Inference Data Structures: The Missing Infrastructure of Modern AI

Abolfazl Asudeh
asudeh@uic.edu
University of Illinois Chicago
Chicago, IL, USA

Mohsen Dehghankar
mdehgh2@uic.edu
University of Illinois Chicago
Chicago, IL, USA

Abstract

Modern AI increasingly depends on efficient inference, yet most inference optimization remains centered on model compression, kernel engineering, and hardware acceleration. These approaches overlook a complementary algorithmic layer that builds *Inference Data Structures* (IDSs) that organize persistent inference state, such as model weights and KV caches, into reusable computational representations that accelerate repeated execution.

We identify two classes of IDSs: static structures over immutable model parameters and dynamic structures over evolving inference state. Recent work provides examples of both classes. Low-bit fixed-weight matrices are organized as reusable static IDS for exact matrix-vector multiplication, while dynamic IDSs built on top of KV caches support efficient retrieval during autoregressive decoding. Viewed together, these examples underscore the importance of inference data structures, alongside models, kernels, and hardware, to make inference more efficient.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; • **Theory of computation** → **Data structures design and analysis**.

Keywords

Inference Data Structures, Efficient AI Inference, Foundation Models, Large Language Models, KV Cache, Model Compression

ACM Reference Format:
Abolfazl Asudeh and Mohsen Dehghankar. 2026. Inference Data Structures: The Missing Infrastructure of Modern AI. In *ACM AI Summit 2026 (AI Summit '26), August 31-September 02, 2026, Atlanta, GA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3806096.3844820

1 Introduction

Modern AI has advanced rapidly through larger models, richer training data, and increasingly capable foundation models. Generative AI models, including Large language models and multimodal models, have become handy tools for daily tasks such as writing, reasoning, and programming. As these models become more capable and widely adopted by everyday users, *inference efficiency* is becoming critical, as it increasingly dominates modern computing workloads [15, 19, 22, 25]. Yet the infrastructure of AI inference has not matured at the same pace as the models themselves. Much of the current inference stack is organized around three strategies: (a) compress the model [22], (b) accelerate the kernels [3], or (c) provision more capable hardware [8]. These strategies are essential. Quantization reduces memory footprint and arithmetic cost [12, 21]; specialized kernels improve throughput [2, 3, 9]; GPUs and emerging accelerators provide the raw computational substrate [7, 8, 16]. However, these strategies do not fully address a deeper structural issue: inference repeatedly operates over large, persistent, and reusable state, but often treats that state as raw tensors to be processed again at every execution step.

The missing algorithmic layer. In this paper, we argue that modern AI lacks a fundamental infrastructure layer for *inference data structures*.

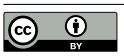
Definition 1 (Inference data structure (IDS)). *An inference data structure is a* **reusable** *index, coupled with algorithms, constructed before or during model execution that organizes inference state to reduce future computational cost while preserving correctness or providing explicit approximation guarantees. The represented state may be static, such as trained model weights, or dynamic, such as a growing KV cache during autoregressive decoding.*

Although prior approaches such as model compression, caching, and specialized hardware design already help to make inference more efficient, the essential role of inference data structures is now more vivid, as the foundation models grow bigger, more versatile, and closer to the ordinary end users than earlier neural networks. As a result, inference becomes more repeated and resource sensitive, as opposed to a specialized back-end operation.

Modern AI models are based on *repeated inference rounds over growing context* to generate tokens. They support long contexts, chain-of-thought reasoning, multimodal inputs, and tool use, which makes them increasingly resource hungry. In such settings, the absence of reusable algorithmic structures becomes *a missed opportunity* to address the resource constraints.

The core observation is that *many variables*[1] *used during the different rounds of inference are fixed and do not change.* Model weights, for example, never change after a model is trained. Such *static variables* remain the same, irrespective of the inference input (e.g., prompt) or context window. As a result, once a model is trained, one can carefully index them to reduce the cost of inference operations. For instance, the weights can be preprocessed into structures that expose repeated patterns and support faster exact multiplication.

[1] We view the model state during the inference as a set of static (e.g., model weights) and dynamic (e.g., context-window vectors such as keys/values) variables. Hence, the words, model state and model variables are used interchangeably.



AI Summit '26, Atlanta, GA, USA

ACM ISBN 979-8-4007-2638-5/2026/08
https://doi.org/10.1145/3806096.3844820

Table 1: Inference data structures as a complementary optimization direction.

	Main question	Typical effect
Model compression	How should parameters be represented numerically?	Smaller models, lower arithmetic cost, possible accuracy tradeoffs.
Kernel optimization	How should inference primitives be implemented efficiently?	Faster primitive execution on a given device.
Hardware acceleration	Where should computation run?	Higher throughput through GPUs, TPUs, NPUs, or specialized memory systems.
Inference data structures	*How should reusable inference state be represented and organized?*	Lower cost through preprocessing, indexing, partitioning, summarization, and efficient query processing.

Another example is the context window variables. While the context window is query-dependent and varies across different prompts, it keeps growing within the repeated rounds of inference to answer one query. That is, the existing context within the window is fixed and does not get updated or removed; only new context gets appended to the window. As a result, rather than viewing it as an append-only tensor, one can construct an index on-the-fly for such *dynamic variables*, and update it as new tokens are generated. Beyond these examples, reusable inference state may also arise at different stages and system-level developments, including multimodal context representations, tool-use traces, and long-lived interaction state.

This perspective is analogous to common practices across various domains of computer systems to enable efficient and scalable operations. Database systems, in particular, develop data structures such as B-trees and Bitmap indices that enable efficient processing of queries without fully parsing the data [1, 13]. Similarly, computer graphics relies on acceleration structures such as bounding-volume hierarchies [14], and compilers construct intermediate representations and control-flow structures that expose reusable program structure [20]. AI inference now appears to be reaching a similar point and facing similar challenges, creating the opportunity to build on top of the rich advancements in these domains to achieve efficiency and scalability in modern AI inference.

The Vision. Figure 1 summarizes our vision, where inference data structures form an algorithmic infrastructure layer between foundation-model inference and the runtime. This layered perspective isolates efficiency optimizations at different layers. As a result, techniques such as model compression and quantization apply at the top layer to reduce the model size. Inference data structures take a model as the input and develop static and dynamic indices that benefit from reusable states. Finally, at the runtime layer, kernel optimization and hardware acceleration target higher throughput, faster primitive implementation at target devices, and tailored hardware design.

Table 1 contrasts the role of IDS alongside other optimization directions to achieve efficiency and scalability in modern AI inference. The role of IDS is especially important because modern

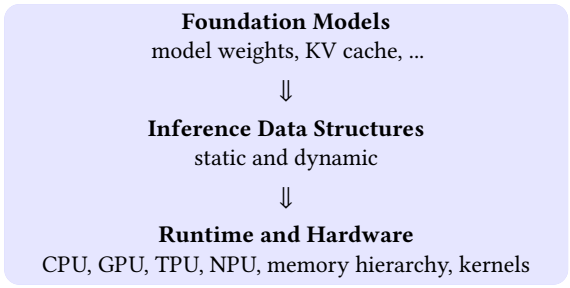


Figure 1: Inference data structures: an algorithmic infrastructure layer between foundation-model inference and the runtime.

inference is constrained by more than arithmetic throughput. Memory movement, cache capacity, bandwidth, latency, energy, device heterogeneity, and correctness under approximation all influence whether a model can be deployed effectively. These constraints are most visible in interactive and resource-limited settings, where users expect fast, private, affordable, and reliable inference without relying entirely on large remote clusters. They also matter in enterprise-scale deployments, where even modest improvements in repeated inference operations can translate into substantial system-wide gains.

Paper organization. In the rest of the paper, we first introduce the design space of static and dynamic inference data structures in Section 2. Next in Section 3, we illustrate some emerging examples, including structures for low-bit weight multiplication and KV-cache retrieval. Section 4 presents a research vision for inference data structures.

2 The Design Space of Inference Data Structures

An inference data structure (IDS) can be characterized by

- the inference variable it organizes,
- the procedure used to construct it,
- the updates it supports,
- the operation it accelerates, and
- the guarantees it provides.

This mirrors the classical view of databases with construction, update, and query procedures. The difference, however, is that IDSs organize reusable inference state, rather than actual data records.

The two most visible IDS classes in today's foundation models correspond to two persistent inference variables that strongly shape foundation-model inference: model weights and the KV cache. Model weights are fixed after training and reused across inference calls. KV caches are generated during autoregressive decoding and evolve as new tokens are produced. This leads to two basic classes: *static IDSs*, which organize immutable model state, and *dynamic IDSs*, which organize evolving runtime state.

A static IDS is constructed once from a variable that remains fixed during the lifetime of a model. The canonical example is a structure over model weights. Since the weights do not change after training, the system can afford an offline preprocessing step that reorganizes them into a representation better suited for repeated inference-time operations. The resulting structure may support

Table 2: Static and dynamic inference data structures. Static IDSs organize immutable model state, while dynamic IDSs organize inference state that evolves during execution.

Class	Organized state	Construction	Operation
Static IDS	Model weights	Built offline; no runtime updates	Repeated exact or approximate operations
Dynamic IDS	KV cache	Built and updated during decoding	Retrieval, pruning, or aggregation over cached state

faster matrix-vector multiplication, lower memory footprint, and improved locality, by reusing the repeated patterns. Note that an static IDS is constructed once, after the model is trained; hence, its construction cost does not add extra overhead to inference.

A dynamic IDS is constructed and maintained over state that changes during inference. The canonical example is the KV cache. During decoding, each generated token appends new key and value vectors. Future tokens repeatedly query this growing cache to identify the prior context needed for attention. A dynamic IDS therefore requires both update and query procedures: it must incorporate new cache entries while supporting efficient retrieval, pruning, or aggregation over prior entries. Unlike static IDSs, dynamic IDSs must operate under strict latency constraints because maintenance occurs during the inference, on the critical path of generation.

The distinction between static and dynamic IDSs results in different algorithmic challenges and objectives. Specifically, some of the main algorithmic questions are:

- *Static IDS*: how much space the preprocessed structure requires, what reusable structure can be extracted from fixed model parameters, what access pattern the structure induces on the target hardware, and whether the supported operation is exact or approximate.
- *Dynamic IDS*: how to maintain the structure incrementally, how to bound update overhead, how to adapt to query-dependent behavior, and what correctness or recall guarantees are required.

Guarantees are important for both classes. Some IDSs preserve the exact output of the original operation, while others provide recall guarantees or bounded approximation error. An IDS should also make its construction cost, update cost, query cost, memory footprint, and correctness behavior explicit.

We next discuss two representative examples based on (a) model weights and (b) KV caches. These are perhaps the most visible reusable inference variables in current foundation-model inference, but they are unlikely to be the last. As AI systems incorporate routing, retrieval, multimodal context, and longer-lived interaction state, other persistent inference variables may emerge. An open problem is to identify such variables early and determine how data structures can effectively exploit their reuse.

3 Emerging Examples

In this section, we discuss two representative examples of inference data structures: (a) static IDSs over low-bit model weights and (b) dynamic IDSs over the KV cache. Table 3 summarizes the correspondence.

Table 3: Two emerging instances of inference data structures.

IDS	Variable	Operation	Guarantee
RSR [6]	Low-bit weight matrices	vector-matrix multiplication	Exact output; less time and memory
Louver [5]	KV-cache	Relevant-token retrieval	Full recall above a threshold

A. Static IDSs over model weights. Modern foundation models contain large weight matrices that remain fixed across inference calls. This creates an opportunity to move work from online execution to offline preprocessing. In low-bit models, repeated column or row patterns can be exposed by reorganizing the matrix. Redundant Segment Reduction (RSR) [6] follows this principle: it preprocesses a fixed low-bit weight matrix into auxiliary structures, such as permutations, group boundaries, and segment metadata, so that repeated patterns are aggregated once and reused during vector-matrix multiplication. Using these auxiliary structures, RSR computes the vector-matrix multiplication exactly, while reducing the time and space complexity by a logarithmic factor.

RSR is an example of a static IDS. The indexed variable is the model weights, which remain fixed after training. The IDS is therefore constructed once offline, without adding construction overhead to inference. During inference, the same structure is reused for repeated vector-matrix multiplications. RSR changes how this operation is executed, but not its mathematical result. It therefore provides exact vector-matrix multiplication with lower time and space complexity.

RSR-core [4] adds system-level optimizations to RSR, including optimizations for CPU and CUDA kernels, compact metadata, and preprocessing artifacts. Recent follow-up work further indicates that structured representations for low-bit matrix multiplication are becoming an active direction, including addition-based sparse GEMM for edge LLM inference [26], sparse segment reduction for ternary GEMM acceleration [18], and multiplication-free ternary matrix multiplication for Transformers [17].

B. Dynamic IDSs over KV caches. The KV cache represents a different class of reusable inference state. Unlike model weights, it is not fixed before inference begins. The KV cache is an append-only tensor, which is created and extended during autoregressive decoding, as each generated token contributes new key and value vectors. In order to generate each token, traditional inference techniques conduct inner product operations over all entries of KV cache. However, it is evident that only a small number of those entries (called active tokens) may be relevant, in the sense that they contribute significantly to the attention output for the current token. Therefore, sparse attention methods reduce the inference cost by "guessing" the active tokens and only computing the inner products of those [2, 10, 11, 23, 24]. Sparse attention techniques can significantly drop the inference runtime but may miss relevant tokens.

Louver provides an example of a dynamic IDS for this setting [5]. It formulates sparse attention as a threshold retrieval problem over cached keys: for a query q_t and threshold τ, the goal is to retrieve

all keys k satisfying $\langle q_t, k\rangle \geq \tau$. This formulation transforms KV-cache access into a halfspace range-searching problem. The IDS organizes cached keys into groups with geometric summaries, uses inexpensive tests to prune groups that cannot contain relevant keys, and performs exact checks only on surviving candidates. New keys are buffered and incorporated into the structure during decoding, so the IDS supports both query and update operations.

Louver is an example of a dynamic IDS. Here, the indexed variable is the KV cache, which grows during decoding and must be maintained online. The query also changes from one decoding step to another, since the number and location of relevant keys depend on the token, layer, context, and task. Louver uses the IDS to avoid computing the inner product with every cached key. It supports sparse attention while guaranteeing that all active tokens with inner product above the threshold τ are detected.

4 Research Vision

After discussing the two illustrative examples, we now consider the broader role of inference data structures. Our view is that future inference systems should include IDSs as a layer between foundation models and runtime implementation (Figure 1), similar to the role of indexing in database systems. Today, efficient inference is mainly achieved by compressing models, optimizing kernels, or improving hardware utilization. These directions remain essential, but they do not directly take advantage of reusable model state. An IDS takes a different approach. It identifies which inference variables persist across computation, how those variables can be reused, what operations are performed over them, and what guarantees are needed for those operations.

Discovering reusable inference variables. The first question is which variables in the inference life cycle can benefit from a data structure. Model weights and KV caches are the most visible examples. In general, constructing and maintaining an IDS is useful only when the corresponding variable has sufficient persistence, reuse, structure, and importance in inference operations. As AI systems incorporate routing, retrieval, multimodal context, and longer-lived interaction state, other persistent inference variables are likely to appear. An important research problem is therefore to identify such variables and understand how they are reused during inference.

Designing principled IDS interfaces. An inference data structure is defined by the operations that it supports. As discussed in Section 2, this includes its construction, update, and query procedures, together with the guarantees provided by these operations. Depending on the application, such guarantees may include exact output preservation, zero false negatives, or bounded approximation error. These guarantees are particularly important for inference because an optimization may change the model output without making this change visible at the system level. Therefore, in addition to defining guarantees for individual IDS operations, we need to understand how these guarantees affect the quality of the model output.

Co-designing IDSs with runtimes. The algorithmic benefit of an IDS does not necessarily translate into an efficient implementation. For example, a structure may reduce the number of arithmetic operations but introduce irregular memory accesses, metadata movement, or synchronization overhead. In such cases, the resulting system may not be faster. The issue also depends on the target device. Random memory accesses that are tolerable on a CPU may be expensive on a GPU, while a layout designed for GPU execution may add unnecessary overhead on smaller devices. As a result, the design of an IDS should also consider its access pattern and the specifications and limitations of the target hardware.

Making IDSs first-class system optimizations. For IDSs to become part of the inference infrastructure, the interfaces between the model, the IDS, and the runtime need to be made explicit. The model needs to expose which inference state is persistent and reusable. The IDS needs to specify its construction and update costs, query operation, memory footprint, and correctness or approximation guarantees. The runtime can then decide when the structure should be constructed, updated, reused, moved across memory tiers, or discarded.

Such interfaces also allow an IDS to be used across different models and devices, rather than being implemented as a one-off optimization inside a particular kernel or inference system. Its evaluation should therefore not be limited to isolated microbenchmarks. In addition to query and update latency, the evaluation should consider storage overhead, token-generation throughput, memory footprint, energy usage, and preservation of model quality. Ultimately, these measurements should determine when constructing an IDS is beneficial, how it should be maintained, and when the runtime should use it.

5 Conclusion

Modern AI inference is commonly optimized through model compression and quantization, runtime kernels, and hardware design. In this paper, we argue that inference data structures provide another algorithmic layer for improving inference efficiency. They organize persistent inference variables, such as model weights and KV caches, into reusable structures that support explicit operations and provide correctness or approximation guarantees.

Our view is that future AI runtimes should treat IDSs similarly to how database systems treat indexes. Instead of repeatedly operating over raw model state, the runtime can construct, maintain, and query structures built over reusable inference variables. This requires mechanisms for deciding when such structures should be constructed, updated, reused, or discarded. As a result, improving inference efficiency is not limited to better models, kernels, and hardware. It also requires data structures that organize persistent inference state and reduce the cost of repeated computation.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Award No. 2348919 and CloudBank computation (NSF ACCESS allocation CIS251215).

References

[1] Elisa Bertino, Beng Chin Ooi, Ron Sacks-Davis, Kian-Lee Tan, Justin Zobel, Boris Shidlovsky, and Daniele Andronico. 2012. *Indexing techniques for advanced database systems*. Vol. 8. Springer Science & Business Media.

[2] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 16344–16359.

[4] Mohsen Dehghankar and Abolfazl Asudeh. 2026. RSR-core: A High-Performance Engine for Low-Bit Matrix-Vector Multiplication. *Proceedings of the VLDB Endowment* 19, 12 (2026), 4726 – 4729.

[5] Mohsen Dehghankar and Abolfazl Asudeh. 2026. Sparse Attention as a Range Searching Problem: Towards an Inference-Efficient Index for KV Cache. *CoRR* abs/2605.06763 (2026).

[6] Mohsen Dehghankar, Mahdi Erfanian, and Abolfazl Asudeh. 2025. An Efficient Matrix Multiplication Algorithm for Accelerating Inference in Binary and Ternary Neural Networks. In *International Conference on Machine Learning*. PMLR, 12969–12986.

[7] Norman P. Jouppi, David Patterson, et al. 2021. A Domain-Specific Supercomputer for Training Deep Neural Networks. In *Communications of the ACM*, Vol. 64. 67–78.

[8] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. doi:10.1145/3079856.3080246

[9] Sunil Kim et al. 2023. TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference. *NVIDIA Technical Report* (2023).

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Chenyu Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. ACM. doi:10.1145/3600006.3613165

[11] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Neural Information Processing Systems* 37 (2024), 22947–22970. doi:10.48550/arXiv.2404.14469

[12] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits. *CoRR* abs/2402.17764 (2024).

[13] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis Tsotras. 2012. *Advanced database indexing*. Vol. 17. Springer Science & Business Media.

[14] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 683–712.

[15] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. 2025. Towards efficient generative large language model serving: A survey from algorithms to systems. *Comput. Surveys* 58, 1 (2025), 1–37.

[16] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. *NVIDIA White Paper* (2020).

[17] Yushi Ogiwara and Hideyuki Kawashima. 2026. SpTMM: Multiplying Ternary Matrix Without Multiplication for Transformers. *Journal of Information Processing* 34 (2026), 525–534. doi:10.2197/ipsjjip.34.525

[18] Adeline Pittet, Shien Zhu, Valerie Verdan, and Gustavo Alonso. 2026. SSR: Sparse Segment Reduction for Ternary GEMM Acceleration. In *Design, Automation and Test in Europe Conference (DATE)*.

[19] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC58863.2023.10363447

[20] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 1–27.

[21] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. Bitnet: Scaling 1-bit transformers for large language models. *CoRR* abs/2310.11453 (2023).

[22] Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. 2024. Model Compression and Efficient Inference for Large Language Models: A Survey. *CoRR* abs/2402.09748 (2024). doi:10.48550/arXiv.2402.09748

[23] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *International Conference on Learning Representations* (2023). doi:10.48550/arXiv.2309.17453

[24] Zhenyu (Allen) Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Neural Information Processing Systems* 36 (2023), 34661–34710. doi:10.48550/arXiv.2306.14048

[25] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294* (2024).

[26] Shien Zhu, Guanshujie Fu, Mila Kjoseva, and Gustavo Alonso. 2026. Efficient Addition-Based Sparse GEMM for Fast Ternary Large Language Model Inference on Edge Devices. *ACM Transactions on Embedded Computing Systems* (2026). doi:10.1145/3807782