

Enhanced CXL Pooled Memory System for Scalable AI via Embedding Access Prediction

Jongho Park¹, Hoyeon Lee¹, Seohyun Kim¹, Minho Ha², Byungil Koh², Jungmin Choi², and Yeseong Kim¹

¹DGIST and ²SK hynix Inc.

{psyractal1123, lhyzone, selenium, yeseongkim}@dgist.ac.kr {minho1.ha, byungil.koh, jungmin.choi}@sk.com

Abstract—The embedding operation, pivotal in modern AI applications such as recommendation systems and natural language processing, transforms high-dimensional sparse data into dense vector representations. However, embedding tables are memory-intensive and pose significant challenges in DRAM-based architectures due to their substantial size. This paper introduces Sage, a scalable architecture for embedding operations in CXL-based pooled memory systems. Sage employs advanced caching and prefetching strategies, leveraging an online clustering algorithm to predict embedding table access patterns, and selectively uses Near-Data Processing (NDP) to mitigate the latency associated with CXL memory access. Our comprehensive evaluation demonstrates that Sage significantly enhances throughput and efficiency, providing a cost-effective solution for large-scale AI models. Our experimental results demonstrate that Sage enhances throughput by $2.84 \times$ as compared to conventional memory management systems.

Index Terms—CXL, Embedding, Near Data Processing

I. INTRODUCTION

Embedding layers transform sparse identifiers into dense vectors and are indispensable in modern AI applications such as the Deep Learning Recommendation Model (DLRM) [30] and Neural Collaborative Filtering (NCF) [15], as well as in natural language processing (NLP) models like BERT and GPT [9]. Their tables, however, span billions of rows and easily exceed host DRAM capacity, making memory the dominant bottleneck rather than compute. Compute Express Link (CXL) pooled memory is attractive because it exposes a large, shareable address space to many hosts and avoids costly table duplication [1], [5], [26], [29], [32], [34]. It introduces an extra few hundred nanoseconds per access over on-board DRAM, which can materially slow end-to-end inference if unmanaged [11], [12], [28].

A growing body of work shows that bringing simple tensor operations closer to memory can lift the memory roofline for sparse embedding pipelines and substantially reduce data movement, particularly for the gather-reduce phase that dominates recommendation inference. Still, directly transplanting near-data processing (NDP) ideas into a multi-host CXL setting is non-trivial: device-side processors must be provisioned for concurrent hosts, and model diversity makes a one-size-fits-all offload policy impractical. These realities motivate a hybrid approach that (a) predicts what to keep close to the host and when, and (b) uses CXL-side compute only where it closes the remaining latency gap.

One widely explored method to mitigate memory latency is to utilize the underlying locality of the memory accesses [19], [23]. In our analysis (refer to Section III, the access pattern in the embedding tables indeed has such localities, however,

frequency- or short-range recency alone is insufficient to overcome the long CXL latencies. In addition, cache design must be made batch-aware since performance turns on whether a *batch* is fully satisfied from local DRAM and a single miss within the batch forces CXL access for the entire step. In particular, we observed that two effects collide as batch size grows: the pipeline becomes less sensitive to single-access latency as bandwidth dominates, but naive caches (e.g., LRU) suffer sharp drops in *batch-level* hit rate because a larger batch is more likely to include at least one cold identifier. These trends surface again under multi-host contention, where table working sets overlap imperfectly across hosts.

In light of these observations, we frame the problem as three technical challenges that guide our design: (i) *Predictive, context-aware locality*. Beyond per-entry frequency or single-query recency, a CXL solution must capture longer-horizon and cross-table associations, so that host DRAM holds the *right* subset to maximize batch-level hits and hide device-latency outliers (ii) *Batch- and host-aware caching and prefetching*. Policies should reason at the granularity of a batch and across hosts, what to prefetch to the host between steps, what to retain, and when to evict, under constrained DRAM and shared-fabric bandwidth, not just per-access recency. (iii) *Selective, low-overhead NDP*. CXL-side compute should be used *only* for the portions that remain unpredictable or cold, and for operations that amortize transfers (e.g., look-ahead feature interaction), so that device compute scales with host concurrency without demanding heavy accelerators.

We propose Sage (Scalable Architecture for General-purpose Embedding in CXL environments), a hybrid memory management framework for efficient embedding inference over CXL pooled memory. Sage learns *contextual locality* online on the device by clustering co-occurring indices across tables and hosts. These clusters become the unit of management, i.e., each host maintains a host-reserved buffer (HRB) and caches, prefetches, and evicts *entire clusters* with a group LRU policy. Before each batch, the device-side model predicts a ranked set of clusters likely to be needed next; the host stages them under byte and bandwidth budgets to raise the probability of *batch-level* hits and to reduce exposure to device latency. For the residual cold slice, Sage issues lookahead near memory execution of the sparse-only interaction and overlaps it with host dense and cross computations. A best effort scheduler admits device tasks only when resources are available, enabling multi-host scalability without heavy device provisioning.

Our key contributions are summarized as follows:

- We analyze locality in production-style embedding accesses and identify *contextual locality* that spans users,

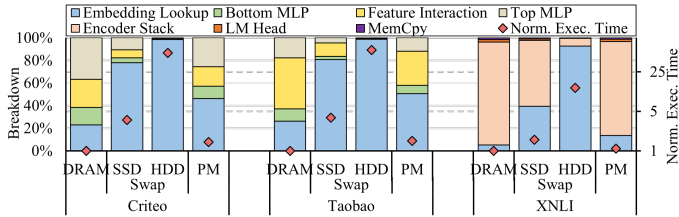


Fig. 1. DLRM Execution breakdown for different memory hierarchies

sessions, and tables, motivating management beyond single-entry recency.

- We design an online, device-resident clustering model that learns from sliding-window co-occurrence, producing a stable cluster partition and a ranked next-batch candidate set with low maintenance cost.
- We develop a runtime that operates at cluster granularity: a group-LRU HRB cache and prediction-guided prefetch that stage clusters under explicit budgets to maximize batch completeness.
- We introduce near memory execution that computes the sparse-only interaction for cold clusters and pipelines it with host MLP work, improving effective bandwidth use without overprovisioning device processors.

In our evaluation, we demonstrate that Sage provides substantial improvements in throughput and efficiency. Specifically, we observe a $2.84 \times$ increase in throughput for DLRM compared to conventional memory management systems on NDP-enabled CXL pooled memory system.

II. RELATED WORK

CXL-Based Memory Expansion. Recent advances in the use of CXL devices have focused on use cases as memory expanders. A few studies [17], [33], [2], [11], [4] have explored integrating CXL memory expanders with PCIe storage to transform it into scalable working memory akin to the canonical cache. These configurations translate PCIe’s block semantics into memory-compatible byte semantics to address the latency and capacity challenges. However, these efforts do not specifically optimize for deep learning applications, which could be key users in the CXL domain.

NDP-Based Optimization. NDP has been explored as a means to bring computation closer to data to reduce latency for embedding optimizations. Several works have investigated NDP to accelerate various applications. [10], [6], [24], [3], [19]. For example, TensorDIMM [21] accelerates DLRM embeddings using a custom DIMM module enhanced with NDP cores. However, these methods are not tailored for CXL systems and face challenges in scalability and cost, as they require dedicated memory systems for each host.

Locality-Based Optimization. Prior work has also studied the locality of embedding operations, particularly for recommendation systems and DLRM. For example, a work in [22] propose caching the ‘next batch’ by leveraging the properties of training. MERCI [25] and SPACE [18] focus on exploiting the locality within the tensor reduction operations. SPACE identifies locality based on the frequency of individual entries and stores partial sums to optimize memory usage. MERCI, on the other hand, leverages locality within individual queries,

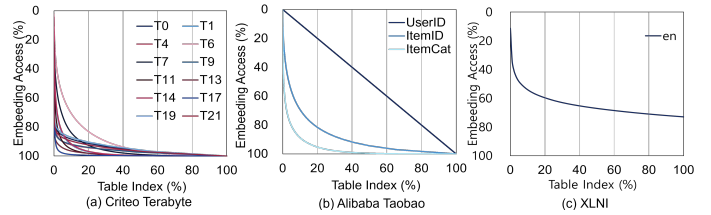


Fig. 2. Variation in Embedding Accesses for Different Datasets

optimizing memory usage by precomputing and reusing partial sums of tensor operations. Both techniques, however, confine their exploration of locality to the access frequency or within a single session of the user input query.

Our approach generalizes locality from short-term, per-interaction temporal effects to long-term, context-aware relations across sessions, users, and tables with a hybrid of selective caching, prefetching, and NDP that adapts online to locality heterogeneity to optimize memory access.

III. EMBEDDING USAGE CHARACTERIZATION

Embedding Overhead Analysis Embedding layers dominate the runtime of large-scale AI models [35], [36]. To see how memory configurations affect this overhead, we compare three cases: (i) *DRAM*, where the full table fits locally, representing an ideal but costly scenario; (ii) *Swap*, where part of the table is swapped to SSD/HDD; and (iii) *PM*, where the CXL-based pooled memory device provides extended memory spaces.

Figure 1 shows that fall-back quickly overwhelms performance making disk-backed solutions-SSD, HDD- impractical. *PM* mitigates the overhead of the embedding procedure without requiring memory dedicated to the host. However, embedding still accounts for more time than in *DRAM* case. This increase is due to the inherent latency in accessing remote DRAM over the CXL interface. These observations show that a more sophisticated memory management solution should leverage the benefits of CXL while mitigating its latency drawbacks.

Embedding Accesses Locality To identify embedding access locality, we analyze access frequency distributions in embedding tables. As shown in Figure 2, a small number of entries dominate the majority of accesses. For example, in Criteo, only 3% of entries account for over 80%. This indicates strong temporal locality and suggests that canonical management strategies, such as caching, could be effective in reducing memory access latency.

However, traditional caching techniques, though effective at the entry level, are inadequate for improving batch-level performance in embedding workloads. As shown in Figure 3, this leads to sharp latency increases despite low entry-level miss rates and rapid degradation of performance as the batch size grows. These results highlight the need for proactive strategies that identify potentially missed embedding entries in advance and place them in memory ahead of request.

Nonetheless, we can expect that it is plausible to recognize certain types of associations between different embedding entries due to usage semantics. To verify this relationship quantitatively, we first examine *in-query locality*-entries accessed within the same query- following the approach in [25]. As shown in Figure 4a, while certain entries are accessed together, the overall patterns are weak and insufficient for robust memory

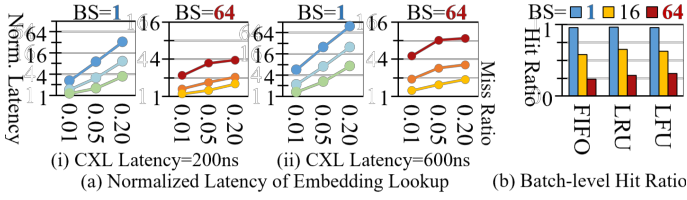


Fig. 3. Limitations of traditional caching for CXL pooled memory systems

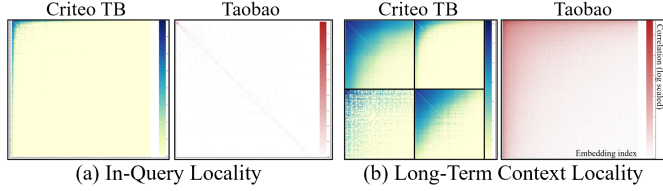


Fig. 4. Locality characteristic in embedding access

management. Therefore, we expand the view to consider a long-term horizon, introducing *context locality*. We examined which embedding entries co-occurred over multiple queries within a window to capture long-term, contextual associations. Figure 4b indicates that certain embedding entries are frequently accessed together, suggesting a potential for optimization through co-located data placement. Context locality highlights how embeddings, though not sequentially related, demonstrate significant associations that can be leveraged to align memory management with the relational patterns inherent in embedding data.

IV. Sage TECHNIQUES

A. Overview of Sage

Sage targets embedding workloads over CXL pooled memory and seeks DRAM-like behavior at the batch granularity. The core idea is to make *clusters learned online* the unit of management. Each host carves out a host-reserved buffer (HRB) in local DRAM and manages its contents at cluster granularity: on a reference in batch t to any member of a cluster g , the runtime admits the *entire* g to the HRB if not resident, packs its members, and updates the *cluster* recency. Clusters touched by the current batch are pinned until the batch completes to avoid intra-batch thrashing. When space is needed, eviction follows group LRU, removing whole clusters from the LRU tail rather than single entries.

A device-side predictor continuously learns *contextual locality* from recent batches by aggregating access streams across hosts. Before batch t , it produces a ranked set of clusters G_t^{pred} expected next; the host then admits top-ranked clusters subject to HRB capacity and a CXL bandwidth budget, skipping clusters that are already resident or pinned. Building the model on the device exposes cross-table and cross-host associations that a single host cannot observe.

During a batch, lookups first hit in the HRB; misses fetch from pooled memory. For the cold or weakly predictable slice, Sage issues a look-ahead request that computes the sparse-only feature interaction near memory while the host proceeds with the dense and cross terms. If device resources are busy, the host executes the full interaction without stalling the critical path. This division of labor overlaps device-side reductions with host

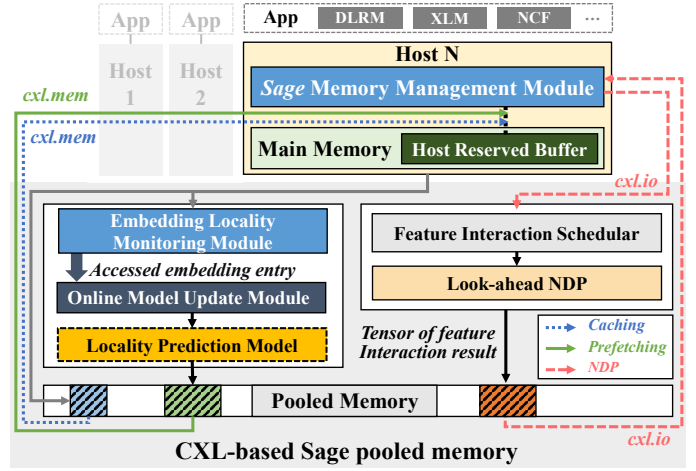


Fig. 5. Sage System Overview

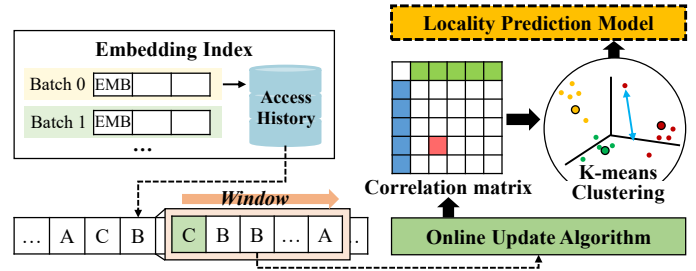


Fig. 6. Context Locality Detection via Clustering

MLP computation and pipelines work across batches, thereby tolerating CXL latency and bandwidth limits.

B. Online Context-Localty Model

The predictor learns *contextual locality* that spans users, sessions, and tables. Unlike array data, embedding indices have no spatial order, so recency at the single-entry level is weak. However, co-occurrence of indices across nearby queries and across hosts is stable enough to guide prefetch and cache admission if captured at the right granularity. The model runs on the CXL device to aggregate access streams from all hosts, which exposes correlations that a single host cannot observe and avoids duplicating state.

The procedure to build the clustering model is illustrated in Figure 6. For each table, the predictor maintains a sparse *co-occurrence affinity* matrix A over a sliding window of the most recent W accesses. Let $\text{freq}(i, j)$ be the number of times identifiers i and j appear together within the window. We define a log-normalized affinity

$$A(i, j) = \frac{\log(1 + \text{freq}(i, j))}{\log(1 + \text{freq}_{\max}) + 1} \in [0, 1],$$

where freq_{\max} is the current maximum pair frequency in the window. The logarithm reduces the effect of heavy-tailed popularity, while the normalization yields a comparable scale across tables. To bound space, for each identifier we retain only its top- K neighbors by $A(i, j)$ and discard weaker edges. The sliding window is updated incrementally at the end of every batch by inserting new accesses, removing expired ones, and

applying the corresponding pair increments and decrements; the amortized update cost is $O(W)$ and the retained graph has $O(KN)$ edges for N identifiers.

Clusters are formed online from the sparsified affinity graph. We represent each identifier i by a K -sparse vector of its retained affinities and apply mini-batch k -means with cosine distance. Only a small fraction of identifiers that changed neighborhood since the previous step are reconsidered, which keeps the per-batch work low. The number of clusters k is selected per table offline using standard Silhouette and Elbow criteria and kept fixed during evaluation; a periodic background sweep can adjust k if table scale changes.

The model produces two outputs that drive the runtime: the *cluster partition* $\{M(g)\}$ that defines the unit of management for the HRB, and a ranked *next-batch candidate set* G_t^{pred} . Ranking favors clusters that complete the likely working set of the next batch under the HRB budget. Concretely, given the multiset of identifiers observed in the recent window U_t , we compute a score for cluster g

$$s(g) = \sum_{i \in U_t} \max_{j \in M(g)} A(i, j) / |M(g)|^\alpha,$$

where $\alpha \in [0, 1]$ penalizes overly large clusters. The device emits clusters in decreasing $s(g)$ until a byte budget for staging is met. The host then admits these clusters into the HRB if not already resident, subject to capacity and bandwidth limits.

C. HRB Group-Cache and Contextual Prefetcher

Group Cache: The HRB in each host DRAM manages content at *cluster* granularity to align cache decisions with batch completeness. On a reference in batch t to any identifier that belongs to g , the runtime admits the *entire* g if it is not already resident, marks g as recently used, and serves subsequent accesses from the HRB. Clusters touched by the current batch are pinned until the batch completes to prevent intra-batch thrashing. Prefetched clusters produced by the predictor before t are treated identically once resident; demand and prefetch share one recency order.

Eviction follows group LRU. When space is required, the runtime removes whole clusters from the LRU tail until the requested cluster fits, skipping pinned clusters. Partial eviction is not allowed, so either all members of g are resident or none are. If g cannot be admitted after evicting all eligible unpinned clusters, the access bypasses the cache and is served from pooled memory without admission. This rule prevents repeated churn on large clusters and makes the miss cost predictable. In thwt way, the group-cache provides a simple contract to the rest of the system, enforcing coherent admission, recency, and eviction at the same granularity.

Contextual Prefetcher: The prefetcher stages *clusters* for the next batch so that the HRB serves most lookups without touching pooled memory. It consumes the device-side predictor’s ranked list G_t^{pred} and admits clusters to the HRB *as units*, matching the cache granularity. Prefetch runs in the window $[t-1 \rightarrow t]$ while the host executes batch $t-1$, which overlaps data movement with computation and hides CXL latency.

At the start of the window, the host receives the tuple $(g, s(g), \text{bytes}(g))$ for each candidate $g \in G_t^{\text{pred}}$. It then walks

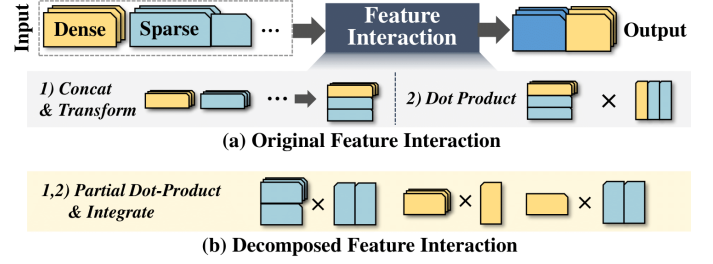


Fig. 7. Look-ahead NDP for Feature Interaction in DLRM

the list in decreasing score and, for each g , skips admission if the cluster is already resident or pinned. Otherwise, the runtime ensures space by evicting whole, unpinned clusters from the group-LRU tail until $\text{bytes}(g)$ fits. Prefetch and on-demand admissions share the same structures and recency order, which prevents duplicated state and avoids policy conflicts.

D. Host-CXL Co-execution and Scheduling

Sage overlaps host computation with optional near-memory execution to hide CXL latency without stalling the critical path. The interaction layer is decomposed so that the host computes $H_{\text{dense}} + H_{\text{cross}}$ while the device computes H_{sparse} only for *cold* clusters that are not resident in the HRB (Fig. 7). This split lets the runtime continue forward progress even when device is busy, since the host can always fall back to full execution.

Scheduling operates at cluster granularity and aligns with the cache and prefetch semantics in Sections IV-C. During batch t , hits are served from the HRB. On the first miss to a cluster g , the runtime admits g if space is available; otherwise the access is served from pooled memory. For clusters that remain cold after admission attempts, the runtime *may* issue a lookahead device task that computes $H_{\text{sparse}}(g, t)$ using embeddings in pooled memory. The task writes its reduced result to a per-batch buffer on the device, which the host later reads with CXL.mem and combines with $H_{\text{dense}} + H_{\text{cross}}$. If the device queue is saturated or the estimated benefit is low, the host immediately executes the sparse term to avoid delay.

The device scheduler, called look-ahead NDP, is best-effort to ensure multi-host scalability. Each host receives a small token budget that bounds outstanding near-memory tasks and prevents head-of-line blocking. Tokens are granted probabilistically in proportion to observed queue depth, which yields natural backoff under contention. The host adapts issuance with simple signals. If tasks for recent batches arrive late, the host reduces its token request; if they arrive early and are consumed, the host increases it up to a table-specific cap. This closed loop keeps the device busy when helpful and invisible when not.

Two forms of overlap improve efficiency. Within a batch, the host begins $H_{\text{dense}} + H_{\text{cross}}$ as soon as the needed dense features are ready and merges in H_{sparse} upon arrival or computes it locally if it is not ready by a short merge point. Across batches, prefetch for $t+1$ proceeds while batch t runs, and device tasks for cold clusters in t execute concurrently with the host MLP for $t-1$ and t (Fig. 8). This pipeline tolerates CXL bandwidth limits and device latency by overlapping data motion, near-memory reduction, and host compute.

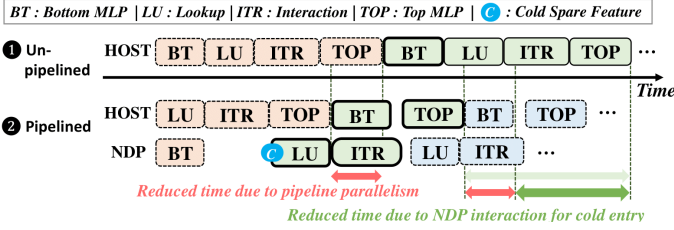


Fig. 8. Pipeline of Look-ahead NDP Operations across Batches

TABLE I
CXL-NDP SIMULATION CONFIGURATIONS

Host DRAM	DDR5, 4 channels, 6400 MT/s per pin (≈ 204.8 GB/s peak)
NDP DRAM	LPDDR5, 8 channels, 6400 MT/s per pin (≈ 102.4 GB/s peak)
Host Config	32 cores \times 2 threads, private 64 KB L1D+I
NDP Config	32 units \times 4 sub-cores, 128 KB scratchpad/sub-core
Frequency Bundle	Host Core/Cache 2.0 GHz; NDP Core/Cache 2.0 GHz; CXL Link 8.0 GHz
CXL mode	CXL 2.0 CXL.mem
Link	64 GB/s/dir (PCIe 5.0 $\times 16$) 200 ns CXL.mem Latency [400, 600]
Sage Host DRAM Cache	1 GB (CriteoTB), 256 MB (Taobao), 64 MB (XLNI)

V. EVALUATION

A. Experimental Setup

We developed our proposed Sage in C++ with LibTorch [31] as the NDP shared library, which is wrapped as a Python module using ctypes and integrated into PyTorch for use in our experiments. We employ the open-source *M2NDP* [14] simulator, which integrates Ramulator [20]-based DRAM timing and BookSim [16]-based interconnect models to provide cycle-level simulation of near-data processing systems. *M2NDP* does not natively support host-NDP co-execution or higher-level memory management policies. To enable evaluation of our proposed framework, we first execute target AI model inference on a local CPU system to obtain embedding access traces for the sequence of indices and their reuse patterns deterministically.

Based on these traces, our framework identifies contextual locality across embedding accesses and deterministically produces caching, prefetching, and offloading decisions, and we replay these mappings as annotated events in the simulator. Dense MLP operations remain on the host side and are modeled analytically using a roofline formulation. The detailed parameters, including host and NDP configurations as well as CXL link settings, are summarized in Table I.

For evaluation, we used three representative models: two DLRLMs [30] and one XLM [7], [27]. The first DLRLM uses the Criteo [8], and the second employs the Alibaba [13], both widely adopted for user behavior prediction in recommendation tasks. For NLP, we evaluated XLM on the XNLI EN [8], which targets cross-lingual classification. All experiments use a batch size of 64, unless otherwise noted.

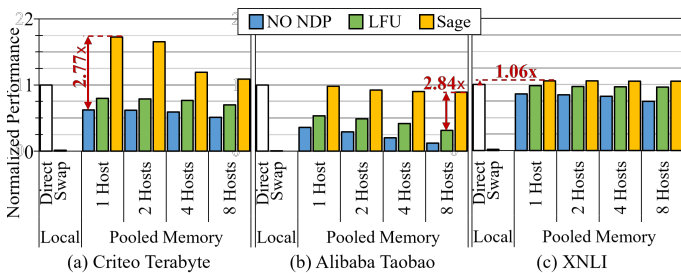


Fig. 9. Performance Improvement in Sage across Multi-Host Environments

B. Efficiency Evaluation

We first evaluate the performance of the proposed Sage system for end-to-end inference performance across various deep learning applications. To understand how Sage effectively mitigates CXL-induced latency and accelerates computation with the NDP functionality, we compare Sage with the following four architectures: (i) **Direct**: This scenario assumes a system equipped with sufficient resources to handle all model data and computations directly on the host, including embedding tables; (ii) **Swap**: The system has a main memory space which is insufficient to cover the embedding tables; **NO NDP**: The vanilla CXL pooled memory system without caching/prefetching as well as the NDP capability; and (iv) **LFU**: The pooled memory system utilizes the HRB memory to cache frequently used data based on observations from prior work [18] for comparison with existing literature.

Figure 9 presents the performance comparison across different architectures, normalized to that of **Direct** for each application. Compared to the **Direct** baseline, the CXL system shows inevitable slowdowns due to higher CXL latency and limited link bandwidth. In contrast, the proposed Sage demonstrates its effectiveness by enabling latency hiding through caching and prefetching mechanisms and further enhancing performance via Lookahead-NDP offloading, achieving up to $7.61\times$ ($1.72\times$) higher performance than the **NO NDP (Direct)**. With more hosts, the benefit relative to ideal DRAM decreases since NDP resources become overloaded, indicating that enlarging NDP capacity would further amplify Sage’s gains.

The evaluation also shows the significance of contextual locality prediction, which is integral to the proposed method. Compared to the **LFU** approach without considering long-term locality, Sage achieve notable performance improvement of $2.84\times$. This indicates that a simple frequency-based prediction, as utilized in previous work [18], is insufficient for hiding latency. Instead, capturing on contextual relationships between embedding entries contributes to performance enhancements.

NLP Tasks Our evaluation also show that the proposed Sage can also accommodate the transformer-based natural language processing applications. Since XNLI does not have a dedicated feature interaction, the results reflect only the effects of HBR caching/prefetching and NDP lookup. It shows that Sage is not only successfully hides the latency induced by the CXL but also achieves a $1.06\times$ performance improvement over **Direct** by pipelining the embedding lookup tasks across batches.

C. Context Locality Evaluation

To better understand where the improvement originates, we compare our proposed framework with conventional caching strategies (LRU/LFU) that only exploit temporal or frequency locality. Figure 9(a) illustrates the caching behavior on the Criteo dataset with varying HRB cache sizes. Although LRU and LFU achieve over 90% entry-wise hit rates even with an 8 GB cache, which is sufficiently large, the probability that an entire batch hits remains only 48.8% and 51.6%. In CXL-based systems, however, even a small number of misses within a batch trigger remote CXL accesses, severely degrading performance.

While Sage provides entry-wise hit rates comparable to conventional approaches, it delivers much higher batch-wise

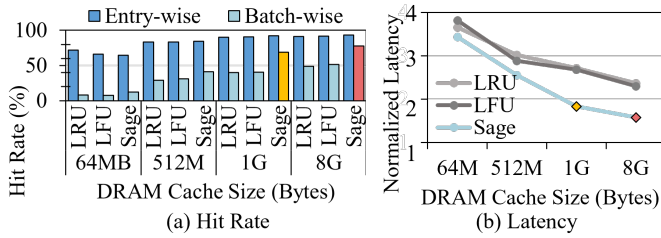


Fig. 10. Impact of Context Locality on DLRM

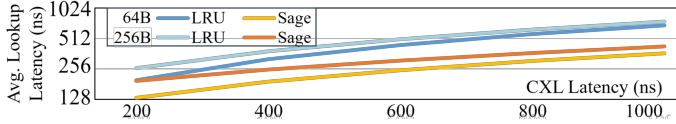


Fig. 11. Impact of CXL.mem Latency

hit rates through context-aware HRB caching and prefetching. For example, Sage achieves batch-level hit probability of 68.85%(77.87%) at 1 GB(8 GB) caches, which is 28.9%(26.3%) higher than LRU/LFU. Even at an extremely small cache size (64 MB, 0.1% of the Criteo table), Sage enhances batch-wise hits by 4.3% (1.53 \times) despite having lower entry-wise rates than LRU/LFU.

Figure 10(b) shows how these higher batch hit rates translate directly into an improvement in system-level performance. With LRU/LFU at 64 MB, batch hits rarely occur, and lookup throughput drops to 0.26 \times of the ideal DRAM case. In contrast, Sage sustains much higher batch hits, reducing CXL.mem overhead and improving lookup latency to 0.64 \times (1.51 \times) of ideal DRAM(LRU). These results demonstrate that traditional caching is inadequate for CXL pooled memory systems, and that proactive, context-aware caching/prefetching is a key enabler to optimize the throughput.

The effect of mitigating CXL memory overhead through contextual locality in Sage also scales well with increasing CXL latency. Figure 11 shows that embedding lookups are highly sensitive to CXL.mem latency. For example, with LRU-based caching, the average lookup time grows from 195ns at 200ns latency to 690ns at 1000ns latency. In contrast, Sage records 133ns and 426ns in the same environments, with sensitivity reduced to about 0.59 \times that of LRU. This suggest that Sage is more resilient to latency increases compared to the traditional caching methods.

D. Comparison with CXL Memory Expansion

Figure 12 compares Sage with a high-performance but costly baseline, CXL-D, where each host is equipped with an NDP-enabled CXL expander. Embedding tables are distributed across

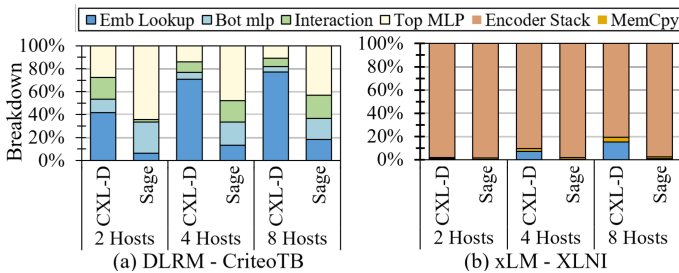


Fig. 12. Sage vs. NDP CXL Memory Expander

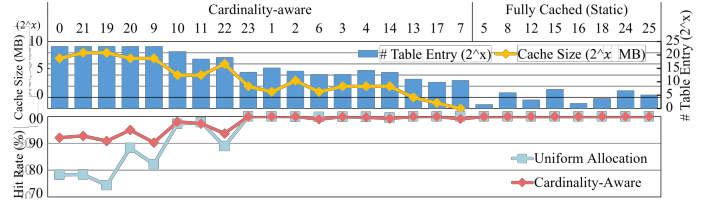


Fig. 13. Cardinality-aware HRB

hosts, with PCIe Gen5 x16 DMA transfers (1.5 us). Although CXL-D offers abundant resources, Lookahead-NDP cannot be effectively utilized because its tensor computations depend on prior lookup results. When those lookups incur DMA misses, the opportunity for hiding diminishes. In addition, the latency bottleneck of DMA communication significantly prolongs embedding lookup stalls. Consequently, with eight hosts, CXL-D shows 16.8 \times longer lookup times than Sage.

In contrast, Sage leverages context-aware management and Lookahead NDP to reduce remote accesses and schedule NDP more effectively, resulting in stronger scalability under increasing host counts. In NLP workloads, CXL-D also suffers from poor caching and DMA bottlenecks, whereas Sage hides up to 98.9% of lookup and CXL communication overhead by parallelizing with heavy host-side computation. These results show Sage 's robustness over conventional expander-based designs and its applicability to large embedding-centric models.

E. Impact of Per-Table Modeling Approach

Sage 's contextual prefetching operates at the granularity of embedding tables. Instead of allocating cache space uniformly by table size, Sage adopts a cardinality-aware policy that assigns larger cache regions to tables with more challenging access patterns, thereby improving overall caching efficiency. Figure 13 shows that providing more cache to such challenging tables significantly boosts their hit rates. Since the probability of a full-batch hit depends on all tables being served, cardinality-aware caching is essential to Sage 's design. This mechanism raises batch-level hit probability by about 10.8% compared to uniform allocation.

VI. CONCLUSION

The exploration and implementation of Sage have substantiated its effectiveness in addressing the complex memory management needs of modern AI systems. By integrating predictive modeling with advanced caching and prefetching strategies, Sage effectively mitigates bandwidth and latency challenges inherent in current CXL implementations. Its capability to anticipate and manage data locality across multiple levels translates theoretical improvements into tangible benefits, especially in environments where multiple computational clusters share CXL memory devices.

VII. ACKNOWLEDGEMENT

This work was supported in part by SK hynix Inc. This work was also supported by the NRF grant funded by the Korea government (MSIT) (RS-2025-24803164) and the Technology Innovation Program "Development of Navigation Technology Utilizing Visual Information Based on Vision-Language Models for Understanding Dynamic Environments in Non-Learned Spaces" (RS-2024-00445759).

REFERENCES

- [1] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebholz, V. Pham, K. Malladi, and Y. S. Ki, "Enabling cxl memory expansion for in-memory database management systems," in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022, pp. 1–5.
- [2] M. Arif, K. Assogba, M. M. Rafique, and S. Vazhkudai, "Exploiting cxl-based memory for distributed deep learning," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.
- [3] B. Asgari, R. Hadidi, J. Cao, S. K. Lim, and H. Kim, "Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, February 2021, pp. 908–920.
- [4] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill *et al.*, "Design tradeoffs in cxl-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.
- [5] D. Boles, D. Waddington, and D. A. Roberts, "Cxl-enabled enhanced memory functions," *IEEE Micro*, vol. 43, no. 2, pp. 58–65, 2023.
- [6] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, "Near data acceleration with concurrent host access," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 818–831.
- [7] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, "Unsupervised cross-lingual representation learning at scale," *CoRR*, vol. abs/1911.02116, 2019. [Online]. Available: <http://arxiv.org/abs/1911.02116>
- [8] Criteo, "Criteo dataset," <https://www.kaggle.com/c/criteo-click-through-rate-prediction/data>, 2014, accessed: 2024-08-03.
- [9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
- [10] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 113–124.
- [11] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory pooling with cxl," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023.
- [12] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with directcxl," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.
- [13] A. Group, "Alibaba click-through rate (ctr) prediction dataset," <https://tianchi.aliyun.com/dataset/dataDetail?dataId=42>, 2018, accessed: 2024-08-03.
- [14] H. Ham, J. Hong, G. Park, Y. Shin, O. Woo, W. Yang, J. Bae, E. Park, H. Sung, E. Lim *et al.*, "Low-overhead general-purpose near-data processing in cxl memory expanders," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 594–611.
- [15] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.
- [16] N. Jiang, D. U. Becker, G. Micheliogiannakis, J. Balfour, B. Towles, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 86–96.
- [17] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 45–51.
- [18] H. Kal, S. Lee, G. Ko, and W. W. Ro, "Space: locality-aware processing in heterogeneous memory for personalized recommendations," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 679–691.
- [19] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 790–803.
- [20] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2015.
- [21] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 740–753.
- [22] Y. Kwon and M. Rhu, "Training personalized recommendation systems from (gpu) scratch: Look forward not backwards," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 860–873.
- [23] D. Lee, J. So, M. Ahn, J.-G. Lee, J. Kim, J. Cho, R. Oliver, V. C. Thummala, R. s. JV, S. S. Upadhyaya *et al.*, "Improving in-memory database operations with acceleration dimm (axdimm)," in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022, pp. 1–9.
- [24] Y. Lee, J. Park, M. Ha, B. I. Koh, K. Park, and Y. Kim, "Sidekick: Near data processing for clustering enhanced by automatic memory disaggregation," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.
- [25] S. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "Merci: efficient embedding reduction on commodity hardware via sub-query memoization," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 302–313.
- [26] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.
- [27] D. Liang, H. Gonen, Y. Mao, R. Hou, N. Goyal, M. Ghazvininejad, L. Zettlemoyer, and M. Khabsa, "Xlm-v: Overcoming the vocabulary bottleneck in multilingual masked language models," *arXiv preprint arXiv:2301.10472*, 2023.
- [28] J. Liu, X. Wang, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li, "Exploring and evaluating real-world cxl: Use cases and system adoption," *arXiv preprint*, 2024.
- [29] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhat-tacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.
- [30] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.
- [31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [32] J. Sim, S. Ahn, T. Ahn, S. Lee, M. Rhee, J. Kim, K. Shin, D. Moon, E. Kim, and K. Park, "Computational cxl-memory solution for accelerating memory-intensive applications," *IEEE Computer Architecture Letters*, vol. 22, no. 1, pp. 5–8, 2022.
- [33] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.
- [34] J. Wahlgren, M. Gokhale, and I. B. Peng, "Evaluating emerging cxl-enabled memory pooling for hpc systems," in *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2022, pp. 11–20.
- [35] M. Xie, Y. Lu, J. Lin, Q. Wang, J. Gao, K. Ren, and J. Shu, "Fleche: An efficient gpu embedding cache for personalized recommendations," in *Proceedings of the Seventeenth European Conference on Computer Systems*, March 2022, pp. 402–416.
- [36] C. Zeng, X. Liao, X. Cheng, H. Tian, X. Wan, H. Wang, and K. Chen, "Accelerating neural recommendation training with embedding scheduling," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1141–1156. [Online]. Available: <https://www.usenix.org/conference/nsdi24/presentation/zeng>