Tailoring LLM Weight Compression For PIM Architectures

Sabiha Tajdari Akhil Shekar Kevin Skadron Sandhya Dwarkadas
University of Virginia
{jvx2tt, shekar, skadron, sandhya}@virginia.edu

***Abstract*—Large language model (LLM) inference is increasingly limited by memory capacity and bandwidth, particularly in processing-in-memory (PIM) architectures where computation is tightly coupled with DRAM banks. These constraints are especially important in memory-constrained edge devices, where available memory capacity limits the size of deployable models. While prior compression techniques target general-purpose accelerators, many introduce irregular decoding or metadata overheads that are difficult to integrate within bank-level PIM datapaths. In this work, we investigate lightweight BF16 weight compression schemes tailored for PIM-based LLM inference. We focus on exponent-oriented compression methods that exploit the locality and redundancy present in BF16 exponent fields.**

We evaluate two practical designs: dictionary-based exponent compression and delta exponent compression. Across multiple LLMs, both approaches achieve approximately 24-25% reduction in weight storage. We further analyze the hardware implications of integrating decompression support within a PIM datapath using RTL synthesis. Our results show that dictionary and delta decompression introduce 10.29% and 8.98% additional gate counts, respectively, relative to the baseline PIM design. By reducing the memory footprint of LLM weights while remaining compatible with bank-level execution, these techniques can help memory-constrained edge systems accommodate larger models.

I. Introduction

The rapid deployment of large language models (LLMs) beyond cloud datacenters has made memory efficiency a first-order system problem. Edge devices such as laptops, phones, and embedded platforms offer privacy, low latency, and disconnected operation, but they do not provide the memory capacity, bandwidth, or parallelism available in server GPUs. As a result, LLM inference on edge platforms is often limited not by arithmetic throughput, but by the model weights' size.

The bottleneck is most pronounced during autoregressive decoding. LLM inference consists of two phases: prefill and decoding. In the prefill phase, the model processes the input prompt and can exploit substantial parallelism. In contrast, the decoding phase generates one token at a time, with each step dependent on previously generated tokens. As a result, computation in this phase is dominated by matrix–vector multiplications (GEMV) rather than matrix–matrix multiplications (GEMM), particularly in edge settings where batching opportunities are limited. This limits reuse of the weight matrix within a single request. Consequently, each token generation requires repeatedly streaming large model weights from memory, making performance and energy efficiency primarily constrained by memory bandwidth rather than compute.

Processing-in-memory (PIM) is a natural architectural response to this problem. Instead of moving all weights through the external memory interface into a distant processor, a PIM system keeps weights resident in DRAM and performs multiply-accumulate (MAC) operations close to banks that store data. The host broadcasts activation values, while memory banks compute partial products in parallel. This organization directly targets the LLM decod's memory-bound nature.

Compression is useful in a PIM system not only for increased storage capacity but also because a compressed weight format can increase the effective number of useful weights delivered per memory access. However, generic compression methods are not automatically suitable for PIM. Variable-length entropy codes, large dictionaries, and complex metadata-dependent control can create irregular latency and area overhead near the bank datapath. A PIM-friendly compression must preserve the regularity expected by DRAM timing and decompress using small, deterministic logic.

This work revisits Brain Float16 (BF16) [10] LLM weight compression under PIM constraints. We observe that BF16 exponent fields show strong local redundancy, while sign and mantissa fields are less immediately compressible without changing numerical behavior. Based on this observation, we evaluate two fixed-format exponent compression schemes in the context of PIM.

This paper makes three contributions. First, it identifies the mismatch between conventional compression schemes and the constraints of bank-level PIM for LLM inference, particularly for GEMV-dominated execution. Second, it proposes two exponent-aware compression schemes - dictionary-based and delta encoding based, that use fixed-length representations and satisfy key PIM requirements such as bounded decompression latency and small local state. Third, it evaluates both schemes across multiple LLMs, characterizing their trade-offs in compression efficiency, accuracy, and hardware overhead, and quantifies their area cost through RTL synthesis.

II. Background

Our reference PIM execution model is based on a JEDEC-compatible architecture proposed in [13], where each DRAM bank integrates a PIM unit containing a GEMV-capable SIMD MAC array. The system supports two modes: (i) *single-bank (SB)* mode, which follows a traditional load/store interface where the host accesses one bank at a time, and (ii) *all-bank (AB)* mode, where data is accessed in parallel across

all banks for PIM execution. The host configures these modes via memory-mapped control region in DRAM address space. In AB mode, bank and bank-group address bits are ignored, and row/column addresses are broadcast to all banks, enabling simultaneous data access.

In this architecture, during LLM inference's decode phase weights are stored in a bank-local, stationary manner, while input activations are broadcast from the host into each bank's register file. The PIM unit performs GEMV by streaming weights from DRAM in 256-bit bursts and computing against a 256-bit activation vector using SIMD MAC units. Each bank contains 16 SIMD lanes, processing 16 BF16 elements per t_{CCD_L} cycles.

Although prior works target HBM [7], [13], [18] and GDDR [23], the design principles extend to LPDDR systems, which share similar DRAM fundamentals with differences in hierarchy and power characteristics. Prior work [5], [9], [19] demonstrates the viability of LPDDR-based PIM, making it an attractive option for energy-constrained edge devices. We adopt the LPDDR-based AquaBolt-style [13] architecture as our baseline. Based on the above reference architecture, we define three requirements for a compression scheme to be deployable in bank-level PIM:

R1: Bounded decompression latency. The decompressor must sustain the throughput of the MAC pipeline. Variable-length encodings (e.g., Huffman coding [8]) introduce data-dependent latency and complicate scheduling and buffering. PIM-friendly formats should therefore use fixed or tightly bounded decoding.

R2: Small local state. Decompression state must be small and localized to each bank. Techniques requiring large global codebooks or cross-bank metadata increase area and control complexity. While prior work [4] shows the effectiveness of such methods in processor-centric systems, PIM designs must replicate state across many banks, amplifying these costs.

R3: Compatibility with the MAC datapath. The decompressor should output values directly consumable by existing compute units. Most PIM systems target FP16 or BF16 MAC datapaths [11], [12]. Reconstructing BF16 values avoids changes to arithmetic units while still reducing memory traffic.

These constraints highlight a key design principle: the most effective compression for PIM is not necessarily the one with the highest compression ratio, but one that enables simple, local, and deterministic decompression without disrupting PIM's compute pipeline.

A. Compression vs. Quantization

Compression and quantization have different impacts on accuracy. Quantization reduces numerical precision in order to handle outliers and scale, often impacting model accuracy. In contrast, compression exploits redundancy in representation, sometimes at the expense of outliers.

In this work, we explore both a lossless compression (dictionary compression) to preserve model fidelity and a constrained lossy variant (delta compression) that introduces bounded approximation. Compression further provides the opportunity to optimize PIM-based inference, where memory bandwidth, not arithmetic precision, is the primary bottleneck. We reconstruct BF16-compatible values prior to computation, maintaining compatibility with the MAC datapath.

III. Architecture

A. Compression Techniques

The **bfloat16 (BF16)** format [10] is widely used for LLM inference due to its balance between range and efficiency. It represents each value using 16 bits: one sign bit, eight exponent bits, and seven mantissa bits.

Prior work [29], [27] and [24] shows that this representation is not uniformly utilized. While the sign and mantissa fields exhibit high entropy, the exponent field is underutilized. Across three evaluated models, exponent values are concentrated in a narrow range (≈100–150), despite supporting 256 possible values. In contrast, sign and mantissa distributions are more uniform, making them less amenable to compression.

This observation motivates targeting the exponent field for compression. We explore two schemes: (i) dictionary compression and (ii) delta compression. These techniques are lightweight and compatible with PIM constraints, particularly low-latency decompression on the memory datapath.

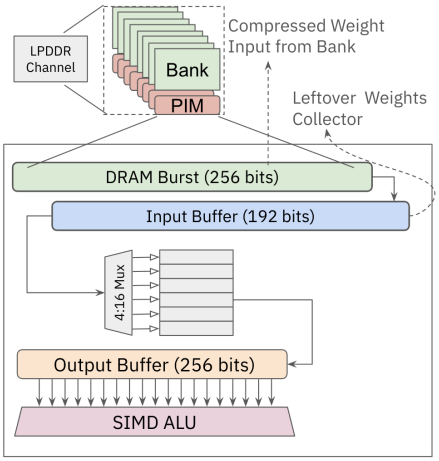
We assume weights are compressed offline and stored on disk. During inference, compressed weights are loaded into PIM-enabled DRAM banks. We introduce a decompression unit on the read path between DRAM and SIMD MAC units, enabling on-the-fly decompression without modifying the compute pipeline. The following subsections describe the design and execution of each scheme.

B. Dictionary Compression

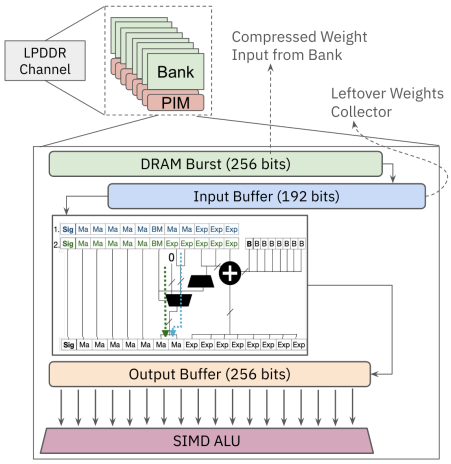
Dictionary compression encodes frequently occurring exponent values using a fixed-length codebook. To satisfy the R1 constraint (fixed-length encoding), we employ a uniform codeword size and organize weights into fixed-size windows aligned with DRAM constraints (e.g., burst size and row buffer size, both powers of two).

Each PIM bank operates on a window of 2048 elements and is provisioned with a local dictionary. An internal counter tracks elements processed; upon reaching the window boundary, the subsequent DRAM burst is interpreted as metadata containing the dictionary for the next window. Based on our design space exploration (section V-B), we use a 4-bit codeword (16 entries), resulting in a 12-bit compressed representation per weight: 1 sign bit, 7 mantissa bits, and 4 encoded exponent bits.

The decompression hardware as shown in Figure 1a consists of a register-based dictionary table (128 bits total) and combinational multiplexer trees capable of performing 4 lookups per cycle. Given that the baseline PIM architecture executes 16-wide SIMD MAC operations every $t_{CCD\_L} = 4$ cycles, the decompressor sustains full throughput by decoding 16 values over 4 cycles. This enables pipelined execution where decompression overlaps with computation, effectively hiding



(a) Bank-level PIM with 16-entry dictionary decompression.



(b) Bank-level PIM with delta decompression.

Cmd	DRAM Burst	Bits Proc.	Decode Buffer	Leftover Input
0	256	192	256	64
1	256	192	256	128
2	256	192	256	192
3	0	192	256	0
4	256	192	256	64
5	256	192	256	128
6	256	192	256	192
7	0	192	256	0

(c) Register occupancy under `tCCD_L`-spaced column reads.

Fig. 1: Architecture of bank-level PIM decompression support for PIM-based LLM inference. The dictionary and delta designs add lightweight decompression logic near the bank-level compute path, while the buffer schedule shows how compressed DRAM bursts are accumulated and decoded into fixed-width compute operands.

decode latency and preserving the downstream SIMD datapath. The evaluations described in Section IV demonstrate that the critical path delay of the proposed hardware is sufficiently low to support the pipelined design under the specified timing constraints.

From a compressed 256-bit DRAM burst, 192 bits are decompressed, resulting in an output of 256 bits that is then fed to the SIMD ALUs. In order to fully utilize the entire burst, we introduce a 192-bit input buffer that accumulates leftover data across three DRAM bursts. After those bursts, the input buffer contains sufficient compressed data to fully utilize the SIMD ALUs during decompression and execution (Figure 1c). The memory controller then waits for a duration of tCCD_L to allow the PIM unit to decompress and consume the remaining buffered data before issuing the next burst from DRAM.

To process the weights in the order in which the weights are streamed from DRAM, the incoming DRAM burst is read into the input buffer in a rolling fashion and weights shifted into the dictionary decompressor.

An additional 256-bit output buffer stages decompressed values for SIMD consumption. In total, the decompressor requires 576 bits of storage per PIM unit (192-bit input buffer, 256-bit output buffer, 128-bit dictionary).

Windows that cannot be compressed are stored without modification. The memory controller receives a bitmap in which each bit encodes the compression status of the corresponding window. During execution, the controller traverses the bitmap sequentially. For windows marked as uncompressed, it adheres to the standard AB-mode PIM execution timings, whereas for compressed windows, it follows the execution sequence outlined in Figure 1c.

C. Delta Compression

Delta compression represents exponent values relative to a window-level mean, exploiting their narrow dynamic range. Each value is encoded as a fixed-length delta from the mean, with an additional bit to indicate outliers that cannot be represented within the chosen bitwidth.

To satisfy R1 and R2 (fixed-length encoding and localized decoding), we use a 12-bit compressed format: 1 sign bit, 7 mantissa bits, 3 delta-encoded exponent bits, and 1 outlier flag. For non-outliers, the exponent is reconstructed by adding the delta to the window mean. For outliers, the encoding dynamically reallocates bits by reducing the mantissa from 7 to 5 bits and expanding the exponent from 3 to 5 bits, enabling representation of larger deviations.

Since all encodings are fixed-width and referenced to a shared mean, decompression is fully parallel and incurs no serialization overhead. The decompressor architecture for delta compression is represented in Figure 1b. Similar to dictionary compression, we employ a 192-bit input buffer and a 256-bit output buffer, along with the same 4-burst execution pattern (three bursts for data accumulation and one for buffer draining) similar to that shown in Figure 1c.

IV. Methodology

We evaluate the proposed compression schemes on Phi2 [17] and TinyLlama-1.1B [25] LLMs using BF16 weights as the baseline representation.

Dictionary Compression. For dictionary-based encoding, we evaluate the impact of window size and dictionary capacity on compression efficiency. The encoding uses fixed-length codewords to ensure compatibility with PIM constraints.

Delta Compression. Weights are partitioned into non-overlapping windows of size W. For each window, we select a base exponent B that maximizes coverage of values whose

8-bit BF16 exponent lies within $[B, B+2^r-1]$. Values within this range are encoded using an r-bit offset, while outliers are flagged and encoded using expanded exponent representation. This phase is a one-time offline process performed during weight storage.

We evaluate compression efficiency as storage reduction relative to BF16. Model quality is assessed on two metrics: perplexity and downstream task accuracy. Perplexity measures how well the model predicts held-out text, while zero-shot accuracy measures whether the same model can solve downstream tasks without task-specific fine-tuning; thus, the former reflects language modeling fidelity, whereas the latter reflects task-level generalization. Perplexity is measured on WikiText-2 [15], Penn Treebank (PTB) [14], and C4 [20] using sliding-window evaluation (context length 2,048, stride 512, over 16,384 tokens) using a representative configuration of $r = 3$ and $W = 4096$. Evaluating across three corpora with distinct statistical properties, encyclopedic prose, newswire, and web text, ensures degradation is not an artefact of a single domain. Downstream accuracy is reported on five zero-shot benchmarks: HellaSwag [28] (HS), ARC-Easy and ARC-Challenge [2], WinoGrande [21] (WG), and MMLU [6], evaluated with lm-evaluation-harness [3]. The low change in accuracy relative to the baseline demonstrates that the technique has negligible downstream quality loss (Table I).

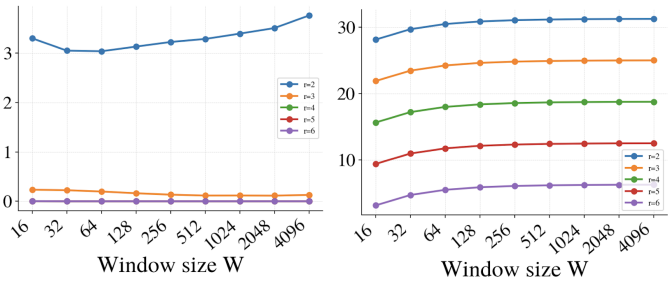
To explore the compression–accuracy trade-off, we sweep $r \in [2, 6]$ and $W \in [16, 4096]$, reporting compression ratio and the fraction of outliers for each configuration.

Hardware Evaluation. We synthesize the decompression hardware using the Yosys RTL synthesis suite [26] and estimate area using the NanGate45 open-cell library [1]. Table III reports the resulting hardware overheads in terms of equivalent gate counts. The estimates are derived from RTL-level implementations and provide a first-order approximation of hardware cost. We first implement the baseline PIM architecture in RTL following the specifications described in [13] and its associated simulator model [22]. We then extend the baseline design with the proposed dictionary- and delta-based decompression units and synthesize the resulting designs to estimate their relative area overheads. The critical path delays of the dictionary decompression and the proposed delta compression architectures are 315.64 ps and 418.53 ps, respectively. These delays are sufficiently small to sustain over 16 lookup operations within tCCD_L=4 cycles at 200 MHz. A more detailed physical design study, including DRAM integration constraints, is left for future work.

V. Results

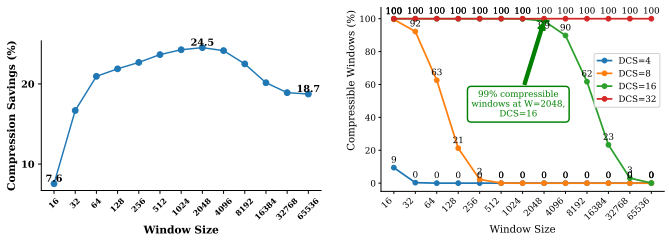
A. Delta Exponent Compression Parameter Sweep

Figure 2 shows the sensitivity to the delta bit-width behavior r in the Phi-2 [17] LLM model. A small r increases the savings but creates more outliers (with rate less than 5%). A large r reduces outliers but spends more bits on compressed values. This behavior is consistent among multiple models tested. The useful PIM design point is therefore the smallest field that keeps outlier density low enough for model accuracy.



(a) Outliers/window vs. r (b) Compression savings vs. r

Fig. 2: Storage savings and outlier fraction of delta compression as a function of window size W and exponent range r for Phi-2 3B model [17]. (Left) Fraction of outlier weights per window. (Right) Compression savings relative to BF16. Larger exponent ranges reduce outliers at the cost of lower compression. At $r = 3$, the scheme achieves approximately 25% storage savings with fewer than 0.1% outliers .



(a) Compression savings vs. window size. (b) Compressible window fraction vs. window size for different DCS (Dictionary Codebook sizes).

Fig. 3: Design-space exploration for dictionary compression. A 2048-element window and 16-entry codebook are selected to balance compression savings and hardware overhead.

1) Latency Interpretation: The compressed pipeline introduces a one-time base-set read at the start of each 32 grouped weight window pack, which adds a single bus transaction of latency before the first MAC vector can issue. In steady state, however, this overhead is fully amortized: each subsequent 256-bit compressed burst delivers >21 weights (21 weights×12 bits per weight = 252 bits) through the decompressor rather than the 16 weights of a direct BF16 read, resulting in fewer (∼25%) bus transactions necessary to maintain MAC utilization.

The decompressor runs between each bus burst and the MAC unit that consumes it, so it must keep up with the bus to avoid introducing stalls. Based on our hardware evaluation discussed in Section IV, this timing requirement is satisfied.

Using delta compression, one 8-bit base exponent is stored per 4096-weight window. Grouping 32 windows into a packet packs all base exponents into one 256-bit burst, covering 131,072 weights. Amortized over 6,144 compressed-data bursts, this adds only 0.016% memory-transaction overhead.

TABLE I: Perplexity and Zero-shot accuracy comparison between original and delta decompressed weights ($r = 3$, $w = 4096$).

Model	Variant	WikiText-2 Perplexity			Zero-shot Accuracy (%)					Mean Δ ZS
		PPL	Δ PPL	Change	HS	ARC-E	ARC-C	WG	MMLU	
TinyLlama-1.1B	Original BF16	6.489	0.000	0.00%	60.56	54.71	32.59	60.30	24.65	0.00
TinyLlama-1.1B	Delta Comp.	6.504	+0.015	+0.23%	60.47	54.50	32.76	60.30	24.76	-0.004
Phi-2	Original BF16	8.194	0.000	0.00%	73.28	78.54	53.67	76.09	54.38	0.00
Phi-2	Delta Comp.	8.226	+0.032	+0.39%	73.20	78.20	53.41	75.14	54.23	-0.36

TABLE II: Delta vs. dictionary compression savings.

Model	Delta Savings (%)	Dictionary Savings (%)
TinyLlama-1.1B	25.4	24.12
Phi-2	25.4	23.32

TABLE III: Gate count across decompression scenarios

Scenario	Equivalent Gate Count	Increase/Decrease (%)
Baseline PIM	75,847	–
PIM with Dictionary Decompression	83,650	10.29%
PIM with Delta Decompression	82,664	8.98%

B. Dictionary Compression Parameter Sweep

We perform a joint design space exploration over window size and dictionary capacity to identify a configuration that balances compression efficiency with hardware constraints. The Llama 3.2 1B model [16] is employed for design-space exploration. Since the findings closely mirror those obtained from the remaining evaluated models, we present only this model's results to maintain figure legibility.

Figure 3a shows the impact of window size on compression savings. Small windows incur higher metadata overhead, limiting achievable compression. As the window size increases, compression improves due to better reuse of exponent values, reaching a peak around 2048 elements. Beyond this point, compression degrades as larger windows introduce greater value diversity, reducing the effectiveness of a fixed-size dictionary. Based on this trend, we select a window size of 2048, which provides near-maximum compression while avoiding diminishing returns at larger sizes.

Figure 3b shows the fraction of compressible windows for different dictionary sizes. Smaller dictionaries (4–8 entries) provide limited coverage, with a significant fraction of windows failing to meet compression criteria. Increasing the dictionary size improves coverage substantially; however, the benefit saturates beyond 16 entries. At a window size of 2048, a 16-entry dictionary achieves ∼99% coverage, leaving only a small fraction of windows to be handled as outliers. Increasing the dictionary size further to 32 entries yields negligible improvement in coverage while increasing storage and lookup complexity.

From a hardware perspective, both parameters must align with PIM constraints. A 2048-element window maps cleanly to DRAM access granularity (e.g., multiple 256-bit bursts across banks), enabling regular data movement. Similarly, power-of-two dictionary sizes simplify indexing and support fixed-length encoding. Considering both compression behavior and hardware cost, we select a configuration with a 2048-element window and a 16-entry dictionary.

Across models, this configuration achieves ∼24% compression savings with consistent behavior across workloads as show in Table II.

C. Area Overhead

Using the area evaluation methodology described in Section IV, Table III compares the equivalent gate counts of the baseline PIM architecture against the decompressor-augmented designs. The results show that integrating dictionary-based decompression increases the gate count by 10.29% over the baseline, whereas delta-based decompression introduces a lower overhead of 8.98%.

The higher overhead of dictionary decompression primarily arises from the storage and lookup logic required for the dictionary table. The design includes additional registers to store the 16-entry codebook, with each entry represented using 8 bits, along with combinational logic for parallel lookup and decoding. In contrast, delta decompression relies mainly on lightweight arithmetic operations and simple control logic, resulting in lower register usage and reduced overall gate count.

D. Comparison between Methods

We compare dictionary and delta compression along three key dimensions: area overhead, compression efficiency, and accuracy in Tables I to III.

Delta compression achieves slightly lower area overhead and generally provides higher compression savings by exploiting the narrow dynamic range of exponent values within a window. However, in the presence of outliers, it may introduce small accuracy degradation due to its lossy encoding.

Dictionary compression, in contrast, provides a lossless alternative that preserves model fidelity. While it achieves slightly lower compression savings compared to delta, it offers a more robust solution when strict accuracy guarantees are required. Thus, the choice between the two methods depends on the target design point: delta compression is preferable for maximizing memory space efficiency, whereas dictionary compression is better suited for accuracy-critical deployments.

E. Limitations

This study focuses on BF16 weight compression for bank-level PIM execution, so its conclusions are most directly

supported for memory-bound, low-batch LLM decoding. Our evaluation covers two LLMs and primarily uses WikiText-2 perplexity; other models, downstream tasks, numerical formats, or deployment settings may show different accuracy and compression behavior. The expected performance benefit also depends on weight streaming being the main bottleneck, so gains may be smaller for highly batched or compute-bound workloads. Finally, our hardware results are based on RTL synthesis with an open library and do not include physical design, timing closure, power, thermal, or DRAM-integration analysis, which we leave for future work.

VI. Conclusion

This work investigates lightweight LLM weight compression techniques tailored for bank-level PIM architectures. We evaluate two exponent-oriented compression schemes: dictionary-based compression and delta exponent compression. Experimental results show that BF16 exponents exhibit sufficient locality across LLM weights to enable effective compression, achieving approximately 24–25% memory reduction with negligible accuracy drop across evaluated models.

We further analyze the hardware overheads of integrating these schemes into a PIM datapath. Dictionary-based decompression incurs a 10.29% increase in equivalent gate count, while the delta-based design requires 8.98% additional gates.

Overall, the results demonstrate that PIM-aware compression formats can significantly reduce LLM memory footprint while remaining compatible with bank-level execution, enabling larger models on memory-constrained edge systems. To support reproducibility, we provide the implementation artifacts at https://github.com/Tajdari-S/PIM-LLM-Compression.

VII. Acknowledgments

This work is funded in part by the National Science Foundation (NSF) under collaborative awards CCF-2312739 as well as PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program, sponsored by DARPA.

References

[1] "Nangate 45nm open cell library," http://www.nangate.com/?page_id=2325, 2008, open-source 45nm standard cell library based on FreePDK45.

[2] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? Try ARC, the AI2 reasoning challenge," *arXiv preprint arXiv:1803.05457*, 2018.

[3] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," https://github.com/EleutherAI/lm-evaluation-harness, 2021.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2016.

[5] S. He, Z. Zhu, Y. He, and T. Jia, "Lp-spec: Leveraging lpddr pim for efficient llm mobile speculative inference with architecture-dataflow co-optimization," 2025. [Online]. Available: https://arxiv.org/abs/2508.07227

[6] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *International Conference on Learning Representations*, 2021.

[7] 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*, ser. ASPLOS '24. ACM, Apr. 2024, p. 722–737. [Online]. Available: http://dx.doi.org/10.1145/3620666.3651380

[8] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[9] M. A. Ibrahim, M. Islam, and S. Aga, "Pimnast: Balanced data placement for gemv acceleration with processing-in-memory," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 970–981.

[10] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[11] S. Kim *et al.*, "FIMDRAM: A function-In-Memory DRAM architecture for efficient ai processing," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2021, pp. 1–3.

[12] Y. Kwon *et al.*, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based accelerator-in-memory supporting 1TFLOPS MAC operation and various activation functions for deep-learning applications," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2022, pp. 1–3.

[13] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[14] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[15] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *International Conference on Learning Representations*, 2017.

[16] Meta, "Llama-3.2-1b-instruct," https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct, 2024, accessed: 2026-04-29.

[17] Microsoft, "Phi-2," https://huggingface.co/microsoft/phi-2, 2023, accessed: 2026-04-29.

[18] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn, "Attacc! unleashing the power of pim for batched transformer-based generative model inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 103–119. [Online]. Available: https://doi.org/10.1145/3620665.3640422

[19] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon, J. Kim, J. Lee, Y. Cho, Y. Tai, J. Cho, H. Song, J. H. Ahn, and N. S. Kim, "An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 970–982.

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[21] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, "WinoGrande: An adversarial Winograd schema challenge at scale," *Communications of the ACM*, vol. 64, no. 9, pp. 99–106, 2021.

[22] Samsung Advanced Institute of Technology (SAIT), "PIMSimulator," https://github.com/SAITPublic/PIMSimulator, 2024, accessed: 2026-04-30.

[23] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, and J. Kim, "Ianus: Integrated accelerator based on npu-pim unified memory system," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. ACM, Apr. 2024, p. 545–560. [Online]. Available: http://dx.doi.org/10.1145/3620666.3651324

[24] M. Sun, A. Kanani, K. Shroff, and U. Ogras, "Lexi: Lossless exponent coding for efficient inter-chiplet communication in hybrid llms," 2026. [Online]. Available: https://arxiv.org/abs/2603.15589

[25] TinyLlama Team, "Tinyllama-1.1b-chat-v1.0," https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0, 2024, accessed: 2026-04-29.
[26] C. Wolf, J. Glaser, and J. Kepler, "Yosys–a free verilog synthesis suite," in *Proceedings of the Austrian Workshop on Microelectronics (Austrochip)*, 2013.
[27] P. Yubeaton, T. Mahmoud, S. Naga, P. Taheri, T. Xia, A. George, Y. Khalil, S. Q. Zhang, S. Joshi, C. Hegde, and S. Garg, "Huff-LLM: End-to-end lossless compression for efficient LLM inference," *arXiv preprint arXiv:2502.00922*, 2025.
[28] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "HellaSwag: Can a machine really finish your sentence?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4791–4800.
[29] T. Zhang, J. Yi, Z. Xu, S. Shen, S. Kishore *et al.*, "70% size, 100% accuracy: Lossless LLM compression for efficient GPU inference via dynamic-length float (DFloat11)," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2025, arXiv:2504.11651.