An Analog Compute-in-Memory Co-Processor for Low-Latency Edge LLM Inference

Aishneet Juneja JiaRong Xu Michael Cluver Jinendra Malekar
Jason Bakos Ramtin Zand
University of South Carolina
{junejaa, jiarongx, mcluver, jmalekar}@email.sc.edu, ,{jbakos, ramtin}@cse.sc.edu

***Abstract*—Autoregressive decoding in Large Language Models (LLMs) is dominated by matrix-vector multiplications (GEMV), which are memory-bound and inefficient on conventional processors, especially in edge settings with batch size one. We propose a hybrid CPU–Compute-in-Memory (CiM) architecture that targets this bottleneck through selective offloading. Specifically, we map weight-stationary GEMV operations from projection and feed-forward layers to a CiM accelerator, while retaining attention and nonlinear computations on the CPU to avoid the overheads of dynamic data movement. We build a compiler-driven system using MLIR to identify and lower GEMV operations to custom CiM instructions, enabling seamless integration with a co-processor execution model. Across GPT and Phi models, our approach achieves up to 36.9× end-to-end speedup over a CPU-only baseline. These results demonstrate that dataflow-aware, selective CiM acceleration is key to enabling low-latency LLM inference.**

I. Introduction

Large Language Models (LLMs) have become the backbone of modern AI systems, enabling a wide range of applications from conversational agents to code generation [25]. Despite their success, deploying LLMs in latency-sensitive environments remains challenging [17]. In autoregressive decoding, tokens are generated sequentially, imposing strict per-token latency constraints. Unlike training or batched inference, in edge setups, decoding mostly operates with an effective batch size of one, limiting parallelism and exposing inefficiencies in conventional architectures [21].

A key bottleneck is the dominance of matrix-vector multiplications (GEMV). Projection layers (Q, K, V, and output) and feed-forward networks repeatedly apply large, static weight matrices to input activations. These operations exhibit high weight reuse and are primarily limited by data movement, making them inherently memory-bound. As model sizes grow, moving data becomes the main cost instead of computation, which reduces hardware efficiency. In contrast, attention computations involve dynamic activation-to-activation interactions, including query-key similarity, softmax, and value aggregation over an evolving key–value cache, resulting in irregular memory access patterns and frequent updates to intermediate data [8].

Prior work has improved transformer efficiency through both algorithmic and architectural approaches. IO-aware attention kernels such as FlashAttention reduce off-chip memory traffic and accelerate attention on GPUs by improving data locality [8], [7], [20]. At the hardware level, specialized accelerators such as TPUs optimize dense matrix operations, while Compute-in-Memory (CiM) and Processing-in-Memory (PIM) architectures mitigate the von Neumann bottleneck by performing dot-product computations within or near memory [19], [5], [16]. Most recent work targets LLM inference using heterogeneous NPU–PIM [11], [10] and PIM-centric designs, which offload GEMV and attention operations to memory-centric hardware units and leverage high internal bandwidth for improved performance [2], [3], [6], [12], [13], [22], [26]. Although these approaches demonstrate substantial throughput improvements, they are primarily optimized for batched execution and often assume that a broad set of transformer operations can be efficiently mapped to in-memory hardware. However, this assumption does not hold in the latency-critical, batch size one decoding regime. In this setting, GEMV operations dominate execution time, while attention introduces dynamic data dependencies and frequent updates that introduce additional write overheads and may exacerbate endurance limitations in memristive devices [23].

We propose a hybrid CPU–CiM architecture that exploits the heterogeneous dataflow characteristics of transformer workloads. Our design offloads projection and feed-forward layers to a CiM co-processor while retaining attention and nonlinear operations on the CPU. To support this execution model, we develop an MLIR-based compilation framework that automatically identifies GEMV-dominated operations and lowers them to custom CiM instructions. This enables seamless integration of the CiM co-processor into the execution pipeline without manual intervention.

II. Background

Transformer-based LLMs consist of a sequence of decoder blocks, each comprising multi-head self-attention and a feed-forward network (FFN), followed by residual connections and normalization layers [24]. During autoregressive decoding, tokens are processed sequentially with an effective batch size of one, while previously computed key and value tensors are reused through a cache.

Each decoder block is dominated by linear transformations. Projection layers map the input token embedding $I \in \mathbb{R}^d$ into query, key, and value vectors:

$$Q = W_Q I, \quad K = W_K I, \quad V = W_V I \tag{1}$$

TABLE I
MATRIX MULTIPLICATION OPERATIONS IN DECODER BLOCKS, HIGHLIGHTING WEIGHT-TO-ACTIVATION AND ACTIVATION-TO-ACTIVATION PATTERNS.

Block	Description	Dimension
Attention Projections	W_Q	$(1 \times d) * (d \times d)$
	W_K	
	W_V	
	W_O	
Attention Head	$Q * K$	$(1 \times d/h) * (d/h \times l)$
	$V * Score$	$(1 \times l) * (l \times d/h)$
FFN	Intermediate FF	$(1 \times d) * (d \times d_{FF})$
	Output FF	$(1 \times d_{FF}) * (d_{FF} \times d)$

These projections, along with the output projection and feed-forward layers, repeatedly apply large weight matrices to input activations. In contrast, attention computations involve interactions between dynamically generated activations, including query-key similarity and value aggregation over the key-value cache.

A. Dataflow Characteristics of Decoder Execution

As summarized in Table I, a decoder block consists of $2h+6$ matrix multiplications. Under autoregressive decoding, these operations reduce to GEMV due to the batch size of one. However, these GEMV operations are not uniform in their dataflow behavior:

- **Weight-to-activation GEMV:** Projection (Q, K, V, output) and FFN layers apply large, static weight matrices (e.g., $d \times d$ and $d_{ff} \times d$) to input activations. These operations exhibit high weight reuse and align naturally with weight-stationary dataflows, making them well-suited for CiM execution.
- **Activation-to-activation GEMV:** Attention computations involve interactions between dynamically generated activations (e.g., query-key similarity and value aggregation over the key–value cache). These operations introduce irregular memory access patterns and require frequent updates to intermediate data, making them less suitable for CiM.

This distinction is critical for hardware mapping, as the efficiency of in-memory execution depends on the underlying dataflow characteristics of each operation. In this work, we exploit this observation to enable selective offloading, targeting only weight-stationary GEMV operations for CiM acceleration while retaining activation-driven computations on the CPU. It is worth noting that, for edge inference scenarios with moderate sequence lengths (e.g., < 8192), the dominant latency arises from projection and feed-forward layers rather than attention [17]. Consequently, optimizing these layers yields the largest impact on end-to-end performance.

III. PROPOSED HYBRID CPU-CIM ARCHITECTURE

We design a hybrid CPU–CiM architecture for LLM inference that leverages the distinct dataflow characteristics of decoder workloads. The system integrates a CiM co-processor alongside a conventional CPU, enabling selective offloading of computation based on operation type.

TABLE II
CIM INSTRUCTION SET FOR TRANSFORMER PROJECTIONS

Instruction	Operation	Description
`CIM.QMUL`	$y = W_Q \cdot x$	Query projection
`CIM.KMUL`	$y = W_K \cdot x$	Key projection
`CIM.VMUL`	$y = W_V \cdot x$	Value projection
`CIM.OMUL`	$y = W_O \cdot x$	Output projection
`CIM.FF1MUL`	$y = W_{FF1} \cdot x$	Feedforward layer 1 (expansion)
`CIM.FF2MUL`	$y = W_{FF2} \cdot x$	Feedforward layer 2 (projection)

In this architecture, projection and feed-forward layers are offloaded to the CiM accelerator, where weights are statically stored and matrix-vector multiplications are executed in-place. In contrast, the CPU handles attention mechanisms, nonlinear functions, and control flow. This partitioning allows the system to exploit the high internal bandwidth of CiM for GEMV operations in projection and FFN layers, while retaining the flexibility of the CPU for dynamic and control-intensive computations.

The CPU and CiM co-processor are connected through a shared LPDDR memory system. During execution, the CPU orchestrates the overall computation by issuing CiM instructions and managing data movement. Input activations are sent to the CiM unit, which performs in-memory matrix-vector multiplication using locally stored weights and writes the results back to memory. The CPU then consumes these outputs to perform attention and subsequent processing stages. This coordinated execution minimizes redundant weight transfers and enables efficient interaction between the CPU and CiM co-processor.

A. CiM ISA Extensions

We introduce a set of six dedicated instructions to offload transformer projection layers to the CiM co-processor, as listed in Table II. Since all model weights are preloaded into CIM banks, the ISA does not need to specify weight addresses. Instead, each instruction only provides a pointer to the input activation vector, a pointer to the output buffer, and a bank identifier that selects the appropriate CIM-resident weight matrix. Each instruction corresponds to a specific projection type (query (Q), key (K), value (V), output projection (O), and the two feedforward layers (FF1 and FF2)), allowing a direct mapping from transformer layers to hardware execution.

All instructions share a uniform operand format, as detailed in Table III, and are executed by routing the input vector to the specified CIM bank, where the matrix-vector multiplication is performed using locally stored weights. The result is written back to output vector register. Execution can be implemented as either blocking or non-blocking, with completion detected via polling or interrupts. This design minimizes instruction complexity while leveraging the static nature of LLM weights to eliminate the need for explicit weight movement or descriptor-based offloading.

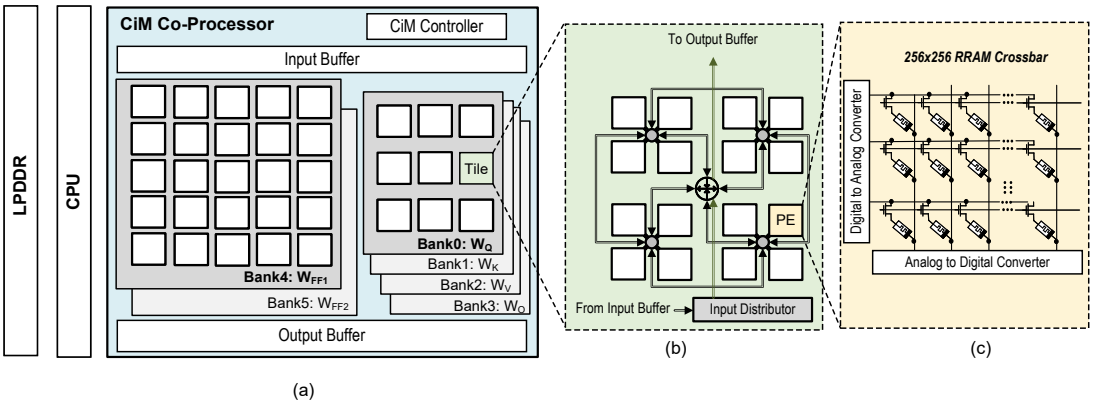

Fig. 1. The proposed hybrid CPU-CiM architecture. (a) The CiM Co-processor architecture with dedicated banks for projection and FF layers. (b) The CiM tile consists of a network of processing elements (PEs). (c) Each PE include four 256×256 RRAM crossbars to perform GEMV operations.

TABLE III
ENCODING FORMAT OF CIM INSTRUCTIONS

Bits	Field	Description
$[31:26]$	Opcode	One of the six `CIM` instructions
$[25:18]$	rX	Input vector pointer register
$[17:10]$	rY	Output vector pointer register
$[9:4]$	Bank ID	Selects CIM bank holding the weight matrix
$[3:0]$	Reserved	Reserved for future extensions

B. CiM Architecture and Execution Model

The proposed architecture augments a conventional processor with a tightly integrated CiM co-processor, as illustrated in Fig.1, designed to accelerate transformer projection layers. The CiM unit is organized into multiple banks, each statically storing the weights of a specific projection (e.g., Q, K, V, output, and feedforward layers). A lightweight CiM controller interfaces with the processor and is responsible for decoding CiM instructions, selecting the appropriate bank, and orchestrating data movement between the processor and the CiM arrays. By keeping weights within the CiM banks, the architecture eliminates repeated weight transfers through the memory hierarchy during inference, significantly reducing data movement, alleviating memory bandwidth pressure, and improving energy efficiency.

The processor interacts with the CiM co-processor through dedicated instructions that specify the input activation, output buffer, and target bank. Upon issuing a CiM instruction, the CiM controller marks the request as active by setting a memory-mapped status flag (or a dedicated status register) and begins execution. The processor monitors completion using a polling mechanism, repeatedly checking the status flag until the operation completes, at which point the CiM controller updates the flag and commits the result to memory. While completion could alternatively be handled via interrupts, we adopt a non-blocking, polling-based approach due to the short latency and high frequency of projection operations in autoregressive LLM inference. Interrupts would introduce non-trivial overheads, including pipeline disruption, context switching, and interrupt handling latency, which can outweigh the compute time of individual CiM operations. Since CiM operations have short execution latency, the overhead of interrupt handling can exceed the execution time of the computation itself. As a result, interrupts introduce disproportionate overhead for such fine-grained tasks. In contrast, polling enables low-overhead completion detection and allows the processor to overlap CiM execution with independent instructions, providing a more efficient and predictable execution model for fine-grained, latency-critical workloads.

IV. SYSTEM SUPPORT

To enable end-to-end execution of transformer workloads on the proposed hybrid CPU–CiM architecture, we build a compiler-driven system stack that bridges high-level model descriptions with CiM-aware execution. Our approach leverages the Multi-Level Intermediate Representation (MLIR) [14] to systematically identify and selectively offload projection and feed-forward operations to the CiM accelerator, while preserving the remaining computation on the CPU.

Starting from a PyTorch-based implementation, the model is lowered by Torch-MLIR [15], where tensor operations are represented in the `linalg` dialect. This level of abstraction provides structured operations along with explicit tensor shapes. However, in the default lowering, both projection layers and attention computations are represented uniformly as `linalg.batch_matmul` operations based on tensor rank, even when the batch dimension is one. This shape-driven representation obscures the distinction between weight-to-activation operations, which involve static model parameters, and activation-to-activation computations arising in attention, both of which may exhibit similar tensor structures.

To address this, we adjust the model definition in PyTorch to remove unit batch dimensions and converts these operations into standard matrix multiplication forms. Specifically, we eliminate redundant batch dimensions in projection and feed-forward layers, ensuring that these operations are lowered as `linalg.matmul` operations with a unit dimension. In contrast, attention computations are preserved in their original batched form as `linalg.batch_matmul`. This transfor-

mation ensures that only projection and feed-forward layers are exposed as candidates for CiM offloading, while attention computations remain on the CPU.

Within the MLIR pipeline, we introduce a CiM-aware transformation stage at the `linalg` level, which provides an appropriate balance between abstraction and structural visibility. Higher-level dialects obscure operation semantics behind generic tensor representations, while lower-level dialects decompose computations into loops and memory operations, losing the structure required for reliable operation classification. The linalg dialect preserves both operation type (e.g., matmul, batch_matmul) and tensor shape, enabling reliable distinction between projection-layer GEMV and attention computations. This makes it the ideal insertion point for introducing CiM-specific dialect transformations. During lowering from the linalg dialect, we apply dimensional constraints to ensure that only operations with at least one unit dimension are selected for offloading. This constraint reflects the defining characteristic of matrix-vector multiplication and prevents incorrect mapping of higher-dimensional matrix multiplications, such as those in attention heads, onto the CiM accelerator.

To formalize this behavior, we define a custom *CiM dialect* that captures the semantics of in-memory matrix-vector execution. The dialect includes a verifier that ensures that only operations with a unit dimension are admitted. This provides a correctness guarantee and prevents invalid transformations by emitting errors. The dialect also introduces explicit operations for invoking CiM execution and retrieving results, abstracting the interaction between the CPU and the CiM co-processor within the compiler IR.

At the backend, operations expressed in the CiM dialect are lowered into custom CiM instructions corresponding to the ISA extensions described earlier (e.g., `CIM.QMUL`, `CIM.FF1MUL`). These instructions directly map projection and feed-forward layers to the CiM accelerator, while all remaining operations including attention score computation, softmax, and normalization are compiled into standard CPU instructions. This results in a unified execution model where the CPU orchestrates control flow while selectively offloads weight-stationary GEMV kernels to CiM.

V. Experiments and Results

A. Experiment Set-up

We evaluate our proposed architecture with fixed batch size of one, which reflect real-time deployment scenarios. Since transformer decoders consist of multiple repeated blocks, we construct a reduced model containing a single decoder block. We first simulate a single decoder block and then scale the measured latency linearly by the actual number of blocks to estimate end-to-end inference latency. To validate this approximation, we additionally perform full-model simulation for selected configurations and observe that the scaled results closely match the full execution profile.

To construct the baseline workload, we generate a reduced transformer model containing a single decoder block for each target configuration using a PyTorch-based implementation derived from pretrained model configurations. The model is then lowered through the MLIR-to-LLVM compilation pipeline, targeting x86, to produce executable binaries. These binaries are then simulated using gem5 [4], to establish baseline latency measurements on a conventional processor. The total end-to-end latency for the CPU-only baseline is computed by scaling the per-block latency with the number of decoder layers.

TABLE IV
Targeted LLMs and their architectural parameters.

Model	Params	$\mathbf{D_{model}}$	$\mathbf{D_{ff}}$	$\mathbf{N_{layers}}$
GPT-2 Small	124 M	768	3072	12
GPT-2 Medium	355 M	1024	4096	24
GPT-2 Large	774 M	1280	5120	36
GPT-2 XL	1.5 B	1600	6400	48
Phi-1	1.3 B	2048	8192	24
Phi-2	2.7 B	2560	10240	32

$$T_{\text{CPU}} = N_{\text{layers}} \times T_{\text{block}}^{\text{CPU}}. \tag{2}$$

To model the proposed hybrid CPU-CiM architecture, we partition the workload according to the target execution units. Projection layers and feed-forward networks are mapped to the CiM accelerator, while the remaining operations, including attention and nonlinear functions, are executed on the CPU. We emulate this partitioning by isolating these operations from the PyTorch model and regenerating binaries through the same MLIR-LLVM flow, which are then simulated in gem5 to capture CPU-side execution latency. The latency of offloaded projection and feed-forward layers is modeled separately using MNSIM [27], which captures the behavior of analog CiM crossbar arrays.

Within each transformer block, matrix–vector operations arise primarily from the projection layers (Q, K, V, and output projection) and the two feed-forward layers. We model the projection layers as two effective matrix-vector multiplications: one accounting for the parallel execution of Q, K, and V projections, and the other corresponding to the output projection. The feed-forward network is modeled as two sequential matrix-vector operations corresponding to the $d \times d_{ff}$ and $d_{ff} \times d$ transformations. Thus, the total hybrid latency is computed as:

$$T_{\text{Hybrid}} = T_{\text{CPU}}^{(\text{attn+other})} + N_{\text{layers}} \times \left(2 \cdot T_{d\times d}^{\text{CiM}} + T_{\text{FFN}}^{\text{CiM}}\right), \tag{3}$$

where $T_{d\times d}^{\text{CiM}}$ denotes the latency of a projection matrix-vector operation, and $T_{\text{FFN}}^{\text{CiM}}$ represents the combined latency of the two feed-forward operations. $T_{\text{CPU}}^{(\text{attn+other})}$ corresponds to the total CPU execution time for all decoder-layer computations after offloading GEMV operations in projection and FFN to CiM. This formulation assumes sequential execution between CPU and CiM components without overlap, providing a conservative estimate of end-to-end latency.

B. Simulation Results and Discussion

We evaluate end-to-end inference latency across a range of transformer-based models and sequence lengths. The evaluated models, summarized in Table IV, span from 124M to

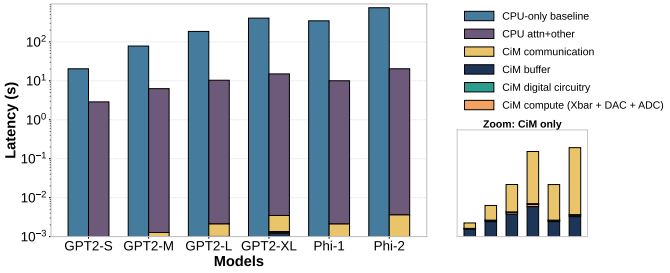

Fig. 2. End-to-end inference latency across GPT-2 and Phi models at sequence length 1024, comparing CPU-only execution with a hybrid CPU–CiM design, where the hybrid latency is decomposed into its constituent CPU and CiM components.

2.7B parameters and include both GPT-2 (Small to XL) [18] and Phi(1–2) [9] [1] configurations with varying embedding dimensions and network depths. As shown in Table V, offloading projection and feed-forward layers yields substantial performance improvements, achieving up to 36.9× speedup for larger models such as Phi-2, while smaller models like GPT-2 Small achieve 7.1× speedup at sequence length 1024.

Fig.2 reports latency breakdowns for both CPU-only and hybrid CPU–CiM configurations across different models at sequence lengths of 1024. Across all models, projection and feed-forward layers account for the majority of inference latency. This behavior arises from both the per-block computational structure and the number of decoder layers. Within each transformer block, the projection and feed-forward layers scale with the model dimensions d and d_{ff}, respectively, resulting in a large number of matrix-vector multiplications. As d and d_{ff}, increase, the cost of these operations grows significantly. Additionally, deeper models contain more decoder blocks, causing this cost to accumulate linearly across layers. This trend is reflected consistently across both GPT and Phi models. Thus, the performance gains from CiM acceleration increase with larger model dimensions (d and d_{ff}).

We further analyze the impact of sequence length on performance using Table V. As sequence length increases from 1024 to 4096, the relative speedup decreases. This trend arises because attention's GEMV computations scale quadratically with sequence length, increasing their share of total execution time. Nevertheless, projection and feed-forward layers continue to account for a substantial portion of latency, allowing the proposed hybrid CPU–CiM approach to deliver consistent performance improvements across all sequence lengths.

Overall, these results demonstrate that GEMV operations in projection and feed-forward layers are a primary bottleneck in transformer inference. By offloading these operations to CiM, the proposed architecture effectively reduces the dominant source of latency and achieves significant end-to-end performance gains.

VI. Limitations

One of the limitations is that we estimate end-to-end latency by simulating a single decoder block and scaling linearly with the number of layers. While this provides a tractable and conservative worst-case estimate, it may differ from a full-system implementation that benefits from runtime optimizations, improved scheduling, or communication–computation overlap. Secondly, our evaluation focuses on a representative set of transformer models (GPT-2 and Phi). While these models capture the key dataflow characteristics targeted in this work, we do not explicitly evaluate more recent architectures such as LLaMA or mixture-of-experts models. However, the proposed approach is not tied to a specific model family and is expected to generalize to other transformer variants that exhibit similar weight-stationary GEMV patterns. Differences in attention mechanisms, sparsity, or routing behavior may affect the magnitude of performance gains, and evaluating these models is left to future work.

TABLE V
End-to-end inference latency and speedup across models and sequence lengths.

Model	Seq Len	CPU-only (s)	Hybrid CPU–CiM (s)	Speedup
GPT-2 Small	1024	20.333	2.876	7.1×
GPT-2 Medium		78.006	6.284	12.4×
GPT-2 Large		185.278	10.337	17.9×
GPT-2 XL		406.759	15.024	27.1×
Phi-1		346.593	10.023	34.6×
Phi-2		751.096	20.367	36.9×
GPT-2 Small	2048	23.403	5.828	4.01×
GPT-2 Medium		84.990	14.130	6.01×
GPT-2 Large		198.819	24.161	8.23×
GPT-2 XL		429.176	38.202	11.2×
Phi-1		379.431	43.003	8.82×
Phi-2		789.376	59.154	13.3×
GPT-2 Small	3072	23.464	5.915	3.97×
GPT-2 Medium		83.996	13.307	6.31×
GPT-2 Large		196.564	22.029	8.92×
GPT-2 XL		424.314	33.159	12.8×
Phi-1		385.350	48.110	8.00×
Phi-2		818.244	86.853	9.42×
GPT-2 Small	4096	29.362	12.184	2.41×
GPT-2 Medium		99.010	28.358	3.49×
GPT-2 Large		223.82	49.711	4.50×
GPT-2 XL		467.353	77.192	6.05×
Phi-1		420.310	82.911	5.07×
Phi-2		854.476	122.409	6.98×

VII. Conclusion

We present a hybrid CPU–CiM architecture for low-latency LLM inference that partitions execution between a general-purpose processor and a memory-centric accelerator. By offloading weight-stationary GEMV operations while retaining attention and nonlinear computation on the CPU. Enabled by a compiler-driven MLIR stack and lightweight ISA extensions, our design achieves up to 36.9× end-to-end speedup across GPT and Phi models. These results demonstrate that selective, dataflow-aware acceleration is key to efficient LLM inference. More broadly, this work highlights the importance of cross-layer co-design in enabling practical, low-latency deployment of large models on emerging architectures.

VIII. Code Availability

The code necessary to reproduce the results presented in this paper is available at https://github.com/iCAS-Lab/llm-cim.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (NSF) under grant number 2409697.

REFERENCES

[1] M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi, S. Gunasekar, M. Javaheripi, P. Kauffmann, Y. T. Lee, Y. Li, A. Nguyen, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, M. Santacroce, H. S. Behl, A. T. Kalai, X. Wang, R. Ward, P. Witte, C. Zhang, and Y. Zhang, "Phi-2: The surprising power of small language models," *Microsoft Research Blog*, vol. 1, no. 3, p. 3, 2023.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd annual international symposium on computer architecture*, 2015, pp. 105–117.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 336–348, 2015.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[5] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[6] J. Choi, J. Park, K. Kyung, N. S. Kim, and J. H. Ahn, "Unleashing the potential of pim: Accelerating large batched inference of transformer-based generative models," *IEEE Computer Architecture Letters*, vol. 22, no. 2, pp. 113–116, 2023.

[7] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[8] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.

[9] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[10] Y. He, H. Mao, C. Giannoula, M. Sadrosadati, J. Gómez-Luna, H. Li, X. Li, Y. Wang, and O. Mutlu, "Papi: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 766–782.

[11] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park, "Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 722–737.

[12] M. Kang, H. Shin, and L.-S. Kim, "A framework for accelerating transformer-based language model on reram-based architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 9, pp. 3026–3039, 2021.

[13] A. F. Laguna, A. Kazemi, M. Niemier, and X. S. Hu, "In-memory computing based accelerator for transformer networks for long sequences," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1839–1844.

[14] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[15] LLVM Project, "Torch-mlir," https://github.com/llvm/torch-mlir, 2023.

[16] J. Malekar, P. Chandarana, M. H. Amin, M. E. Elbtity, and R. Zand, "Pim-llm: A high-throughput hybrid pim architecture for 1-bit llms," *arXiv preprint arXiv:2504.01994*, 2025.

[17] J. Malekar and R. Zand, "Amdahl's law for llms: A throughput-centric analysis of extreme llm quantization," *Transactions on Machine Learning Research*, 2025.

[18] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[19] 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.

[20] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 658–68 685, 2024.

[21] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[22] S. Sridharan, J. R. Stevens, K. Roy, and A. Raghunathan, "X-former: In-memory acceleration of transformers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 8, pp. 1223–1233, 2023.

[23] D. B. Strukov, "Endurance-write-speed tradeoffs in nonvolatile memories," *Applied Physics A*, vol. 122, no. 4, p. 302, 2016.

[24] 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.

[25] H. Xu, A. Sharaf, Y. Chen, W. Tan, L. Shen, B. Van Durme, K. Murray, and Y. J. Kim, "Contrastive preference optimization: Pushing the boundaries of llm performance in machine translation," *arXiv preprint arXiv:2401.08417*, 2024.

[26] M. Zhou, W. Xu, J. Kang, and T. Rosing, "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1071–1085.

[27] Z. Zhu, H. Sun, T. Xie, Y. Zhu, G. Dai, L. Xia, D. Niu, X. Chen, X. S. Hu, Y. Cao, Y. Xie, H. Yang, and Y. Wang, "Mnsim 2.0: A behavior-level modeling tool for processing-in-memory architectures," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 42, no. 11, pp. 4112–4125, 2023.