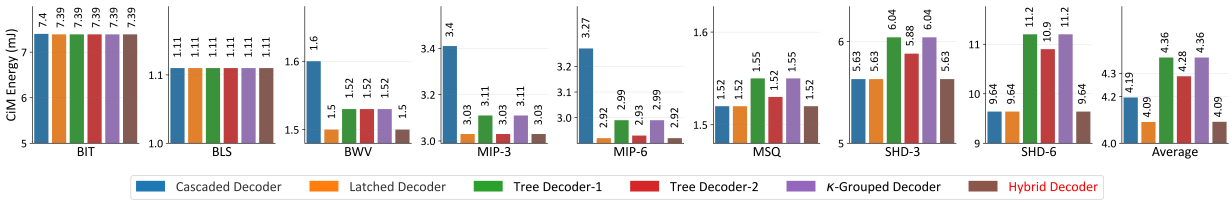

Fig. 8. Energy Consumption of different applications using CiM with the capacity of **32 rows**

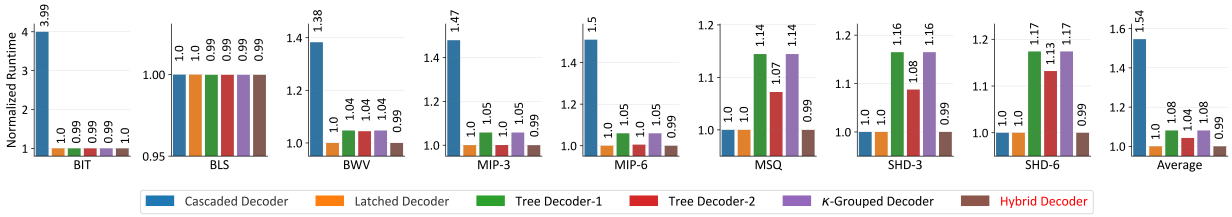

Fig. 9. Normalized Runtime Latency of different applications using CiM with the capacity of **32 rows**

TABLE VI
CIRCUIT-LEVEL COMPARATIVE SIMULATION OUTCOMES FOR 9 DISTINCT DECODER DESIGN STYLES OF SIZE 32 OUTPUT LINES. NOTE: κ IS THE NUMBER OF OUTPUT LINES TO BE ACTIVATED

Decoder Design	**Area** (μm^2)	**% of CiM Area**	**Power** (mW)	**Latency** (ps)	**Input Lines**	**PDP** (fJ)
Traditional	77	4.89	0.63	19	5	12
2-Cascaded [26]	92	5.40	5.60	20	2×5	112
4-Cascaded [11]	167	7.12	8.51	32	4×5	273
Latched [28]	198	7.59	2.3	$54 \times \kappa$	5	$125 \times \kappa$
SIPO	100	5.64	2.2	$48 \times \kappa$	1	$106 \times \kappa$
κ-Grouped Hier.	**142**	**6.66**	**0.54**	**41**	**5 + 1**	**23**
Tree Decoder-1	**122**	**6.22**	**0.45**	**45**	**5 + 2**	**21**
Tree Decoder-2	**60**	**4.20**	**0.53**	**29**	**2× 5**	**16**
Hybrid	**367**	**9.05**	**2.45**	**80**	**5 + 1**	**190**

impact of decoders' energy and latency on system-wide performance.

We conducted simulations on six applications, as detailed in Section V-A, measuring the runtime latency and energy consumption of the CiM system across different decoder designs. The number of rows within the CiM region was limited to 16 and 32. This limitation ensures that the application data does not fully utilize the CiM region, making the differences among decoders more noticeable. The results of CiM system simulations for energy consumption and normalized runtime latency are respectively shown in Figs. 8 and 9. For instance, in the BIT application, where all data fit within the CiM region, the energy consumption and normalized runtime remain almost the same, except for the normalized runtime of the 2-Cascaded Decoder. The 2-Cascaded Decoder performs poorly because most operations involve more than two operands. Breaking down operations into two operands and storing intermediate results in the CiM region requires 4x runtime and slightly more energy. In the case of κ-Grouped Decoder and Tree Decoder-1, their capability of activating more than 2 rows in the form of hierarchical group combinations allows them to outperform the 2-Cascaded Decoder. This advantage is especially noticeable in the BWV, MIP-3, and MIP-6 applications. However, MSQ, SHD-3, and SHD-6 applications need more complex combinations of rows that exceed the range of supported hierarchical combinations, necessitating frequent data relocation and leading to performance degradation. Whereas the Tree Decoder-2, benefiting from a broader range of row selection options comparing to κ-Grouped Decoder and Tree Decoder-1 shows improved performance.

Notably, the standout performance at the system-level comes from the inclusion of the Latched or Hybrid Decoders. The Latched Decoder's flexibility in selecting any row combination eliminates the need for data relocation, enhancing system efficiency. This advantage is significant, highlighting that data relocation is more energy-intensive and time-consuming for the system than selecting multiple rows of the decoder over several clock cycles, despite the suboptimal circuit-level performance of the decoder. Similarly, Hybrid Decoder excels in the system-level context while demonstrating less favorable performance metrics at the circuit level, as shown in Table VI, due to its sophisticated combinational logic and latching mechanisms. As explained in detail in the Section IV-C, this decoder is specifically engineered to activate the most frequently required row combinations in a single clock cycle, covering 80-95% of use cases, varying for each tested application as determined from the analysis of application requirements. Hence reducing number of clock cycles overall to activation further improving on Latched Decoder. For remaining patterns, activation is efficiently managed through the additional latching functionality. Hence, this decoder outperforms other designs in the system-level performance by effectively minimizing data movement and reducing the clock cycles.Finally, despite the Hybrid's enhanced performance over the Latched Decoder, its impact on overall system efficiency—in terms of energy and latency reductions—remains limited, as decoder operations account for only a small portion of the total execution time of CiM operations. Nevertheless, with advancements in memory technology, the impact of these improvements is expected to become noticeable, leading to further enhancements in system performance.

In conclusion, integrating the Latch and Hybrid Decoder within our NVM-CiM framework results in a 35% average reduction in runtime and a 3% decrease in energy consumption compared to popularly reported 2-Cascaded Decoder. These findings emphasize the importance of evaluating decoder de-

signs within their operational context, rather than in isolation, providing a deeper understanding of their potential impact on NVM-CiM system-wide performance.

VI. Conclusion

In this work, we have explored the design and optimization of address decoders within NVM-CiM systems for multi-row activation of the memory array and parallel data processing. We have evaluated various decoder designs, including cascaded decoder styles, hierarchical group activation decoders, latched decoders, and hybrid decoders, within our NVM-CiM framework on the gem5 simulator. This framework has enabled the implementation of six applications at the architectural level. This phase has been vital for capturing the practical demands on decoders and integrating those insights into our circuit design strategy. A notable finding from this exploration has been the significant improvement in system performance by enhancing decoder flexibility—which has reduced the reliance on data remapping operations, despite the increase in operation time or energy consumption of the decoder.

Our co-optimization strategy has been designed to ensure that decoder functionality is finely tuned to meet the specific demands and efficiency criteria of various applications. This has been demonstrated by the superior performance of the Latched and Hybrid Decoders compared to other decoders within the NVM-CiM framework. The Latched Decoder, with its ability to directly select any output line combinations in multiple cycles, has reduced latency and power usage by eliminating unnecessary data movement. Meanwhile, the Hybrid Decoder has further improved on this by offering quicker row activation for a diverse array of line combinations, thereby boosting system performance. Consequently, this decoder stands out not only for its adaptability to application-specific requirements but also for its ability to effectively manage system resources, thus explaining its enhanced performance in system-level evaluations compared to its circuit-level drawbacks. Although the impact of these decoder operations on the overall energy and latency of CiM systems has been relatively modest compared to Latched Decoder, the potential of the Hybrid Decoder design for future advancements in CiM technology is undeniable.

Acknowledgment

This work was supported by the German Research Council (DFG) through the CIMWARE project (502196634).

References

[1] Anonymous, "Beyond von neumann," *Nature Nanotechnology*, vol. 15, no. 7, pp. 507–507, Jul 2020.

[2] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE HCS*. IEEE Computer Society, 2019, pp. 1–24.

[3] W. Wan *et al.*, "A compute-in-memory chip based on resistive random-access memory," *Nature*, vol. 608, no. 7923, pp. 504–512, 2022.

[4] A. Agrawal *et al.*, "X-sram: Enabling in-memory boolean computations in cmos static random access memories," *IEEE TCAS-I*, vol. 65, 2018.

[5] V. Seshadri *et al.*, "Fast bulk bitwise and and or in dram," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[6] B. Yan *et al.*, "Resistive memory-based in-memory computing: From device and large-scale integration system perspectives," *Advanced Intelligent Systems*, vol. 1, no. 7, p. 1900068, 2019.

[7] D. Ielmini *et al.*, "In-memory computing with resistive switching devices," *Nat. Electron.*, vol. 1, no. 6, pp. 333–343, Jun. 2018.

[8] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[9] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[10] "gem5 GitHub Repository," https://github.com/gem5/gem5, accessed: 2023-07-25.

[11] V. Jamshidi *et al.*, "MagCiM: A Flexible and Non-Volatile Computing-in-Memory Processor for Energy-Efficient Logic Computation," *IEEE Access*, vol. 10, pp. 35 445–35 459, 2022.

[12] A. Sebastian *et al.*, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, no. 7, 2020.

[13] W.-H. Huang *et al.*, "A nonvolatile al-edge processor with 4mb slc-mlc hybrid-mode reram compute-in-memory macro and 51.4-251tops/w," in *ISSCC*, 2023, pp. 15–17.

[14] S. D. Spetalnick *et al.*, "A 40-nm compute-in-memory macro with rram addressing ir drop and off-state current," *IEEE SSCL*, vol. 7, 2024.

[15] Y.-C. Chiu *et al.*, "A 22nm 8mb stt-mram near-memory-computing macro with 8b-precision and 46.4-160.1tops/w for edge-ai devices," in *ISSCC*, 2023, pp. 496–498.

[16] L. Xie *et al.*, "Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing," in *IEEE-ISVLSI*, 2017, pp. 176–181.

[17] W. Zhao *et al.*, "Cross-point architecture for spin-transfer torque magnetic random access memory," *IEEE TNANO*, 2012.

[18] B. Mohammad *et al.*, *In-Memory Computing Hardware Accelerators for Data-Intensive Applications*. Springer Nature, 2023.

[19] M. R. H. Rashed *et al.*, "Logic synthesis for digital in-memory computing," in *ICCAD*, 2022, pp. 1–9.

[20] A. A. Khan *et al.*, "The landscape of compute-near-memory and compute-in-memory: A research and commercial overview," 2024.

[21] O. Mutlu *et al.*, "Processing data where it makes sense: Enabling in-memory computation," 2019.

[22] Z. Wan *et al.*, "Accuracy and resiliency of analog compute-in-memory inference engines," *J. Emerg. Technol. Comput. Syst.*, vol. 18, 2022.

[23] B. Yan *et al.*, "On designing efficient and reliable nonvolatile memory-based computing-in-memory accelerators," in *IEDM*, 2019.

[24] H. Liu *et al.*, "Afpr-cim: An analog-domain floating-point rram-based compute-in-memory architecture with dynamic range adaptive fp-adc," 2024.

[25] S. Li *et al.*, "Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016.

[26] S. Hamdioui *et al.*, "Testing computation-in-memory architectures based on emerging memories," in *IEEE ITC*, 2019, pp. 1–10.

[27] K. Monga *et al.*, "A Novel Decoder Design for Logic Computation in SRAM: CiM-SRAM," in *IEEE-INDICON*, 2021, pp. 1–4.

[28] *SN74HCS259-Q1 Automotive 8-Bit Addressable Latches with Schmitt-Trigger Inputs*, Texas Instruments, 2020. [Online]. Available: https://www.ti.com/document-viewer/sn74hcs259-q1/datasheet

[29] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.

[30] M. Herlihy *et al.*, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. Morgan Kaufmann Publishers Inc., 2012.

[31] A. Peleg *et al.*, "Mmx technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, 1996.

[32] C. Munch *et al.*, "MBIST-supported Trim Adjustment to Compensate Thermal Behavior of MRAM," in *IEEE-ETS*, Belgium, 2021, pp. 1–6.

[33] S. B. Mamaghani *et al.*, "Smart hammering: A practical method of pinhole detection in mram memories," in *DATE*, 2023, pp. 1–6.

[34] K. Stockinger *et al.*, "Using bitmap index for joint queries on structured and text data," in *New Trends in Data Warehousing and Data Analysis*. Springer, 2008, pp. 1–23.

[35] S. F. Altschul *et al.*, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[36] F. Y. Shih, *Image processing and mathematical morphology: fundamentals and applications*. CRC press, 2017.

[37] W. E. Lorensen *et al.*, "Marching cubes: A high resolution 3d surface construction algorithm," in *Seminal graphics: pioneering efforts that shaped the field*, 1998, pp. 347–353.

[38] H. Xin *et al.*, "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, 01 2015.

[39] Y. Li *et al.*, "Bitweaving: Fast scans for main memory data processing," in *ACM SIGMOD*, 2013, pp. 289–300.

[40] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.