# LLM Inference in a Flash!

**Sebastian Zhao**[*,1] **Minseo Kim**[*,1] **Coleman Hooper**[*,1] **Luca Manolache**[1]
**Michael W. Mahoney**[1,2,3] **Yakun Sophia Shao**[1] **Kurt Keutzer**[1] **Amir Gholami**[1,2]
[1]University of California, Berkeley [2]ICSI [3]LBNL
[*]Equal contribution
{sebbyzhao,minseo.kim,chooper,luca.manolache,ysshao,keutzer,amirgh}@berkeley.edu
mmahoney@stat.berkeley.edu

***Abstract*—LLM inference is increasingly limited by memory capacity, bandwidth, and KV-cache traffic, especially as applications move toward longer contexts and heavier decoding. Compute-in-Flash is a promising substrate because it offers large capacity and near-data computation, but current flash CIM devices have limited floating-point support, low off-chip bandwidth, and finite write endurance. We adapt LLM inference to these constraints through two algorithmic changes: an end-to-end integer-only decoding pipeline, including nonlinear and reduction operators, and a dictionary-based KV-cache representation that stores dynamic state as sparse codes over static flash-resident dictionaries. Across Llama-3.1-8B and Qwen-2.5-7B, the combined method exhibits limited accuracy degradation while reducing dynamic KV traffic by 15×. Analytical modeling projects up to 3.1×/2.7× short-context latency/energy savings and 4.4×/6.8× long-context latency/energy savings over a DRAM-backed NPU baseline.**

## I. Introduction

LLM inference is now a core workload for chatbots, agents, retrieval-augmented generation, and long-context applications [6], [20], [28], [36]. As requests become longer and decoding becomes more expensive, inference is increasingly constrained by memory rather than arithmetic: each generated token streams model weights and repeatedly accesses a growing KV cache [8], [18]. These workload pressures arrive as memory capacity, bandwidth, and data-movement energy scale more slowly than compute [14], [25]. Flash offers much higher density than SRAM/DRAM, but conventional use of flash for LLMs is limited by off-chip transfer costs and by the poor fit between write-intensive KV caching and flash endurance [3], [37].

Compute-in-memory (CIM) addresses the memory wall by moving computation near stored data [10], [24], [26]. Compute-in-Flash is especially attractive for LLM inference because large weights and dictionaries can remain stationary in dense arrays while the device exploits internal read bandwidth [33]. However, flash CIM also changes the algorithmic contract: high-precision floating-point operators are costly or unavailable, host-device bandwidth is limited, and frequent KV-cache writes can shorten device lifetime [25]. This motivates inference algorithms that are read-heavy, integer-friendly, and careful about dynamic state.

This paper adapts LLM decoding to flash CIM with two complementary techniques. First, we use end-to-end integer-only quantization: linear layers use W8A8 arithmetic, and Softmax, RMSNorm, and SiLU are approximated with integer polynomials. Second, we replace the dense KV cache with sparse dictionary codes (Figure 1). The dictionary is static (read-only) and flash-resident, while storing each new token requires only compact indices and coefficients. Together, these choices mitigate challenges with lack of support for expensive floating-point operations and with write endurance in Compute-in-Flash systems.

Our contributions are:

- An integer-only LLM decoding pipeline tailored to flash CIM, including nonlinear operators.
- A dictionary-based KV-cache compression method that converts write-heavy dynamic KV state into sparse codes that reference read-only flash-resident dictionaries.
- Our accuracy and system evaluation shows that with minimal accuracy degradation, we attain 15× dynamic KV compression and up to **4.4×/6.8×** energy/latency savings in long-context decoding.

## II. Related Work

Compute-in-memory has been proposed as a way to reduce LLM inference data movement by moving computation near weights and activations [25], [33], [39]. For flash-backed inference, prior systems explore several points in the design space. LLM-in-a-Flash stores model weights in SSDs and schedules inference to minimize transfers under limited SSD bandwidth [3]. Cambricon-LLM uses flash as high-density model storage for edge LLM inference, combining near-flash processing with an NPU co-processor and tiling support [37]. LLM-on-the-Palm stores weights in Compute-in-Flash for mobile devices with constrained DRAM bandwidth [15], AiF modifies the flash chip to support highly concurrent reads for LLM inference [19], and Lincoln improves the host-device interface with higher-bandwidth LPDDR communication [31]. Recent work also studies KV placement and endurance for SSD-backed decoding: KVNAND places model weights and KV cache on SSDs while optimizing placement for lifetime, and HiFC quantifies the write pressure from long-context KV swapping [5], [12]. These systems motivate flash as an inference substrate, but they primarily address hardware organization, scheduling, interfaces, or placement. Our work instead reshapes the inference computation so that both weights and KV-cache operations better match flash CIM constraints, especially limited floating-point support and write endurance.

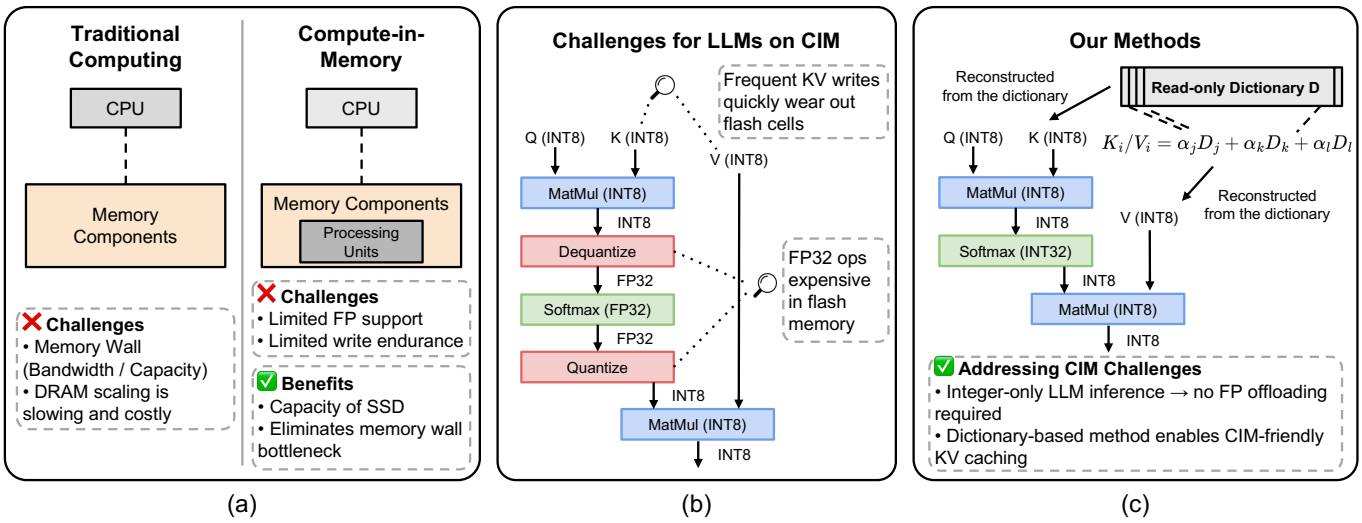


Fig. 1. Motivation and overview of LLM inference on flash compute-in-memory (CIM). (a) Comparison between traditional computing and CIM, illustrating the memory-wall bottleneck. (b) Key challenges for LLM decoding on flash CIM: frequent KV-cache writes (endurance) and costly FP32/nonlinear operations. (c) Proposed methods: end-to-end integer-only inference and dictionary-based KV caching.

Quantization methods such as SmoothQuant, AWQ, OmniQuant, and SpinQuant reduce LLM weight/activation bandwidth by controlling outliers to mitigate errors with low-precision quantization [21], [22], [29], [34]. Integer-only systems such as I-BERT and I-LLM further avoid floating-point nonlinearities [11], [17]. For KV memory, methods such as eviction, sparse attention, quantization, and dictionary coding reduce cache footprint or bandwidth [13], [16], [23], [32], [35], [38]. We combine these methods and design an inference strategy to exploit Compute-in-Flash hardware, which leverages integer-only decoding to avoid offloading floating point operations, and which uses a sparse dictionary coding method to minimize the amount of dynamic KV cache state which must be written and referenced during LLM decoding, while mapping read-only dictionaries to the CIM arrays.

## III. Algorithm

### A. Integer-Only Decoding

Flash CIM controllers provide limited support for high-precision floating-point arithmetic, so we keep both linear and nonlinear decoding operations in the integer domain. Linear layers use W8A8 quantization. To control activation outliers, we apply SmoothQuant-style activation smoothing [34] and use group-wise scales along the hidden dimension with group size 128, thereby preserving accuracy with integer quantization.

We then apply integer-only approximations for the nonlinear operations to avoid expensive floating-point computations. Softmax and SiLU use second-order polynomial approximations in the style of I-BERT [17]; RMSNorm computes $1/\sqrt{\mathrm{var}}$ using integer Newton–Raphson iterations [2]. Normalization uses fixed-point reciprocal multiply-and-shift rather than explicit division, and intermediate shifts prevent INT32 overflow. The remaining challenge is that Softmax and RMSNorm reduce across dimensions that may have different group-wise scales. We requantize their inputs into token-wise INT16 so reductions use a single scale per token. The result is an end-to-end integer-only path for LLM decoding that eliminates floating-point offloading.

### B. Dictionary-Based KV Compression

With conventional KV caching, we need to write a new key and value vector for each new token, and every subsequent token re-reads an ever-growing cache. This is poorly matched to flash memory with limited write endurance. We instead represent KV cache entries as sparse codes over static dictionaries stored in flash. We calibrate the read-only dictionary offline, and then compress incoming tokens online by projecting them onto the dictionary. The runtime also keeps a small dense local window for recent tokens and compresses tokens that age out of the window into index/coefficient pairs. Because the dictionaries are read-only and reused across requests, this representation shifts the dominant KV state from dense dynamic writes to compact code writes with most of the memory operations being static dictionary reads.

We train a separate overcomplete dictionary per layer and per key/value stream. Rather than using a learned encoder, we use a non-parameterized sparse-coding procedure and train only the decoder matrix to minimize reconstruction mean-squared-error (MSE) over buffered attention activations. After training, dictionaries are quantized to INT8 with a single per-tensor scale for flash CIM execution. During decoding, each KV vector is encoded by projection-based greedy pursuit: compute $D^{\mathsf{T}}\mathbf{r}$, select the largest-magnitude atom, use that correlation as the coefficient, and update the residual. This avoids the repeated least-squares operations required for algorithms such as OMP [27], making compression a sequence of matrix-

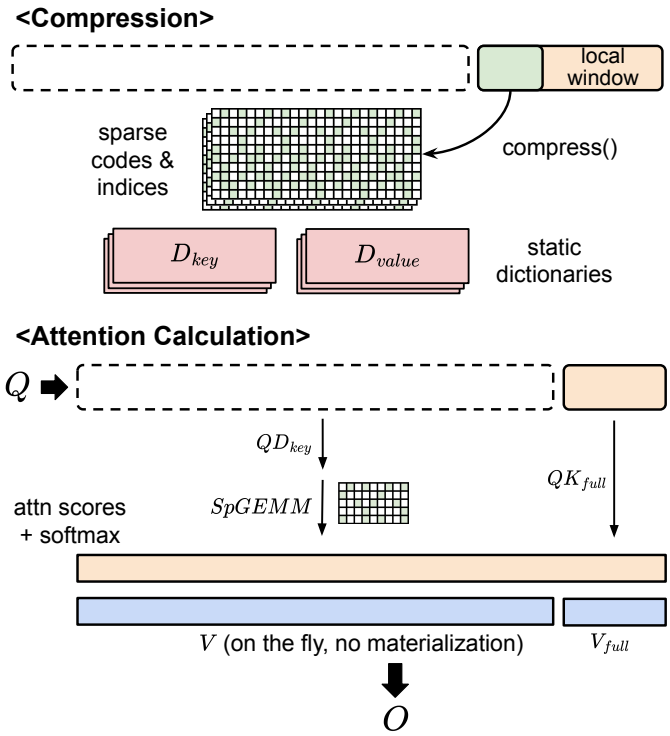


Fig. 2. End-to-end dictionary-based KV cache compression for flash CIM. Top: KV compression into sparse codes with static dictionaries. Bottom: on-the-fly attention over compressed KV without materializing the full compressed context.

vector projections and scaled updates that maps naturally to CIM.

For attention over the compressed context, we avoid materializing dense reconstructed keys and values. The CIM array computes query-dictionary projections ($QD_K^\top$), and we then combine those scores with sparse codes to form approximate logits (and similarly, the value-side output is aggregated through $D_V$ using stored coefficients). We further reduce dictionary reads with *query-aware* hierarchical sparsity: all tokens have their attention score estimated with a small reconstruction budget $K_{\text{low}}$ number of coefficients, and only the top-scoring tokens are refined to $K_{\text{high}}$.

## IV. System Modeling

We use an analytical model to compare a PCIe-attached inference NPU baseline against our CIM-SSD design. The model decomposes each decode step into compute and data movement, using published hardware parameters summarized in Appendix A. We model Llama 3.1–8B with GQA ($L$=32, $H_Q$=32, $H_{KV}$=8, $d$=128) at batch size 1, focusing on autoregressive decoding where weight and KV-cache traffic dominate.

We evaluate four configurations. **B1** is an NPU with DDR5 DRAM, FP16 weights, and a full FP16 KV cache. **B2** adds dictionary KV compression to the same NPU+DRAM platform, with dictionaries and sparse codes stored in DRAM. **S1** is our CIM-SSD: weights and dictionaries are stationary in CIM flash, the controller executes INT8/INT32 operations, and sparse codes remain on-controller. **S2** models long-context pressure by storing sparse codes in DRAM while keeping dictionary operations in CIM flash. We assume PCIe Gen4×4 (8 GB/s) between host and SSD and a 16 GB/s internal controller-to-CIM bus.

### A. Cost Model

For an operation count $N_{\text{op}}$ on an engine with throughput $P$, we model

$$t_{\text{comp}} = \frac{N_{\text{op}}}{P}, \qquad E_{\text{comp}} = N_{\text{op}} e_{\text{op}} + \left\lceil \frac{N_{\text{op}}}{S} \right\rceil e_{\text{ovhd}}, \quad (1)$$

where $S$ is SIMD width and $e_{\text{ovhd}}$ captures instruction and register-file overhead. Transfers use

$$t_{\text{xfer}} = t_\ell + t_o + \frac{\text{Bytes}}{B}, \qquad E_{\text{xfer}} = 8 \cdot \text{Bytes} \cdot e_b, \quad (2)$$

with fixed protocol terms for PCIe/internal buses and per-bit energy for DRAM and flash. When compute and transfers use independent engines, we overlap them as

$$\texttt{overlap}(t_1, t_2) = \max(t_1, t_2) + (1 - \rho)\min(t_1, t_2), \quad (3)$$

using $\rho$=0.8 for the NPU baseline and $\rho$=0.9 for CIM-SSD. Dependent stages, including QKV projection, KV update, attention, output projection, and FFN, are serialized at the decode-step level because they share engines and channels. Pursuit residual updates also remain sequential because each atom selection depends on the previous residual.

### B. Decode Data Path

Each layer follows the standard decode path: QKV projection, RoPE, KV update, attention, output projection, residual operations, SwiGLU FFN, and RMSNorm (Figure 3(b)). Linear layers are modeled as batch-1 GEMV operations. For B1, each QKV, output, and FFN projection streams FP16 weights from DRAM each token, and attention reads/writes the dense FP16 KV cache in DRAM. For CIM-SSD, weights remain stationary in flash: the controller broadcasts INT8 activations over the internal bus, the CIM array performs in-place INT8 MACs, and INT32 partial sums return to the controller for requantization. Integer nonlinear operators execute on the SSD controller, avoiding PCIe round trips for FP32 Softmax, RMSNorm, and SiLU.

### C. KV Write and Read Paths

New keys and values first enter a small local window in controller SRAM. When tokens age out of this window, the controller batch-compresses them: the residual vector is sent to the CIM array, the array computes $D^\top r$, and the controller selects atoms, updates the residual in INT32, and records sparse index/coefficient pairs. For reads, B1 scans the full dense KV cache from DRAM, while B2 reads both compressed codes and per-layer dictionaries from DRAM before reconstructing on the NPU. S1 avoids both dense-KV reads and dictionary streaming: the controller handles the uncompressed local window in SRAM, CIM computes

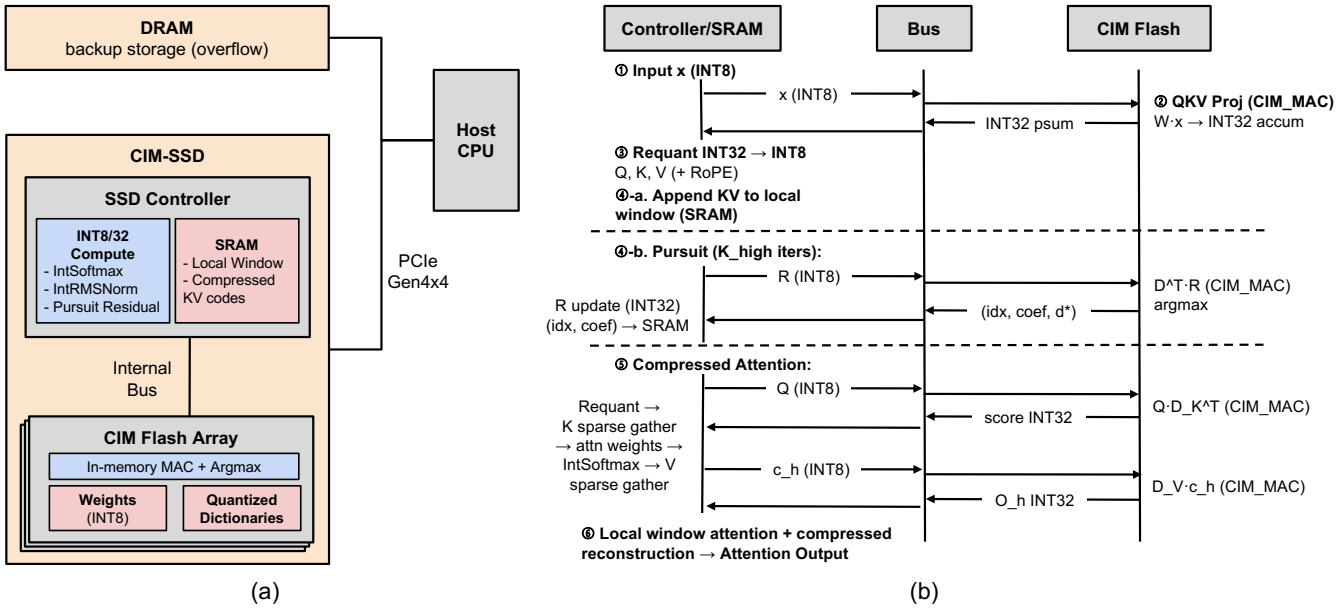


Fig. 3. System overview for analytical modeling. (a) Modeled CIM-SSD platform: a host CPU with DRAM backing and a PCIe-attached CIM-SSD comprising an SSD controller with SRAM and a CIM flash array. (b) Modeled decode pipeline and data movement across controller/SRAM, bus, and CIM flash.

$QD_K^\top$ for the compressed context, the controller combines dictionary scores with sparse codes to form logits and value-code accumulators, and CIM applies $D_V$ for the value-side contribution. The charged traffic is therefore local-window KV bytes, sparse code bytes, dictionary-score tables, partial sums, and controller/CIM transfers, rather than a materialized dense KV cache. S2 uses the same dictionary-resident computation but stores sparse codes in DRAM, modeling the regime where long contexts exceed controller SRAM capacity.

## V. Results

### A. Experimental Setup

We evaluate Llama-3.1-8B and Qwen-2.5-7B. Quantization calibration uses the Pile validation set following SmoothQuant [34], with group size 128. Dictionary training extracts key/value activations from Pile calibration sequences, trains one dictionary per layer and stream with $M$=32768 atoms, and quantizes dictionaries to INT8. Unless otherwise stated, we use a local window of $W$=128 tokens and hierarchical sparsity with $K_{\text{low}}$=3 and $K_{\text{high}}$=16.

TABLE I
Quality summary. End-to-end quality is evaluated with LongBench average score (↑). KV compression counts dynamic KV state read per decode step and excludes the static dictionary.

| Model | Configuration | LongBench | KV Compr. |
|---|---|---|---|
| Llama-3.1-8B | FP16 full KV | 44.07 | 1× |
| Llama-3.1-8B | Integer-only + dict. KV | 43.65 | 15× |
| Qwen-2.5-7B | FP16 full KV | 44.44 | 1× |
| Qwen-2.5-7B | Integer-only + dict. KV | 44.23 | 15× |

TABLE II
System-level decode latency (TBT, ms) and energy (J) for Llama-3.1-8B. B1/B2 are NPU+DRAM baselines; S1/S2 are CIM-SSD variants. Parenthesized values are relative to B1.

| | 1K | | 256K | |
|---|---|---|---|---|
| System | TBT | Energy | TBT | Energy |
| B1 | 183.7 | 101.8 | 631.5 | 354.5 |
| B2 | 190.4 (0.96×) | 108.2 (0.94×) | 217.4 (2.91×) | 135.1 (2.62×) |
| S1 | 54.1 (3.40×) | 37.4 (2.72×) | 208.4 (3.03×) | 40.9 (8.66×) |
| S2 | 59.9 (3.07×) | 37.8 (2.69×) | 143.5 (4.40×) | 52.5 (6.75×) |

### B. Model Quality

Table I summarizes the end-to-end accuracy trade-off. For long-context tasks, the combined integer-only and dictionary-KV pipeline retains approximately 99% of the FP16 full-KV LongBench score for both Llama-3.1-8B and Qwen-2.5-7B, while reducing dynamic KV state by 15×. The detailed ablations are provided in Appendix B.

### C. System Impact

Table II shows that dictionary compression alone (without leveraging Compute-in-Flash) is insufficient to attain the benefits of our algorithm: B2 is slightly slower and less energy-efficient than B1 because the dictionary must still be streamed from DRAM. CIM-SSD removes that overhead by keeping weights and dictionaries stationary in flash. At 1K context, S1 improves latency by 3.4× and energy by 2.7×. At 256K context, S2 is the fastest configuration (4.4× latency improvement) because host DRAM provides more scalable sparse-code storage and CPU-side reconstruction bandwidth, while S1 remains the most energy-efficient configuration (8.7× energy reduction) by avoiding DRAM code traffic. Overall,

configuration S2 presents a more practical serving setup that can support short and long context lengths.

## VI. Conclusion and Limitations

Compute-in-Flash can reduce LLM inference data movement, but LLM inference algorithms must be adapted to match the flash CIM constraints. We show that integer-only decoding and dictionary-coded KV caching allow both weights and long-context KV operations to exploit flash-resident computation. The resulting pipeline preserves model quality while reducing dynamic KV traffic by 15× and improving projected decode latency and energy over a DRAM-backed NPU baseline.

In terms of limitations, this study focuses on generation rather than prefill, which is more compute-bound and less naturally matched to current flash CIM throughput. Our value-dictionary path also assumes both transposed and non-transposed dictionary layouts (so pursuit and attention can each use efficient near-data access), which requires an extra copy of the value dictionaries to be stored in the Flash array.

## Acknowledgment

We acknowledge the gracious support from the FuriosaAI, Intel, Apple, NVIDIA, Macronix, and Mozilla teams. Furthermore, we appreciate the support from Google Cloud, the Google TRC team, Prof. David Patterson, along with support from the Google Gemini team and Divy Thakkar. Prof. Keutzer's lab is also sponsored by funding through BDD and BAIR. We also acknowledge support by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. MWM would also like to acknowledge DARPA, DOE, NSF, ONR, and the DOE SciGPT grant. Our conclusions do not necessarily reflect the position or the policy of our sponsors, and no official endorsement should be inferred.

## References

[1] Advanced Micro Devices, Inc., *AMD EPYC 7002 Series Processors: A New Standard for the Modern Datacenter*, Feb. 2020, document ID: LE-70002-01. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/products/epyc/amd-epyc-7002-series-datasheet.pdf

[2] S. Akram and Q. U. Ann, "Newton raphson method," *International Journal of Scientific & Engineering Research*, vol. 6, no. 7, pp. 1748–1752, 2015.

[3] K. Alizadeh, S. I. Mirzadeh, D. Belenko, S. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 12 562–12 584.

[4] CORE-MATH Project, "The CORE-MATH project," https://core-math.gitlabpages.inria.fr, accessed: 2026-02-27.

[5] L. Deng, S. Xu, J. Chen, C. Yan, J. Wang, Z. Jiang, and W. Shan, "Kvnand: Efficient on-device large language model inference using dram-free in-flash computing," *arXiv preprint arXiv:2512.03608*, 2025.

[6] Z. Durante, Q. Huang, N. Wake, R. Gong, J. S. Park, B. Sarkar, R. Taori, Y. Noda, D. Terzopoulos, Y. Choi, K. Ikeuchi, H. Vo, L. Fei-Fei, and J. Gao, "Agent ai: Surveying the horizons of multimodal interaction," 2024. [Online]. Available: https://arxiv.org/abs/2401.03568

[7] A. Fog, *Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD, and VIA CPUs*, 2025, last updated 2025-09-20. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[8] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.

[9] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[10] H.-W. Hu, W.-C. Wang, Y.-H. Chang, Y.-C. Lee, B.-R. Lin, H.-M. Wang, Y.-P. Lin, Y.-M. Huang, C.-Y. Lee, T.-H. Su, C.-C. Hsieh, C.-M. Hu, Y.-T. Lai, C.-K. Chen, H.-S. Chen, H.-P. Li, T.-W. Kuo, M.-F. Chang, K.-C. Wang, C.-H. Hung, and C.-Y. Lu, "Ice: An intelligent cognition engine with 3d nand-based in-memory computing for vector similarity search acceleration," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 763–783.

[11] X. Hu, Y. Cheng, D. Yang, Z. Yuan, J. Yu, C. Xu, and S. Zhou, "I-llm: Efficient integer-only inference for fully-quantized low-bit large language models," *arXiv preprint arXiv:2405.17849*, 2024.

[12] I. Jeong, S. Woo, S. Namkung, and D. Jeon, "HiFC: High-efficiency flash-based KV cache swapping for scaling LLM inference," in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. [Online]. Available: https://openreview.net/forum?id=onhjdWCxZY

[13] H. Jiang, Y. LI, C. Zhang, Q. Wu, X. Luo, S. Ahn, Z. Han, A. H. Abdi, D. Li, C.-Y. Lin, Y. Yang, and L. Qiu, "MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://openreview.net/forum?id=fPBACAbqSN

[14] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[15] H. Kim, S. Han, and J.-J. Kim, "Llm-on-the-palm: Mobile llm inference with pim-enhanced nand flash memory," in *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2025, pp. 1–9.

[16] J. Kim, J. Park, J. Cho, and D. Papailiopoulos, "Lexico: Extreme KV cache compression via sparse coding over universal dictionaries," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=Yh9vxlxnjA

[17] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *International conference on machine learning*. PMLR, 2021, pp. 5506–5518.

[18] S. Kim, C. R. C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "SqueezeLLM: Dense-and-sparse quantization," in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=0jpbpFia8m

[19] J. Lee, H. Kim, S. Oh, M. Chun, M. Kim, and J. Kim, "Aif: Accelerating on-device llm inference using in-flash processing," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 529–543.

[20] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[21] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of machine learning and systems*, vol. 6, pp. 87–100, 2024.

[22] Z. Liu, C. Zhao, I. Fedorov, B. Soran, D. Choudhary, R. Krishnamoorthi, V. Chandra, Y. Tian, and T. Blankevoort, "Spinquant: Llm quantization with learned rotations," *arXiv preprint arXiv:2405.16406*, 2024.

[23] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu, "KIVI: A tuning-free asymmetric 2bit quantization for KV cache," in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=L057s2Rq8O

[24] Z. Lu, X. Wang, M. T. Arafin, H. Yang, Z. Liu, J. Zhang, and G. Qu, "An rram-based computing-in-memory architecture and its application in accelerating transformer inference," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 3, pp. 485–496, 2023.

[25] X. Ma and D. Patterson, "Challenges and research directions for large language model inference hardware," *arXiv preprint arXiv:2601.05047*, 2026.

[26] O. Mutlu, "Lightning talk: Memory-centric computing," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–2.

[27] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad, "Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition," in *Proceedings of 27th Asilomar conference on signals, systems and computers*. IEEE, 1993, pp. 40–44.

[28] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *Advances in neural information processing systems*, vol. 36, pp. 68 539–68 551, 2023.

[29] W. Shao, M. Chen, Z. Zhang, P. Xu, L. Zhao, Z. Li, K. Zhang, P. Gao, Y. Qiao, and P. Luo, "Omniquant: Omnidirectionally calibrated quantization for large language models," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=8Wuvhh0LYW

[30] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE micro*, vol. 40, no. 2, pp. 45–52, 2020.

[31] W. Sun, M. Gao, Z. Li, A. Zhang, I. Y. Chou, J. Zhu, S. Wei, and L. Liu, "Lincoln: Real-time 50˜ 100b llm inference on consumer devices with lpddr-interfaced, compute-enabled flash memory," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1734–1750.

[32] J. Tang, Y. Zhao, K. Zhu, G. Xiao, B. Kasikci, and S. Han, "QUEST: Query-aware sparsity for efficient long-context LLM inference," in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=KzACYw0MTV

[33] C. Wolters, X. Yang, U. Schlichtmann, and T. Suzumura, "Memory is all you need: An overview of compute-in-memory architectures for accelerating large language model inference," *arXiv preprint arXiv:2406.08413*, 2024.

[34] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International conference on machine learning*. PMLR, 2023, pp. 38 087–38 099.

[35] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=NG7sS51zVF

[36] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The eleventh international conference on learning representations*, 2022.

[37] Z. Yu, S. Liang, T. Ma, Y. Cai, Z. Nan, D. Huang, X. Song, Y. Hao, J. Zhang, T. Zhi, Y. Zhao, Z. Du, X. Hu, Q. Guo, and T. Chen, "Cambricon-llm: A chiplet-based hybrid architecture for on-device inference of 70b llm," in *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '24. IEEE Press, 2024, p. 1474–1488. [Online]. Available: https://doi.org/10.1109/MICRO61859.2024.00108

[38] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen, "H2o: Heavy-hitter oracle for efficient generative inference of large language models," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=RkRrPp7GKO

[39] Z. Zhu, H. Li, W. Ren, M. Wu, L. Ye, R. Huang, and T. Jia, "Leveraging compute-in-memory for efficient generative model inference in tpus," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

## APPENDIX A
## SYSTEM MODELING PARAMETERS

Tables III-VII summarize the hardware parameters used in our analytical system model (Section IV).

### A. Compute Energy

Table III lists per-operation energy for CPU and SSD-CIM. CPU operation energies are from [14], assuming 7nm technology node. Each CPU instruction incurs an additional $\sim$13 pJ overhead from instruction fetch and register file access, estimated from [9] (scaled down to 7nm using SRAM energy ratios from [14]). We assume a digital Flash CIM device as in [5], [31], [37]. I8 MAC energy is 0.1 pJ [14], and input/output activation buffer energy is 0.8 pJ per 8 bits (extrapolated from 7nm data in [14], assuming 2KB I/O buffer as in [37]). SSD-CIM SRAM overhead ($\sim$7 pJ per 32 bits) is estimated from 1 MB SRAM read energy [14]. FP32 Div $\approx 6\times$ FP32 MAC and FP32 Sqrt $\approx 12\times$ FP32 MAC, estimated from Agner Fog instruction tables for Zen 7 nm [7]. FP32 Exp is typically approximated using $\sim$8 FMA instructions [4], incurring $8\times$ computation and instruction overhead.

TABLE III
PER-OPERATION COMPUTE ENERGY.

| Device | Operation | Op Energy (pJ) | Overhead (pJ) |
|---|---|---|---|
| CPU | I32 Add | 0.03 | $\sim$13 |
| CPU | I32 Mul | 1.5 | $\sim$13 |
| CPU | FP32 Add | 0.4 | $\sim$13 |
| CPU | FP32 Mul | 1.3 | $\sim$13 |
| CPU | FP32 Div | 10.2 | $\sim$13 |
| CPU | FP32 Sqrt | 20.4 | $\sim$13 |
| CPU | FP32 Exp | 13.6 | $\sim 8 \times 13$ |
| SSD-CIM | I8 MAC | 0.01 | $\sim$0.8 |
| SSD-CIM | I32 Mul | 1.5 | $\sim$7 |
| SSD-CIM | I32 Add | 0.03 | $\sim$7 |
| SSD-CIM | Argmax | 0.5 | $\sim$7 |

### B. Memory Energy

Table IV reports memory access energy per bit. DRAM read/write energy (7 pJ/bit) is from [5]. SSD energy (7.9 pJ/bit) and internal Flash read energy (3 pJ/bit) are from [5].

TABLE IV
MEMORY ACCESS ENERGY PER BIT (READ/WRITE).

| Device | Energy/Bit (pJ) | Source |
|---|---|---|
| DRAM | 7 | [5] |
| SSD (Flash) | 7.9 | [5] |
| SSD (Flash, internal) | 3.0 | [5] |

### C. Bandwidth

Table V lists the memory interface bandwidths assumed in our model: 76.8 GB/s for DDR5 DRAM (Dual-Channel) and 8 GB/s for PCIe 4.0.

TABLE V
MEMORY INTERFACE BANDWIDTH.

| Interface | Bandwidth | Standard |
|---|---|---|
| CPU $\leftrightarrow$ DRAM | 76.8 GB/s | DDR5 |
| CPU $\leftrightarrow$ SSD | 8 GB/s | PCIe 4.0 |

### D. Capacity

Table VI lists our storage capacity assumptions. We assume a 512 GB SSD (32 GB per die) with 192 MB of on-chip SRAM (using 6MB/die from [31]). We assume 64GB for the DRAM capacity.

TABLE VI
STORAGE CAPACITY ASSUMPTIONS.

| Device | Capacity | Source |
|---|---|---|
| DRAM | 64GB | – |
| SSD (Flash) | 512 GB (32 GB per die) | – |
| SSD (on-chip SRAM) | 192 MB | [31] |

### E. Throughput

Table VII reports peak compute throughput. CPU throughput (2 TFLOPS FP32) is based on Zen 2 (7 nm) and is within the range of listed configurations [1], [30]. NPU throughput (32 TFLOPS BF16) assumes a $16\times$ speedup over the CPU with matrix extension, and is consistent with [31]. SSD-CIM I8 MAC throughput (1 TB/s) is from [31] (determined by peak internal read bandwidth), scaled assuming the capacity from Table VI. We assume SSD-CIM peak computation throughput for postprocessing in the peripheral logic is 20x lower than internal CIM throughput (50 GFLOPS).

TABLE VII
PEAK COMPUTE THROUGHPUT.

| Device | Peak Throughput | Source |
|---|---|---|
| CPU (FP32 MAC) | 2 TFLOPS | [1], [30] |
| NPU (BF16 MAC) | 32 TFLOPS | [31] |
| SSD-CIM (I8 MAC) | 1 TB/s | [31] |
| SSD-CIM (I32 Mul) | 50 GFLOPS | – |
| SSD-CIM (I32 Add) | 50 GFLOPS | – |

## APPENDIX B
## NONLINEAR OPERATION OFFLOADING ABLATIONS

Table VIII shows ablations for offloading nonlinear operations to the NPU vs. our approach using integer approximations in the controller across short (1k)/medium (64K)/long (256K) context lengths. Both scenarios assume a streaming implementation of nonlinear operations. Offloading uses the NPU to do the nonlinear operations, and our approach (integer ops) uses the controller to approximate nonlinear operations with integer approximations.

Table IX shows full system modeling results comparing offloading nonlinear operations to the NPU vs. our approach using integer approximations in the controller across short

TABLE VIII

NONLINEAR LAYER FORWARD PASS LATENCY (MS) AND ENERGY (J) ACROSS CONTEXT LENGTHS FOR LLAMA-3.1-8B (BATCH SIZE 1). PARENTHESIZED VALUES SHOW SPEEDUP (↑) / ENERGY REDUCTION (↑) RELATIVE TO OFFLOADING.

| | Configuration | 1K Latency | 1K Energy | 64K Latency | 64K Energy | 256K Latency | 256K Energy |
|---|---|---|---|---|---|---|---|
| Softmax | CIM, offloading | 0.018 | 6.04e-06 | 0.264 | 9.66e-05 | 4.197 | 1.54e-03 |
| | CIM, integer ops | 0.010 (1.78×) | 1.42e-06 (4.25×) | 0.147 (1.80×) | 2.27e-05 (4.25×) | 2.334 (1.80×) | 3.63e-04 (4.25×) |
| RMSNorm | CIM, offloading | 2.099 | 7.71e-04 | 33.556 | 1.23e-02 | 536.873 | 1.97e-01 |
| | CIM, integer ops | 0.664 (3.16×) | 1.47e-04 (5.25×) | 10.613 (3.16×) | 2.35e-03 (5.25×) | 169.792 (3.16×) | 3.76e-02 (5.25×) |
| SiLU | CIM, offloading | 7.342 | 2.72e-03 | 117.450 | 4.35e-02 | 1879.168 | 6.96e-01 |
| | CIM, integer ops | 2.616 (2.81×) | 5.13e-04 (5.30×) | 41.839 (2.81×) | 8.21e-03 (5.30×) | 669.412 (2.81×) | 1.31e-01 (5.30×) |

TABLE IX

SYSTEM-LEVEL DECODE LATENCY (TBT, MS) AND ENERGY (J) ACROSS CONTEXT LENGTHS FOR LLAMA-3.1-8B (BATCH SIZE 1). CIM, OFFLOADING REFERS TO OFFLOADING NONLINEAR OPERATIONS TO THE NPU; CIM, INTEGER OPS IS OUR APPROACH. PARENTHESIZED VALUES SHOW SPEEDUP (↑) / ENERGY REDUCTION (↑) RELATIVE TO OFFLOADING. NL % SHOWS THE FRACTION OF TOTAL TBT AND ENERGY ATTRIBUTABLE TO NONLINEAR LAYERS.

| Configuration | 1K TBT (ms) | 1K Energy (J) | 64K TBT (ms) | 64K Energy (J) | 256K TBT (ms) | 256K Energy (J) |
|---|---|---|---|---|---|---|
| CIM, offloading | 54.86 | 37.44 | 111.47 | 39.72 | 283.99 | 46.66 |
| NL % of total | 2.1% | 0.1% | 30.7% | 4.0% | 47.5% | 13.6% |
| CIM, integer ops | 54.53 (1.01×) | 37.41 (1.00×) | 96.48 (1.16×) | 38.49 (1.03×) | 224.33 (1.27×) | 41.80 (1.12×) |
| NL % of total | 1.5% | 0.0% | 19.9% | 1.0% | 33.5% | 3.6% |

TABLE X

SYSTEM-LEVEL DECODE LATENCY (TBT, MS) AND ENERGY (J) ACROSS CONTEXT LENGTHS FOR LLAMA-3.1-8B (BATCH SIZE 1). B1/B2 ARE NPU+DRAM BASELINES; S1/S2 ARE CIM-SSD VARIANTS. PARENTHESIZED VALUES SHOW SPEEDUP (↑) / ENERGY REDUCTION (↑) RELATIVE TO B1.

| | Configuration | 1K TBT | 1K Energy | 64K TBT | 64K Energy | 256K TBT | 256K Energy |
|---|---|---|---|---|---|---|---|
| B1 | NPU, no compr. | 183.7 | 101.8 | 294.3 | 164.2 | 631.5 | 354.5 |
| B2 | NPU + dict. compr. | 190.4 (0.96×) | 108.2 (0.94×) | 197.1 (1.49×) | 114.9 (1.43×) | 217.4 (2.91×) | 135.1 (2.62×) |
| S1 | CIM-SSD, coeff. on ctrl. | 54.1 (3.40×) | 37.4 (2.72×) | 92.2 (3.19×) | 38.3 (4.29×) | 208.4 (3.03×) | 40.9 (8.66×) |
| S2 | CIM-SSD, coeff. on DRAM | 59.9 (3.07×) | 37.8 (2.69×) | 80.6 (3.65×) | 41.4 (3.96×) | 143.5 (4.40×) | 52.5 (6.75×) |

(1k)/medium (64K)/long (256K) context lengths. Our approach results in speedup and less energy consumption, and scales better as the sequence length increases.

## APPENDIX C
## FULL SYSTEM RESULTS

Table X provides full system modeling results across short (1K)/medium (64K)/long (256K) context length configurations.