CARINA: An Efficient CXL-Oriented Embedding Serving System for Recommendation Models

PEIQI YIN, The Chinese University of Hong Kong, Hong Kong SAR
QIHUI ZHOU, The Chinese University of Hong Kong, Hong Kong SAR
XIAO YAN*, Centre for Perceptual and Interactive Intelligence (CPII), Hong Kong SAR
CHAO WANG, The Chinese University of Hong Kong, Hong Kong SAR
ERIC LO, The Chinese University of Hong Kong, Hong Kong SAR
CHANGJI LI, The Chinese University of Hong Kong, Hong Kong SAR
LAN LU, University of Pennsylvania, USA
HUA FAN, Alibaba Cloud, China
WENCHAO ZHOU, Alibaba Cloud, China
MING-CHANG YANG, The Chinese University of Hong Kong, Hong Kong SAR
JAMES CHENG, The Chinese University of Hong Kong, Hong Kong SAR

Embedding-based recommendation models (ERMs) require large memory to host huge embedding tables and involve massive data traffic to read the embeddings. As a new interconnect, CXL suits ERMs since it can scale up single-machine memory with performant remote memory devices. However, directly running DRAM-based ERM serving systems on CXL yields poor performance because the bandwidth of CXL is notably lower than DRAM and can be easily saturated, making CXL memory the bottleneck. The non-uniform memory access (NUMA) architecture in modern CXL servers further decreased the system performance. In this paper, we design Carina for ERM serving on heterogeneous memory with CXL by considering such bandwidth asymmetry. In particular, Carina balances the memory access from different memory devices by storing hot embeddings with high access frequencies on DRAM and specifying the placement of embedding tables on the NUMA nodes. Moreover, Carina adopts bandwidth-aware task execution, which decomposes each batch of ERM requests into fine-grained tasks and schedules the tasks to control the real-time utilization of CXL bandwidth to avoid instantaneous saturation. We evaluate Carina under real CXL devices and find that it outperforms a CXL-oblivious baseline by an average of 5.38x and 4.04x in system throughput and request latency, respectively.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Hardware** → **Emerging technologies**.

Additional Key Words and Phrases: Recommendation Model, Model Serving, Compute eXpress Link

*Dr. Xiao Yan is the corresponding author.

Authors' Contact Information: Peiqi Yin, The Chinese University of Hong Kong, Hong Kong SAR, pqyin22@cse.cuhk.edu.hk; Qihui Zhou, The Chinese University of Hong Kong, Hong Kong SAR, qhzhou@cse.cuhk.edu.hk; Xiao Yan, Centre for Perceptual and Interactive Intelligence (CPII), Hong Kong SAR, yanxiaosunny@gmail.com; Chao Wang, The Chinese University of Hong Kong, Hong Kong SAR, cwang@cse.cuhk.edu.hk; Eric Lo, The Chinese University of Hong Kong, Hong Kong SAR, ericlo@cse.cuhk.edu.hk; Changji Li, The Chinese University of Hong Kong, Hong Kong SAR, cjli@cse.cuhk.edu.hk; Lan Lu, University of Pennsylvania, USA, lanlu@seas.upenn.edu; Hua Fan, Alibaba Cloud, China, guanming.fh@alibaba-inc.com; Wenchao Zhou, Alibaba Cloud, China, zwc231487@alibaba-inc.com; Ming-Chang Yang, The Chinese University of Hong Kong, Hong Kong SAR, mcyang@cse.cuhk.edu.hk; James Cheng, The Chinese University of Hong Kong, Hong Kong SAR, jcheng@cse.cuhk.edu.hk.

ACM 2836-6573/2025/6-ART137
https://doi.org/10.1145/3725274

ACM Reference Format:
Peiqi Yin, Qihui Zhou, Xiao Yan, Chao Wang, Eric Lo, Changji Li, Lan Lu, Hua Fan, Wenchao Zhou, Ming-Chang Yang, and James Cheng. 2025. CARINA: An Efficient CXL-Oriented Embedding Serving System for Recommendation Models. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 137 (June 2025), 29 pages. https://doi.org/10.1145/3725274

1 Introduction

Embedding-based recommendation models (ERMs) have good accuracy [17, 26, 28, 53] and hence are widely used in applications such as e-commerce [29, 68], advertisement [23, 90], and search [11, 52]. For instance, ERMs take up 80% of the AI inference compute cycles at Facebook [28]. An ERM consists of many embedding tables and a dense neural network. To serve the ERM for a batch of requests, the embeddings specified by the requests are first aggregated from the embedding tables (i.e., *lookup*) and then fed to the dense network to compute predictions (i.e., *compute*). *Lookup* requires large memory to store the embedding tables and usually runs on CPUs, while *compute* involves heavy model computation and typically runs on GPUs. Thus, *lookup* and *compute* are commonly deployed as separate services in production [35, 40, 56, 83, 89] to account for their distinct resource requirements. We focus on embedding serving (i.e., *lookup*) following existing research [47, 81, 82] as model computation is considered easy to handle.

The embedding tables can contain trillions of embeddings and take up TBs of memory [54]. Storing them in a server with large DRAM provides low lookup latency, which is crucial for quality of service (QoS). However, such high-end servers are expensive and limited by the DRAM capacity of a single machine. Some systems store the embeddings in SSDs to reduce cost [69, 72, 80] or over-distributed machines to improve scalability [25, 39, 57]. However, SSDs have much higher latency than DRAM and suffer from read amplification due to their 4KB block access granularity, while distributed solutions have high deployment costs and network latency.

CXL (Compute eXpress Link) [4] is a new interconnect technology that allows a server to access remote memory via PCIe and suits ERM embedding serving for two reasons: (i) The latency of CXL memory (e.g., 260ns) is much shorter than SSDs and network, and even comparable to DRAM (e.g., 110ns); and (ii) CXL has good scalability since a server can connect to up to 256 CXL devices (e.g., Micron 256GB CZ120 memory expansion module [7]) via a CXL switch [8] under CXL 2.0 protocol [3].

Since CXL memory is byte-addressable and can be accessed like DRAM, some systems directly interleave CXL memory with DRAM (i.e., using numactl [1]) as the underlying storage and reuse the execution engine for DRAM. Such a CXL-oblivious design yields DRAM-like performance for workloads including database OLTP [10, 73], key-value store [48, 75], and graph processing [66], but is inefficient for embedding serving due to *CXL congestion*. Specifically, embedding serving involves massive data traffic to read the embeddings but CXL bandwidth is notably lower than DRAM. For instance, a PCIe 5.0 x8 lane connects a server to one CXL memory device with 16GB/s bandwidth, while DRAM bandwidth can reach 250GB/s. As such, CXL bandwidth can be easily saturated by embedding access, making it a bottleneck for system performance. In addition, modern servers usually adopt the non-uniform memory access (NUMA) architecture, where the speeds of data transfer from CPU to CXL devices on different NUMA nodes are not the same. As a result, directly interleaving CXL memory devices across NUMA nodes amplifies CXL congestion.

More specifically, we show that two problems need to be tackled to resolve CXL congestion.

- *How to place the embeddings on heterogeneous memory*? Instead of simply interleaving all memory devices (i.e., DRAM and CXL memory across NUMA nodes) for storage, the placement should jointly consider the access pattern of the embeddings and the bandwidth of the memory devices. Ideally, the data traffic handled by each device should be proportional to its bandwidth such that

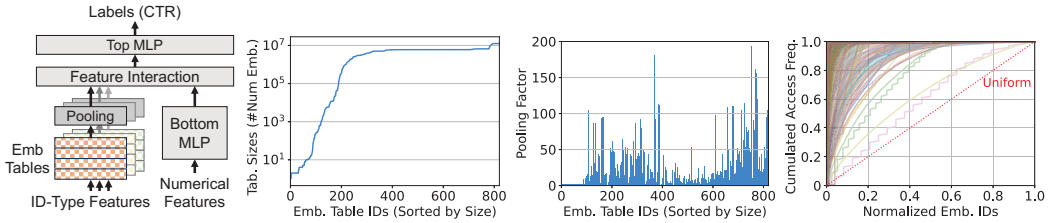
the loads of the devices are balanced. Otherwise, the overloaded devices will become stragglers and stall system performance. Moreover, the placement should be adjusted when there are shifts in the access patterns of the embeddings.

- *How to schedule the CPU threads to the read embeddings*? A good placement ensures that the traffic loads of memory devices are balanced *statistically*, but a device could still become a *temporary* bottleneck when too much traffic is requested on it in a short time. Moreover, the workloads of the threads may be skewed, making heavily loaded threads stragglers that stall requests. Different from static embedding placement, task scheduling should consider dynamic real-time memory access to avoid bandwidth saturation and balance the workloads of CPU threads.

To address the two problems, we propose Carina, a CXL-oriented ERM embedding serving system. Carina features two key designs for heterogeneous memory systems with CXL, namely *balanced embedding placement* and *bandwidth-aware task execution*. Since CXL servers adopt the NUMA architecture, we conduct embedding placement in two steps: (i) balancing the access across the NUMA nodes, and (ii) balancing DRAM and CXL access within the same NUMA node. In the first step, we assign the NUMA nodes to host different embedding tables (i.e., *table-wise sharding*) such that the workloads on one table can be processed locally by its resident NUMA node. To balance the memory consumption and lookup workloads of the NUMA nodes, we formulate a constraint satisfaction problem (CSP) to solve the table assignment. In the second step, we adopt *embedding-wise tiering* that stores hot embeddings in DRAM and cold embeddings in CXL memory to balance their traffic according to their bandwidths. When the access pattern changes, Carina migrates the embedding tables in the background for re-balance and minimizes the data movement overhead in this process.

To execute *lookup* for a batch of requests, Carina apply an *asynchronous task-based parallelism* strategy. In particular, a task reads embeddings from one table for a group of requests in the batch (called a micro-batch); each thread runs a task, and once the current task finishes, the thread proceeds to another task without waiting for the other threads. To avoid *instantaneous saturation* of CXL bandwidth, Carina scheduler monitors the embedding accesses of the running tasks and limits the real-time data traffic on CXL by properly selecting the tasks to execute. In comparison, DRAM-based embedding serving systems (e.g., TorchRec [30]) adopt *synchronous intra-table parallelism*, which uses all threads to read embeddings from one table each time. Our execution strategy avoids making the threads idle for synchronization and allows fine-grained control of the data traffic on the memory devices. We also identify and decompose the large tasks (i.e., targeting heavy embedding tables) into smaller ones such that they do not stall the requests. In particular, we formulate a mixed integer linear programming (MILP) problem to determine a task size for each embedding table and ensure that system throughput is not degraded. Note that the MILP is solved *offline* and thus does not add to the costs of online serving.

We evaluate Carina on a real CPU server with CXL 1.1 memory devices and compare it with a CXL-oblivious design that interleaves DRAM and CXL memory for storage and adopts TorchRec as the execution engine. The results show that Carina consistently outperforms the baseline by a large margin, e.g., increasing the request processing throughput by 5.38x on average, and reducing the average and 99th percentile request latency by an average of 4.04x and 4.08x, respectively. The results also suggest that the designs of Carina are effective. In particular, our balanced embedding placement avoids CXL congestion and fully utilizes the bandwidth of all memory devices. The task-based parallelism improves throughput by 3.03x, and the task scheduler improves throughput and reduces latency by 1.29x and 1.39x, respectively. The MILP-based task decomposition also reduces the average latency by up to 2.18x. Moreover, the results also show that Carina outperforms OS-level memory management systems such as TPP [60] and AutoNUMA [2].



(a) ERM architecture (b) Table size (c) Pooling factor (d) Access distribution

Fig. 1. The architecture of embedding-based recommendation model (a), and the embedding access patterns on the *FB856* [61] dataset (b-d).

To summarize, we make the following contributions.

- We observe that CXL memory suits ERM embedding serving and identify CXL congestion as the core problem that makes existing DRAM-based systems inefficient for CXL.
- To tackle CXL congestion, we propose two key designs, i.e., balanced embedding placement and bandwidth-aware task execution for heterogeneous memory platforms.
- We implement the designs in the Carina system, evaluate Carina's performance extensively on real CXL devices, and validate the effectiveness of our designs.

2 Background and Motivation

2.1 Embedding-based Recommendation Models

As shown in Figure 1(a), an embedding-based recommendation model (ERM) consists of dense neural networks and many embedding tables [17, 28]. Each embedding table corresponds to an ID-type feature and contains many vector embeddings. For instance, an embedding table may model the product categories (e.g., electronics and automobiles), and each embedding is trained to represent a category. A request for ERM serving comes with both ID-type features (e.g., the product categories a user visited) and numerical features (e.g., the price of the last purchase). To process the request, ERMs first lookup embeddings from the embedding tables based on the ID-type features (i.e., *lookup*). Multiple embeddings may be fetched from one table (e.g., a user visited multiple product categories) and pooled into one embedding (e.g., by `sum` or `max`). The number of embeddings pooled from a table t is called the pooling factor and is denoted as $pool_t$. For a model with T tables, *lookup* accesses $\sum_{t \in T} pool_t$ embeddings and produces T pooled embeddings. The pooled embeddings are then fed into a dense neural network (i.e., *compute*), where they are associated with the numerical features to compute the final prediction (e.g., click-through rate).

In practice, the embedding tables are large (e.g., in TBs) and dominate the model parameters of ERMs [26, 53]. Embedding lookup incurs large memory traffic as each request reads many high dimension embeddings. A disaggregated architecture is usually employed to serve ERMs [35, 40, 56, 89], which hosts *lookup* and *compute* on separate servers. Specifically, *lookup* runs on servers with large memory and are designed to handle memory-intensive embedding read, while *compute* runs on servers equipped with GPUs to process dense neural network computations. This paper focuses on the storage servers for embedding serving.

Access patterns. In Figure 1, we profile the embedding access patterns of ERM on a dataset open-sourced by Meta. Figure 1(b) shows 856 embedding tables and there are hundreds of tables that have millions of embeddings. Note that the table IDs (x-axis) are sorted by the table sizes (y-axis), i.e., the number of constituent embeddings. Figure 1(c) shows that the average pooling factor varies across

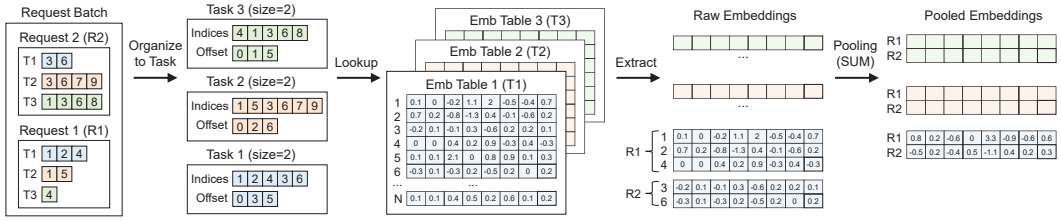


Fig. 2. A running example for embedding lookup for ERMs.

the embedding tables, ranging from 1 to 200. This suggests that the tables have different lookup workloads in the number of fetched embeddings. Figure 1(d) plots the cumulative distribution function (CDF) of the embedding accesses with each curve corresponding to an embedding table. The results show that most tables follow a *skewed distribution*, and a small portion of hot embeddings account for the majority of accesses. Notably, for some highly skewed tables, more than 99% of the accesses are conducted on less than 1% of the embeddings. As we will show later, the designs of Carina are tailored for these access patterns.

Embedding lookup. Figure 2 provides a running example for embedding lookup works for ERMs. Each embedding table (e.g., T_1) is a 2D array that consists of an embedding ID and an embedding entry in each row. The ID-type features take discrete values and specify the embedding IDs to access. For instance, an ID-type feature can record the product categories a user visited recently (e.g., sports, food, and electronics), and another ID-type feature may record the geographic location of the user (e.g., which country, and which state/province of the country). A request is a user query that specifies the embeddings to fetch from the embedding tables. For instance, in Figure 2, request $R_1 = \{[1, 2, 4],\ [1, 5],\ [4]]\}$ will read $T_1[1]$, $T_1[2]$, and $T_1[4]$ from table T_1, $T_2[1]$ and $T_2[5]$ from table T_2, and $T_3[4]$ from table T_3. After a request retrieves its embeddings, the embeddings from the same table are pooled (e.g., via SUM, MAX, or AVG) into a single embedding. In Figure 2, for request R_1 on embedding table T_1, the pooled embedding is $\mathsf{SUM}(T_1[1],\ T_1[2],\ T_1[4])$. For each request, embedding lookup returns the pooled embeddings, which are passed to the compute servers for subsequent model computation (i.e., feature interaction and MLPs).

A batch contains multiple requests, e.g., R_1 and R_2 for the batch in Figure 2. Batches are formed by an external system/service and submitted to embedding lookup. A task handles the embedding lookup of some requests on one embedding table, e.g., Task1 processes embedding table T_1 in Figure 2, and the size of a task is the number of requests it handles. In a task, offsets indicate the positions to write the embeddings fetched for the requests and are generated by Carina. For instance, Task1 reads embeddings $[1, 2, 4]$ from T_1 for R_1 and $[3, 6]$ from T_1 for R_2, and the embeddings for R_1 and R_2 start from the zeroth and 3rd output entries, respectively. Embedding lookup uses the embedding IDs to fetch the corresponding embedding entries, and a scan over the embedding table is not required. When the entries of an embedding table are partitioned over DRAM and CXL, Carina uses an index for embedding lookup.

2.2 CXL Basics

CXL (Compute eXpress Link) [4] is an emerging interconnect technology that leverages PCIe for efficient communication among CPUs, smartNICs, accelerators, and other devices. A server can conduct cache-coherent access to all devices connected to it via CXL. CXL classifies the devices into three types: smartNICs as Type-1, accelerators (e.g., GPUs and FPGAs) as Type-2, and memory expansion modules (e.g., DRAM and PMEM) as Type-3. Notably, CXL Type-3 devices, referred to

Table 1. Latency and bandwidth (BW) of READ access over different memory devices. The server connects to two CXL memories with 16GB/s bandwidth on each NUMA node.

	Local NUMA		Remote NUMA	
	DRAM	CXL-Mem	DRAM	CXL-Mem
Latency (ns)	114.5	260.5	191.9	456.6
BW (GB/s)	254.2	31.9	118.4	31.6

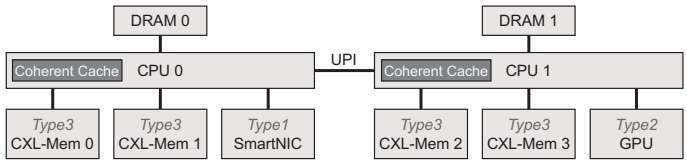


Fig. 3. CXL-enabled server with NUMA architecture.

as CXL memory subsequently, are production-ready and can scale CXL servers to terabytes of memory with good bandwidth and latency. Such large memory capacity benefits ERM embedding serving, which requires substantial space to keep the embedding tables.

Figure 3 shows the architecture of a CXL server and the attached devices, and Table 1 profiles the connections in the server. We observe that the latency of CXL memory is comparable to DRAM, and thus using CXL to expand server memory will not harm request latency. However, the bandwidth of CXL is significantly smaller than DRAM (i.e., 32GB/s vs. 256GB/s). This can make CXL a bottleneck for ERM embedding serving when CXL is used to store the embeddings and receives high data traffic. Moreover, CXL servers generally adopt the NUMA (Non-Uniform Memory Access) architecture, which contains multiple NUMA nodes with their local DRAM and CXL memory. Accesses to data on remote NUMA nodes go through the UPI (Ultra Path Interconnect) link and are thus significantly slower than local NUMA. For instance, the latency for reading remote NUMA CXL is 4x of local DRAM in Table 1.

Another type of emerging storage device, PMEM (Persistent Memory) [5], also has properties similar to CXL memory as shown in recent studies [13]. However, existing commercial PMEM devices (i.e., Intel's Optane DC Persistent Memory [5]) use the DDR-T protocol and share memory channels with DRAM, which results in bandwidth contention between DRAM and PMEM [77]. In contrast, the CXL protocol operates over PCIe, enabling the CXL memory devices to utilize separate bandwidth from DRAM.

2.3 Challenges

DRAM-based serving systems (e.g., TorchRec, TFRS [24]) can be deployed on CXL-enabled servers by simply interleaving DRAM and CXL memory to store the embedding tables. In particular, interleaving means that consecutive memory addresses are assigned to different memory devices such that the system uses multiple memory devices simultaneously. Although this CXL-oblivious method suits some applications (e.g., KV store and graph processing), it yields poor efficiency for ERM embedding serving. By profiling TorchRec on CXL memory, we identify two key challenges.

Heterogeneous memory devices. The memory devices of a CXL-enabled server are heterogeneous in performance and should be treated differently. However, the interleaving approach treats them as the same and assigns the embeddings evenly to them. We interleave DRAM and CXL

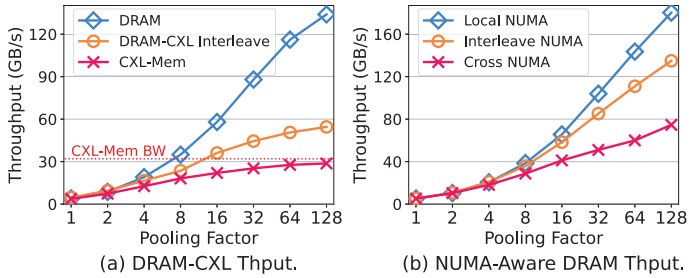


Fig. 4. Throughput of TorchRec on one embedding table that contains $4{\times}10^7$ embeddings. (a) Store embeddings with/without CXL memory. (b) Interleave embeddings among NUMA nodes and use threads from one node.

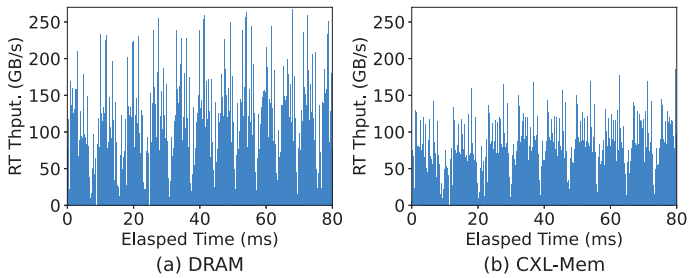


Fig. 5. Real-time throughput of TorchRec on the *FB856* dataset. The throughput can be higher than the device bandwidth because some embeddings are in CPU cache.

memory to store an embedding table and compare it with pure DRAM and CXL storage. Figure 4(a) shows that DRAM-CXL interleaving yields much lower throughput than DRAM when the pooling factor is large. This is because CXL bandwidth is saturated due to heavy data traffic when each request reads many embeddings. In this case, CXL memory becomes the bottleneck and stalls the system, evidenced by the fact that DRAM-CXL interleaving eventually achieves 2x the throughput of CXL since CXL stores half of the embeddings. Figure 4(b) shows that interleaving an embedding table across NUMA nodes performs worse than storing the table on one NUMA node and using threads from the local NUMA node to access the table. This is because NUMA-interleaving results in remote NUMA memory access, which is slower than local access.

Skewed workloads. Existing systems process the embedding tables sequentially and use all threads to lookup one table each time (i.e., intra-table parallelism). We plot the real-time throughput of TorchRec on DRAM and CXL memory in Figure 5, which shows that TorchRec's throughput fluctuates significantly. This is because the workloads on the embedding tables are skewed, which is caused by their varying pooling factors as shown in Figure 1(c). Specifically, the data traffic is high when processing embedding tables with large pooling factors, and CXL bandwidth may be saturated. Conversely, the data traffic is low when processing embedding tables with small pooling factors, and the bandwidth of the memory devices is not fully utilized. Intra-table parallelism does not allow fine-grained control of the data traffic on the memory devices, which is essential for handling the skewed workloads of the embedding tables.

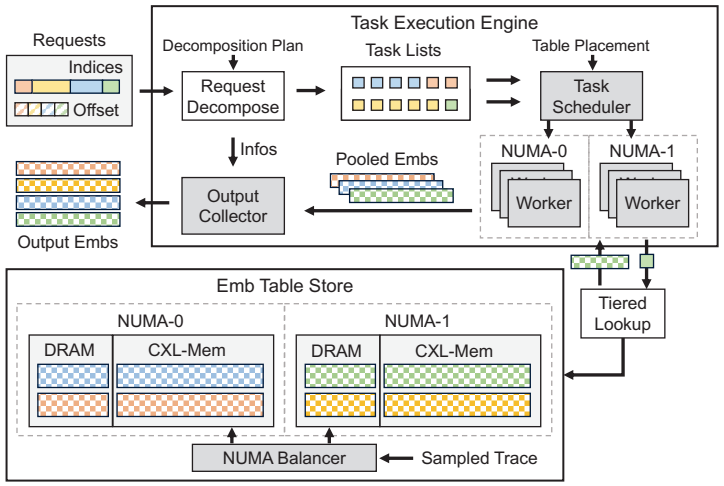


Fig. 6. The system architecture and workflow of Carina.

3 Carina System Overview

Carina receives ERM embedding serving requests from clients and processes the requests in batches. The batch size is specified by users (e.g., to balance between request throughput and latency) as in existing systems [35, 78, 81]. The requests specify the embeddings to read from the embedding tables and Carina returns the pooled embeddings for all requests in a batch. Carina runs on a CPU server with CXL memory and its architecture is shown in Figure 6. Carina consists of two main components, i.e., embedding table store and task execution engine.

- *Embedding table store* manages the placement of the embedding tables across heterogeneous memory devices. Upon system initialization, the store loads the embeddings from disk to CXL memory. Each embedding table is then assigned to one NUMA node by solving a constraint satisfaction problem (CSP) to balance both embedding access and space consumption across the NUMA nodes (§4.1). The store also employs a tiering policy (§4.2) to keep the frequently accessed embeddings on DRAM to reduce CXL data traffic. During serving, the NUMA balancer monitors the traffic of the NUMA nodes. If the traffic is unbalanced, embedding tables are migrated across NUMA nodes in the background for re-balance without stopping request serving (§4.3).
- *Task execution engine* processes the requests by accessing the embedding table store. The engine adopts an efficient asynchronous task-based parallelism strategy for execution (§5.1), which processes embedding lookups at the granularity of task (i.e., handling a group of requests in the batch on an embedding table) without synchronization. When receiving a request batch, the engine decomposes it into a set of tasks according to an MILP (i.e., mixed integer linear programming) generated task decomposition schema (§5.3) to avoid large tasks. The MILP is solved *offline* to determine a task size (i.e., number of requests in a task) for each table. The task scheduler also monitors the running tasks and chooses the tasks to run to avoid CXL saturation and maximize local NUMA access (§5.2). Once assigned a task, the worker lookups embeddings from the store and forwards the pooled embeddings to the output collector. When all pooled embeddings are collected for a batch, the engine returns the outputs to the client.

We consider a single server but our designs can be extended to a distributed environment. Since CXL 2.0 [3] allows multiple servers to access the same CXL device via a CXL switch, a CXL

shared memory pool can be built following existing works [12, 88]. To scale up request throughput, each server can provide service independently by keeping its own copy of the hot embeddings in DRAM and sharing one copy of full data in the CXL shared memory pool. This saves CXL memory compared with replicating data across the servers.

Although Carina targets ERM embedding serving, some of our designs, e.g., balanced data placement on DRAM and CXL memory, and task scheduling over heterogeneous memory devices, also apply to other applications. Ideally, the applications should (i) require large memory to keep many embeddings and (ii) conduct skewed access to the embeddings. Two applications meet these requirements, i.e., graph learning and vector similarity search. In particular, graph learning such as graph neural networks (GNNs) [15, 21, 76, 85], knowledge graph reasoning (KGR) [65, 92] and graph embedding (GE) [50, 63, 91] use the node/edge embeddings, which can be large for huge graphs. Moreover, accesses to these node/edge embeddings are usually skewed due to the graph structures [74]. Given a query vector, vector similarity search finds its most similar vectors from a large vector dataset [59]. It has been reported that some popular vectors are more likely to become the results of vector similarity search because the queries are biased [36, 62].

4 Balanced Embedding Placement

For a CXL server with heterogeneous memory (i.e., CXL memory and DRAM under multiple NUMA nodes), simply interleaving all memory devices may overload some devices and make them bottlenecks. To avoid such bottlenecks, we conduct embedding placement in two steps: (i) balancing accesses and memory consumption to the NUMA nodes via table-wise sharding in §4.1, and (ii) balancing accesses to DRAM and CXL memory within the same NUMA node via embedding-wise tiering in §4.2. Finally, We discuss how Carina adjusts its embedding placement when the access pattern changes in §4.3.

4.1 Table-wise Cross-NUMA Sharding

We assign each embedding table entirely to one NUMA node because by using the local threads of the NUMA node to process the embedding table, inefficient remote NUMA access is avoided (see Figure 4). Randomly assigning the tables may lead to unbalanced data access and space consumption across the NUMA nodes. Balanced data access is important because a batch of requests is processed in parallel on the NUMA nodes, and one NUMA node will stall the batch if it receives too many accesses. Balanced space consumption is also favorable because it avoids the case of a NUMA node hosting many large tables and going out of memory. Additionally, by assigning an equal number of embedding tables to each NUMA node, the workloads of the NUMA nodes are balanced. This is because Carina processes the lookup task on one embedding table on its resident NUMA node for good locality (see Section 5).

We consider three balancing objectives across the NUMA nodes, i.e., *storage size*, *data access*, and *number of tables*. To achieve these objectives, we formulate table assignment as a constraint satisfaction problem (CSP). In particular, we denote the set with T embedding tables as $\mathcal{T}$, the set containing N NUMA nodes as $\mathcal{N}$, the embedding tables assigned NUMA node $n \in \mathcal{N}$ as $\mathcal{T}_n$. We constrain that $\mathcal{T}_n$ contains T/N tables, and the goal is to solve $\mathcal{T}_n$ for each NUMA node n. The CSP is expressed as

$$|\mathcal{T}_n| = \frac{T}{N}, \quad \Big|\sum_{t\in\mathcal{T}_n} P(t) - \frac{T}{N}\sum_{t\in\mathcal{T}} P(t)\Big| \geq \phi \cdot \frac{T}{N}\sum_{t\in\mathcal{T}} P(t)$$
$$\forall n \in \mathcal{N}, \quad \forall P \in \{size,\ access\}, \tag{1}$$

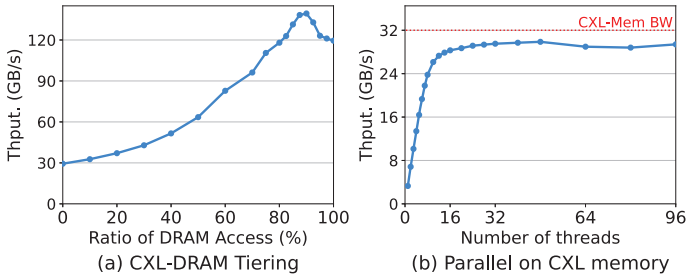


Fig. 7. (a) CXL-DRAM tiered lookup with different DRAM access ratios using the DRAM and CXL memory of a NUMA node. (b) Parallel lookup performance on CXL memory.

where P stands for both storage size (i.e., *size*) and data access (i.e., *access*). Take storage size *size* as an example. The term $\sum_{t \in \mathcal{T}_n} size(t)$ is the storage size on NUMA node n as it sums over the tables in $\mathcal{T}_n$; $\frac{T}{N} \sum_{t \in \mathcal{T}} size(t)$ is the average storage size over all NUMA nodes. ϕ is a tolerance factor that determines how far each NUMA node can deviate from the average and is set as 0.01 by default. The CSP can be solved quickly using the CBC solver [22], e.g., it takes less than 100ms for the FB856 dataset with more than 800 tables.

4.2 Embedding-wise DRAM-CXL Tiering

We consider the embedding tables assigned to a NUMA node and decide how to store their embeddings in the local DRAM and CXL memory of the NUMA node. The naive placement, i.e., interleaving CXL memory with DRAM and randomly placing the embedding, is observed to be inefficient in Figure 4(a). This is because DRAM and CXL memory receive the same data traffic but the bandwidth of CXL memory is notably smaller than DRAM. This makes threads contend for CXL bandwidth, and Figure 7(b) shows that using more threads to access CXL does not improve the throughput due to bandwidth saturation. Thus, we should reduce the data traffic on CXL memory to avoid making it the bottleneck.

Many researches observe that accesses to the embeddings are skewed [37, 47, 67] (see Figure 1(d)) and thus cache hot embeddings in fast memory devices. Similarly in our case, storing the hot embeddings in DRAM can reduce the data traffic on CXL memory. However, Figure 7(a) shows that conducting more accesses to DRAM does not always lead to higher throughput. This is because DRAM and CXL memory can be accessed in parallel, and thus the accesses to them should be proportional to their bandwidth to fully utilize both devices. Building upon this intuition, we conduct DRAM-CXL tiering by ensuring that the following equation holds.

$$\frac{\sum_{e \in \mathcal{E}_{DRAM}} \mathcal{A}^e}{\sum_{e \in \mathcal{E}_{CXL}} \mathcal{A}^e} = \frac{\mathcal{B}_{DRAM}}{\mathcal{B}_{CXL}}, \tag{2}$$

where $\mathcal{B}_{DRAM}$ and $\mathcal{B}_{CXL}$ are the bandwidth of DRAM and CXL memory, respectively; $\mathcal{E}_{DRAM}$ and $\mathcal{E}_{CXL}$ denote the sets of embeddings assigned to DRAM and CXL memory; and $\mathcal{A}^e$ is the access frequency of embedding e. To meet Equation (2), an embedding table is first loaded into CXL memory. The embeddings are ranked in descending order of their access frequencies and copied to the DRAM starting from the most popular ones. Once the left hand side of Equation (2) becomes larger than the right hand side, we stop and keep the remaining embeddings on CXL memory.

To adapt to dynamic changes in embedding access patterns, we use background threads to update the DRAM storage periodically according to the recent embedding access (i.e., recorded

from sampled traces). We apply this strategy independently to each embedding table, allowing for separate and concurrent updates to the DRAM storage for each table.

4.3 Runtime Placement Adjusting

The access pattern of the embeddings may change over time [67], and thus the initial table placement can become unbalanced between NUMA nodes and lead to inferior performance. To solve this problem, Carina introduces a *balancer* that tracks the access frequencies of tables and triggers placement adjusting when the current sharding plan is found to be unbalanced (i.e., by a threshold). A naive solution is to stop the system and generate a new sharding plan. To avoid halting request serving, we migrate embedding tables across the NUMA nodes for re-balance in the background and we minimize the data movement during migration.

Minimum data movement. We formulate an MILP (i.e., mixed integer linear programming) problem to minimize the data movement when adjusting the original sharding plan into a balanced sharding plan. In particular, NUMA node n holds table $\mathcal{T}_n$ in the current plan and tables $\mathcal{T}_n'$ in the new plan. The new sharding plan should satisfy the storage size and data access balancing constraints in Equation (1), and we set the following objective function

$$\mathcal{T}_c = \bigcup_{n \in \mathcal{N}} (\mathcal{T}_n' - \mathcal{T}_n), \quad \min \sum_{t \in \mathcal{T}_c} size(t), \tag{3}$$

where $\mathcal{T}_c$ are the migrated tables. Solving the MILP is also quick (e.g., taking 50ms). By applying the minimum target, the new placement can minimize the data size to be moved. Since the original plan already has a balanced storage size, the new plan will generate based on the old one while most of the tables will keep the original placement.

Moreover, some tables can be copied among all NUMA nodes to reduce data movement. For example, Figures 1(b-c) show that 25% of the tables in *FB856* dataset account for 22% of the total data accesses but take up less than 1% of the total size. We also observed a similar phenomenon on other datasets. By duplicating these tables across NUMA nodes (i.e., on CXL memory), the migration of them only needs to move the data in the DRAM part.

Consistency. During table migration, we ensure that new requests can locate the embedding tables and ongoing requests can finish by using a lazy release strategy [41] for the moved tables and optimistic concurrency control (OCC) [46] on the scheduler. In particular, we adopt an atomic location tag L_t to record the latest location of each table t (i.e., on which NUMA node), and the task scheduler assigns tasks to workers based on L_t. L_t is modified after table t is copied to a new NUMA node. We also use an atomic counter C_{L_t} to record the number of unfinished tasks on table t for each NUMA node. Table t on the original NUMA node are deleted after the copy finishes and its local C_{L_t} becomes 0. Such lazy table release strategy prevents workers from accessing freed memory spaces. Note that reading L_t and updating C_{L_t} are not atomic operations in the scheduler. Since conflicts rarely occur, we use OCC by letting the scheduler recheck L_t after updating C_{L_t}.

The lazy table release strategy may result in higher memory consumption due to unreleased tables. To avoid out-of-memory (OOM) errors, the migration process will move tables individually, checking if the target NUMA node has sufficient memory for each table. If not, the migration for the current table will be blocked until the unreleased tables are freed. Additionally, a portion of the memory space in each NUMA node is reserved to prevent deadlocks during the migration process.

Existing researches [37, 47, 67] has observed that embedding access is skewed and proposed to cache the hot embeddings in fast memory. Although we adopt a similar tiering strategy, our data placement tackles several problems specific to a CXL-based memory system, e.g., balancing the

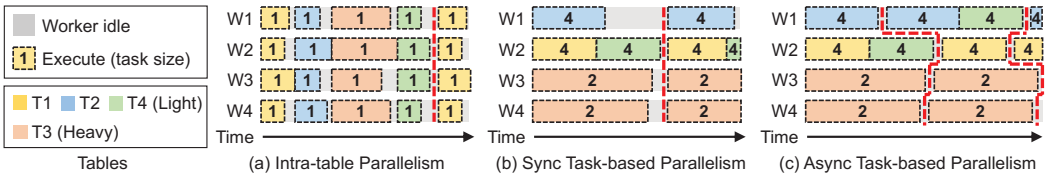


Fig. 8. Different parallelism strategies for request serving. There are 4 tables, the request batch size is 4, and the server contains 4 workers (i.e., W1-W4). The size of a task is the number of requests it processes, and at the same size, tasks on tables with larger pooling factors (e.g., T3) have long running times. The red dash line indicates the completion of the current batch.

workloads, storage, and access across the NUMA nodes, balancing the access-to-bandwidth ratios of DRAM and CXL, and dynamically adjusting the embedding placement to adapt to the changes in embedding access pattern. These designs distinguish Carina from existing researches.

5 Bandwidth-aware Task Execution

DRAM-based ERM embedding serving systems (e.g., TorchRec) use intra-table parallelism to process requests. As shown in Figure 8(a), all CPU threads are launched to perform embedding lookups on one table simultaneously and synchronize before proceeding to the next table. Intra-table parallelism is inefficient for heterogeneous memory platforms with CXL because it may cause bandwidth under-utilization and saturation. In particular, although Carina statistically balances the bandwidth usage of DRAM and CXL memory with embedding placement, the instantaneous data access may still be unbalanced when processing an individual table. For instance, if a table mainly contains cold embeddings (e.g., a large table with a small pooling factor) and stores most of its embeddings on CXL memory, most data traffic will be requested on CXL, causing saturation. Conversely, if a table mainly contains hot embeddings, CXL bandwidth may be under-utilized because most data traffic is directed to DRAM. Besides, the synchronization after processing each table may also make some threads idle waiting since threads reading CXL-resident embeddings may run for a long time.

To replace intra-table parallelism and tackle its problems, we propose asynchronous task-based parallelism. Specifically, a request batch is decomposed into tasks and then assigned to the workers (i.e. CPU threads) for execution by a scheduler. We first introduce the task definition and the working pattern of the parallelism strategy in §5.1. The scheduler controls the parallel processing of the tasks to prevent instantaneous saturation of CXL bandwidth and also takes NUMA locality into account (§5.2). In addition, we introduce an MILP-based task decomposition method to reduce request latency by breaking large tasks into small ones in §5.3.

5.1 Asynchronous Task-based Parallelism

Carina manages request processing at the task granularity and assigns each task to a single worker for embedding lookup. A task involves the lookup on a single embedding table for a micro-batch of requests (i.e., a subset of the requests in a batch). We specify a task by indices and offsets, where indices determine the embeddings to access, and offsets tell the locations to pool the accessed embeddings as output. There are many embedding tables with different workloads for an ERM. For tables with light access, all requests in a batch can be consolidated into a single task. In contrast, tables with heavy access are decomposed into multiple tasks.

Our task decomposition is efficient for execution because it groups all embeddings that contribute to the same output (i.e., for pooling). An alternative is to decompose the table accesses based on

Algorithm 1: Runtime task scheduling algorithm.

```
1  R: Current request batch under processing.
2  N_comp: NUMA node ID for a CPU.
3  N_task: NUMA node ID for a task's embedding table.
4  W_g[n]: CXL load factor for NUMA node n.
5  def Working(N_comp):
6      while True do
7          task ← Schedule(N_comp)
8          W_g[N_task] += task.cxl_ratio
9          Process(task)
10         W_g[N_task] −= task.cxl_ratio
11 def Schedule(N_comp):
12     L ← GetTasks(R, N_comp)                          // local NUMA
13     if L is Empty then
14         L ← GetTasks(R, *)                           // remote NUMA
15     task ← next task from L
16     if W_g[N_task] > W_cxl then
17         task ← min(cxl_ratio) task from L
18     Pop task from L
19     return task
```

their target memory devices (i.e., DRAM tasks and CXL tasks). However, such a method incurs extra overheads for scattering the indices to access and merging the partially pooled embeddings (i.e., from DRAM and CXL memory). Compared with intra-table parallelism, our task decomposition enjoys higher CPU cache hit rate during embedding lookup. A batch may have repeatedly accessed embeddings, and performing the lookups on the same CPU thread increases the likelihood that the embeddings are in the private CPU cache (i.e., L1 and L2) for the CPU core.

To avoid the workers being idle after task completion, Carina adopts an asynchronous execution pattern between batches. In synchronous task-based parallelism, the processing of the next batch starts only after all tasks of the previous batches are finished. In this case, some heavy tasks may become stragglers and leave the workers idle waiting, as shown in Figure 8(b). With asynchronous task-based parallelism, workers can start executing tasks from the next batch before the finish of the current batch as depicted in Figure 8(c), thus enabling the parallel embedding lookups from different batches and effectively improving CPU utilization.

5.2 Runtime Task Scheduling

We have two targets for task scheduling: (i) preventing instantaneous saturation of CXL bandwidth, and (ii) maximizing local NUMA memory access. Instantaneous saturation can result in contention among workers for limited CXL bandwidth, resulting in degraded system throughput and latency. Moreover, remote NUMA access can also degrade the performance because it is less performant than local access. To prevent CXL saturation, our scheduler monitors the CXL accesses of all running tasks and avoids scheduling CXL-intensive tasks when the data traffic on CXL is already high. To maximize local NUMA access, our scheduler prioritizes assigning tasks to workers that are co-located with the embedding tables targeted by the tasks.

Carina applies Algorithm 1 to schedule tasks. In particular, each worker is bound to a NUMA node and continuously processes tasks (lines 5-10). An array $\mathcal{W}_g$ is employed to monitor the CXL saturation level, where $\mathcal{W}_g[n]$ denotes the saturation level on NUMA node n's CXL memory. $\mathcal{W}_g[n]$ is calculated as the sum of CXL memory access ratio (i.e., CXL ratio for short) for all running tasks on node n (lines 8-10). To compute the CXL ratios with low overhead, we estimate a task's CXL ratio as the CXL memory access ratio of its target table. The scheduling steps are in lines 11-19. When an idle worker from NUMA node $\mathcal{N}_{comp}$ attempts to obtain the task list $\mathcal{L}$ of the NUMA node, it checks if all tasks of the current batch in $\mathcal{L}$ are either being processed or completed. If so, this indicates that node $\mathcal{N}_{comp}$ is ahead of other NUMA nodes. In this case, a task list from another NUMA node is randomly selected. Subsequently, a task is chosen from this task list, and the scheduler will check the CXL saturation level of the corresponding NUMA node. If $\mathcal{W}_g[\mathcal{N}_{task}]$ exceeds a predetermined CXL saturation threshold $\mathcal{W}_{cxl}$, the system seeks an alternative task with lower CXL ratio requirements within the task list. The value of $\mathcal{W}_{cxl}$ is determined by profiling the throughput of different numbers of threads concurrently accessing CXL memory (i.e., Figure 7(b)). Specifically, $\mathcal{W}_{cxl}$ is recorded as the thread number at which further increases fail to improve throughput. Finally, the chosen task will be processed by the worker.

Carina's runtime task scheduling effectively manages the bandwidth usage of different memory devices, preventing the latency caused by saturation. Additionally, it prioritizes local NUMA access for workers and only resorts to remote NUMA memory when request distribution is imbalanced.

5.3 MILP-based Task Decomposition

For our task-based parallelism, the system achieves the maximum throughput when the task size (i.e., number of requests in a task) is set as the batch size. Reducing the task size can lead to lower request latency as large tasks are divided into smaller ones, preventing them from stalling a batch. However, a small task size degrades throughput due to the increased overhead of task scheduling and CPU launching.

Using a common task size for all embedding tables is inefficient. This is because the tables have different workloads due to the differences in their pooling factors. For a table with light workloads, a batch can be completed in a short time, and decomposing the batch into smaller tasks can introduce significant overheads. In contrast, for a table with heavy workloads, the overheads of task management can be marginal. As such, to achieve low request latency while maintaining the system throughput, we identify the tables that generate long latency tasks and assign smaller task sizes to them, while maintaining the largest task sizes (i.e., batch size B) for the other tables.

We devise an MILP (i.e., mixed integer linear programming) based solution to configure task size for each table. Given a batch size B and table set $\mathcal{T}$, we aim to determine task size s_t for each table $t \in \mathcal{T}$. We constrain that the task size is selected from the following set.

$$\mathcal{S} = \{1, 2, ..., \frac{1}{4}B, \frac{1}{2}B, B\}, \quad s_t \in \mathcal{S}, \quad \forall t \in \mathcal{T} \tag{4}$$

Running the system under different task sizes $s \in \mathcal{S}$ and recording the average latency for each table yields a latency matrix L, where $l_t^{s_t}$ represents the task latency of table t under task size s_t. The throughput of table t is defined as the product of task size s_t and the size of the data read by each request (i.e., $pool_t \cdot d$, pooling factor $pool_t$ times dimension d), divided by the latency $l_t^{s_t}$. Consequently, the system throughput $\mathcal{P}$ is the summation of the throughput over all tables. To account for a tolerable decrease in throughput, we introduce a factor ϵ to represent the allowed reduction (e.g., $\epsilon = 0.01$ by default). We constrain the system throughput must exceed $(1-\epsilon) \cdot \mathcal{P}_{max}$.

$$\mathcal{P} = \sum_{t \in \mathcal{T}} \frac{s_t \cdot pool_t \cdot d}{l_t^{s_t}}, \quad \mathcal{P}_{max} = \sum_{t \in \mathcal{T}} \frac{B \cdot pool_t \cdot d}{l_t^B}$$
$$(1 - \epsilon) \cdot \mathcal{P}_{max} \leq \mathcal{P} \tag{5}$$

Our objective is to minimize the request latency. Therefore, we define our objective function as minimizing the maximum task latency among the tables where latency l_t^B is smaller than the average task latency $\bar{l}$.

$$\mathcal{T}' = \{t \in \mathcal{T} : l_t^B \geq \bar{l}\}, \quad \min_{t \in \mathcal{T}'} \max_{s_t \in \mathcal{S}} l_t^{s_t} \tag{6}$$

The MILP-generated task sizes enable us to identify tables with long latencies, facilitating the application of smaller task sizes exclusively to those tables. Meanwhile, tables that do not significantly contribute to request latency can retain larger task sizes to maintain system throughput. The MILP problem can be efficiently *offline* solved within minutes using CBC solver [22].

6 Performance Evaluation

In this section, we evaluate Carina with extensive experiments. The key findings include:

- Carina consistently outperforms the baselines, achieving significantly higher throughput and much lower latency.
- Our key designs, balanced embedding placement and task-based parallelism, solve the CXL congestion problem.
- Other designs of Carina (e.g., task decomposition and dynamic scheduling) also enhance system performance.

6.1 Experiment Settings

Datasets. We conducted experiments on the three datasets in Table 2. *FB856* and *FB788* are Open-sourced by Meta, they are synthetic but preserve the embedding access patterns of ERMs in production [61]. Multi-hot Criteo [19] is older and has smaller pooling factors than the two more recent datasets. The datasets provide requests with ID-type features, and we built an embedding table for each ID-type feature by randomly generating float vectors as the embeddings. Following EMS-i [78], we used the first 128 embedding tables of *FB856* and the first 112 embedding tables of *FB788* due to the limited memory capacity of our experiment platform. *Pool.* is the average pooling factor of the used embedding tables, and *Tot. Embs.* denotes the total number of embeddings. *Dim.* is the default dimension of the embeddings, and *Size* is the storage size of the embedding tables.

Baseline systems. Open-source ERM embedding serving systems such as TorchRec [30] and TFRS [24] store embeddings on DRAM and adopt intra-table parallelism. Carina is built upon TorchRec, which is included in Meta's production stack and widely used by many companies. We adapt TorchRec to construct the following baselines:

- *Base (B)* uses numactl [1] to interleave the memory devices (i.e., DRAM and CXL memory of all NUMA nodes) to store the embeddings and runs intra-table parallelism. It serves as a naive CXL-oblivious baseline.
- *B+P* enhances *Base* with balanced embedding placement, which includes embedding-wise tiering for DRAM and CXL, and table-wise sharding among NUMA nodes.
- *B+T* improves *Base* with asynchronous task-based parallelism. Tasks are randomly assigned to workers.

Table 2. Statistics of the datasets used in the experiments.

Name	Used Tabs.	Pool.	Tot. Embs.	Dim.	Size (GB)
FB856	128	36.9	433,890,022	128	206.9
FB788	112	21.9	947,384,548	128	451.7
Criteo	26	8.2	204,184,588	128	97.4

- *B+P+T* adds both balanced embedding placement and task-based parallelism to *Base*. Task decomposition is adopted.
- *B+P+T+S (i.e., Carina)* adds the task scheduling in Algorithm 1 to *B+P+T* and completes our Carina system.

Since CXL memory is commonly used as an extension of DRAM, we also compare Carina with OS-level memory management baselines, i.e., interleaving [1], TPP [60], and AutoNUMA [2]. Other hardware-specific ERM serving systems (e.g., EV-Store [47] and PetPS [82]) have designs that are tightly coupled with their targeted hardware features, and running them in the CXL environment will discard essential parts of their designs. As such, we do not compare them with Carina.

Testbed. We conducted all experiments on a server that hosts a Genuine Intel(R) SPR@2.5GHz CPU with two NUMA nodes, each having 48 cores and 251GB DRAM. Each NUMA node is also connected to two 128GB CXL 1.1 memory devices via PCIe 5.0 x8 lanes. Each CXL memory device has a bandwidth of 16GB/s. Notably, although we evaluate our system on a CXL server with CXL memory modules, Carina's designs are compatible with CXL-enabled servers that utilize other memory expansion devices, such as PMEM with CXL support [6]. The operating system is Ubuntu-22.04.1 with Linux kernel version 6.2.0-26, and the version of Libtorch is 1.13.0.

Performance metrics. Our primary metrics are system throughput, measured by the average number of requests processed per second (RPS), and average request latency. We also report the 99 percentile (i.e., P99) request latency since tail latency is critical for quality of service (QoS). They are measured by taking the average (or P99) of 5,000 runs. To understand how well the systems utilize the bandwidth of the memory devices, we report the real-time throughput measured as the amount of data loaded per second (GB/s) over 100μs time windows.

Implementation. We implemented Carina on the top of TorchRec with about 5k lines of C++ code. For tiered lookup, we follow FBGEMM [42] and use asmjit [45] to generate efficient assembly code, which allows a CPU thread to access the hot embeddings in DRAM and cold embeddings in CXL memory. For each table, we encode a hash function whose keys are the embedding IDs and values are the memory address to access the embeddings. We use `prefetcht0` to prefetch the embedding to the CPU cache.

6.2 Main Results

Figure 9 reports the throughput, average request latency, and P99 request latency of Carina and the baseline systems with different request batch sizes. Note that the request batches are formed by the users, and the bath size is usually configured to meet a latency or throughput requirement. We first observe that Carina consistently outperforms the CXL-oblivious *Base* (i.e., *B*). In particular, regarding throughput, the improvement of Carina over *Base* is 5.38x on average (across the datasets and batch sizes) and up to 7.88x. The average latency of Carina is 4.04x on average and up to 7.28x lower than that of *Base*. Moreover, the baselines that incorporate the designs of Carina (e.g., *B+P* and *B+T*) all outperform *Base*, and incorporating more Carina designs leads to better performance

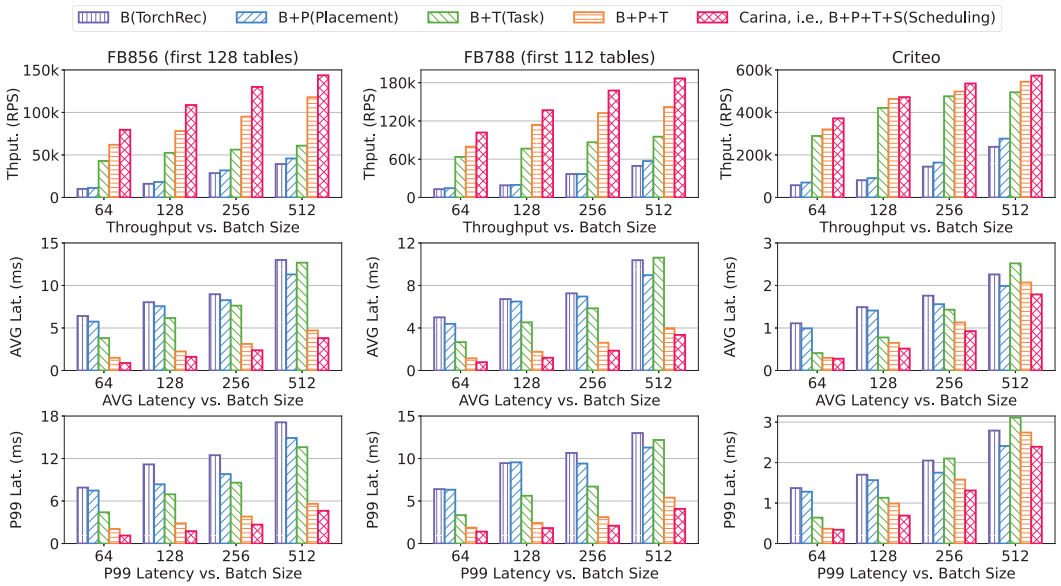


Fig. 9. The throughput, average latency, and P99 latency of Carina and the baselines at different batch sizes.

(e.g., comparing *B+P+T* with *B+P*). This suggests that Carina's designs are effective in improving the performance. We also observe that the average latency and P99 latency are similar in all cases. This is because the workloads of the requests are similar for ERM serving, and thus we omit the P99 latency in subsequent experiments.

Comparing the baselines, we observe that *B+P* improves *Base* only marginally but *B+T* yields much better performance than *Base*. This is because *B+P* still adopts synchronous intra-table parallelism, which introduces a synchronization barrier for each embedding table and keeps other threads waiting for the threads accessing slower CXL memory. However, embedding placement is important as *B+P+T* significantly improves *B+T*. This is because without our balanced embedding placement, *B+T* directs too much traffic to CXL memory, which saturates CXL bandwidth and makes CXL the bottleneck. Finally, by incorporating our fine-grained task scheduling, Carina improves the throughput of *B+P+T* by up to 1.39x and reduces the latency by up to 1.69x. This is because statistically balancing the traffic load of CXL and DRAM with embedding placement (i.e., *B+P+T*) is insufficient, and CXL may still become temporary bottlenecks if too many tasks access it in a short time. Our runtime task scheduling solves this problem by controlling the real-time utilization of CXL bandwidth.

Figure 9 also shows that Carina achieves larger improvement over *B+P+T* and *B+T* on both FB856 and FB788 than the Criteo dataset. This is because Criteo has fewer embedding tables and smaller pooling factors, which result in lighter data traffic during request processing and hence make the effect of CXL congestion less significant. For the same reason, both *Base* and *B+P* have low throughput on Criteo, especially at a small batch size. This is because they use all threads to process each table at a time (i.e., intra-table parallelism), and the parallelism is under-utilized since the workload is light on each table for Criteo. We note that FB856 and FB788 are more representative of recent ERMs with large embedding tables and pooling factors [67].

Varying embedding dimension. Figure 10 reports the throughput and average latency of Carina and the baselines when changing the dimension of the embeddings. As the results for different

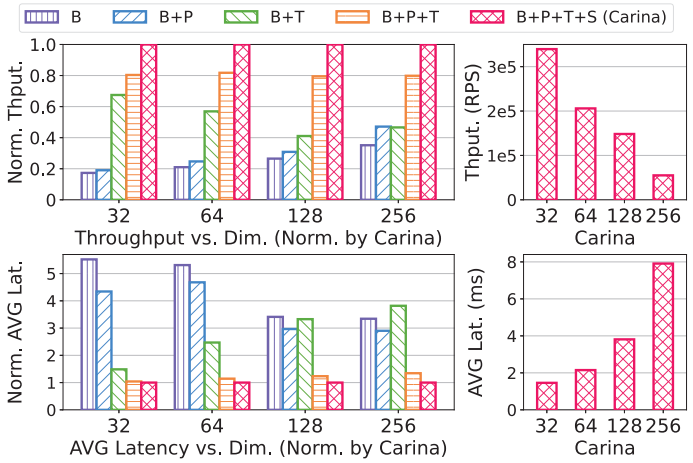


Fig. 10. Normalized throughput and average latency (with Carina as 1) when using different embedding dimension. The dataset is *FB856*, and the batch size is 512.

embedding dimensions span a large range, we normalize them by the results of Carina to make Figure 10 easy to read. The results show that Carina consistently outperforms all baselines at different embedding dimensions. The performance of Carina (and all baselines) degrades when increasing the dimension because a larger dimension means that reading each embedding consumes more data traffic. The throughput degradation can be more than linear (e.g., Carina's throughput at dim-128 is 2.6x of dim-256) because DRAM caches fewer embeddings due to limited capacity, and thus slower CXL handles the traffic for more embeddings.

An interesting phenomenon is that *B+T* performs like *B+P+T* at a small embedding dimension (e.g., 32) but becomes similar to *Base* at a large embedding dimension (e.g., 256). This is because issuing enough tasks is important to fully utilize CXL bandwidth when each embedding is small but a small number of tasks can already saturate CXL bandwidth when each embedding is large. In this case, placement (i.e., *B+P+T* vs. *B+T*) significantly improves throughput because the data traffic on CXL is reduced by storing the hot embeddings in DRAM. We also observe that *B+T* may yield longer request latency than *Base*. This is because *Base* processes one batch at a time while *B+T* may use some threads for the subsequent batches with asynchronous task execution. As such, the current batch can be stalled due to the contention for CXL bandwidth caused by the other batches. However, such a negative impact on latency is only observed at high data traffic (i.e., large embedding dimension) and resolved by jointly considering embedding placement and task execution (i.e., *B+P+T*).

Bandwidth utilization. Figure 11 reports how Carina and the baselines utilize DRAM and CXL bandwidth, and the results validate our observations for Figure 9. In particular, *Base* has similar traffic on CXL and DRAM because it interleaves embedding storage on the two types of devices. By placing the hot embeddings on DRAM, *B+P* has much larger DRAM traffic than *Base* but both CXL and DRAM bandwidth are not fully utilized because the threads are stalled due to synchronization and cannot issue enough read requests. *B+T* can issue enough read requests by avoiding synchronization but CXL bandwidth becomes the bottleneck since it lacks the embedding placement optimizations. By incorporating fine-grained task scheduling, *B+P+T+S* (i.e., Carina) utilizes both CXL and DRAM bandwidth more effectively than *B+P+T*, justifying the good performance of Carina.

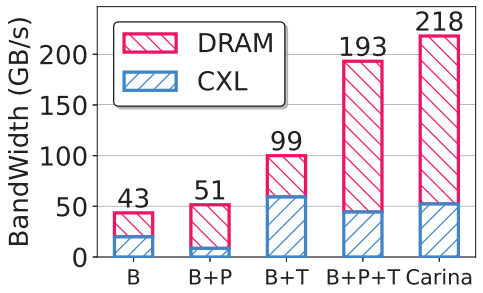


Fig. 11. DRAM and CXL memory bandwidth usage. The dataset is *FB856*, and the batch size is 512.

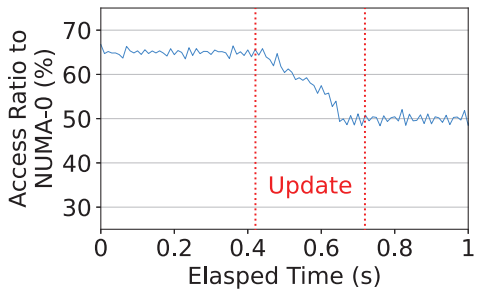


Fig. 12. The ratio of the data traffic on one NUMA node with embedding placement adjusting. The machine has two NUMA nodes. The NUMA node is overloaded initially but the traffic is balanced after the adjusting.

Placement adjusting. Figure 12 shows the access ratio between two NUMA nodes during a placement adjusting. The dataset is *FB856*, and the batch size is 512. Since there are no public traces, we simulated a scenario to trigger an update. Initially, the accesses to both NUMA nodes are balanced, and then we switch to an imbalance query trace, with a 15% disparity in access. The results show that adjusting is quick, taking 298ms, which is much faster than a complete re-initialization (taking 330s). This is because we generate the new plan with minimum data movement and copy small tables among NUMA nodes, thereby ensuring that the amount of data to be copied is small and can be transferred quickly.

Comparing with OS-level memory management. We compare Carina with the SOTA OS-level memory management systems in Figure 13. Specifically, *Interleave* uses numactl [1] to interleave all memory devices (i.e., equals to *B+T*). We also evaluate two dynamic and transparent OS page migration systems: *TPP* [60], a transparent page placement for CXL-enabled tiered memory that migrates frequently-accessed pages from CXL memory to DRAM, and *AutoNUMA* [2], also known as NUMA Balancing, automatically moves pages between NUMA nodes to reduce remote-NUMA access. AutoNUMA is also the default page migration policy of Linux. For a fair comparison, we enable the Carina execution engine for all baselines and disable the memory-related optimizations (i.e., embedding placement and NUMA control) for the OS memory management baselines.

Figure 13 shows that Carina outperforms all OS-level memory management baselines under all settings. This performance advantage can be attributed to Carina's tailored optimizations for the ERM workload, as opposed to solely focusing on hardware settings. When comparing *Interleave*, *AutoNUMA* and *TPP*, we observe similar performance across all settings. The OS-level dynamic

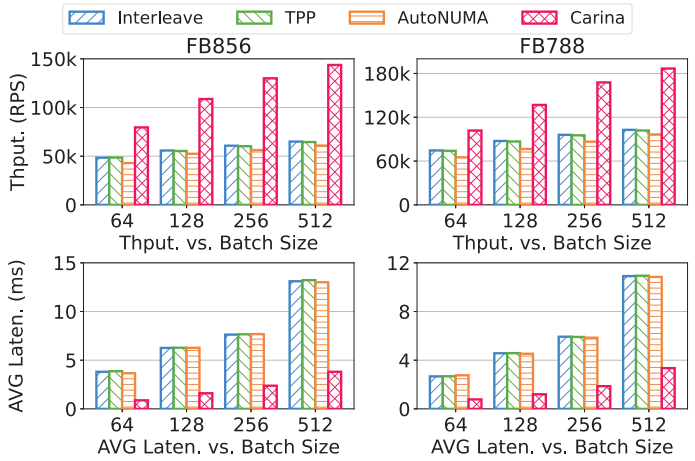


Fig. 13. Comparing Carina and OS-level memory management methods with the same execution engine.

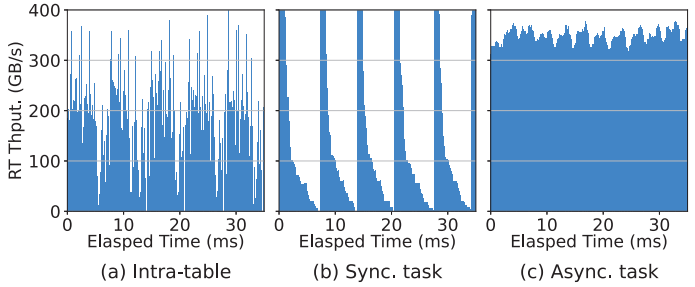


Fig. 14. Real-time throughput when Carina uses different parallelism strategy for task execution.

page migration incurs a higher data movement overhead between CXL memory and DRAM due to their 4KB data migration granularity, and they result in performance nearly the same as *Interleave*. Notably, *AutoNUMA* exhibits a 10% reduction in throughput compared to *Interleave*, primarily because the NUMA data migration policy does not recognize CXL memory devices, leading to higher overhead, as previously discussed in [60].

6.3 Design Choices

In this part, we evaluate the designs of Carina. We use the *FB856* dataset and a batch size of 512 by default, and remark that the observations are similar under other configurations.

Asynchronous task-based parallelism. Figure 14 compares different parallelism strategies by reporting their real-time data read throughput. In particular, intra-table parallelism (*Intra*) is adopted by existing DRAM-based ERM embedding serving systems; synchronous task-based parallelism (*Sync*) synchronizes for each batch of requests; asynchronous task-based parallelism (*Async*) is our final solution. Since our bandwidth-aware task scheduling does not apply to intra-table parallelism, we disable it in this experiment for a fair comparison.

We observe that the real-time throughput of *Intra* fluctuates over time. This is because some embedding tables have a small read workload and cannot fully utilize memory bandwidth. The throughput fluctuation is more significant for *Sync* because the data traffic is high when a batch

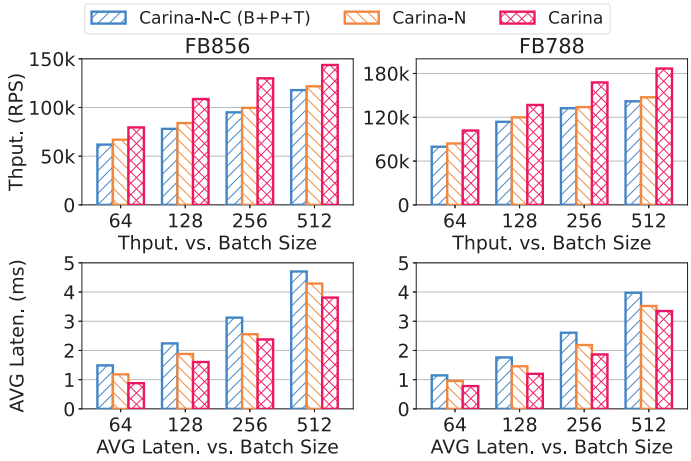


Fig. 15. Performance of Carina with/without the runtime task scheduling designs.

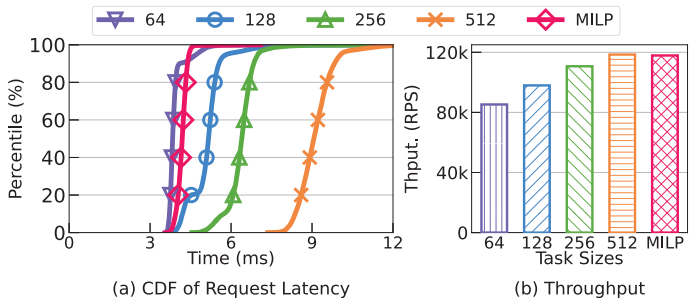


Fig. 16. Latency CDF and throughput when using different static task sizes (e.g., 64 means a task size of 64 requests) and our MILP-based task decomposition.

starts and all threads are running tasks, while the data traffic becomes low when the batch is waiting for some long-running tasks. The throughput of task parallelism may exceed hardware bandwidth because some very hot embeddings are in CPU cache, and they can be read without a load from the memory. By removing synchronization, *Async* maintains a high throughput over time by avoiding making the threads idle waiting for synchronization.

Runtime task scheduling. Figure 15 checks the influence of Carina's task scheduling designs. In particular, *Carina-N-C* disables both NUMA locality control and CXL saturation control, while *Carina-N* disables only NUMA locality control (by randomly assigning the tasks to workers). Comparing *Carina-N* with *Carina-N-C*, we observe that CXL saturation control reduces request latency by up to 1.26x. This is because CXL saturation control ensures that CXL is not saturated, and thus the threads will not be blocked by CXL access. With NUMA locality control, Carina improves the throughput of *Carina-N* by up to 1.30x. This is because NUMA locality control ensures that most embedding accesses are conducted on local NUMA nodes while remote NUMA access has lower bandwidth and higher latency.

MILP-based task decomposition. Figure 16 compares the performance of our task decomposition using fixed task sizes. The results show that larger task sizes yield higher throughput but longer

latency. This is because larger task sizes produce fewer tasks and thus reduce the task initialization and thread communication overheads. However, a request batch is also more likely to be stalled by the long-running tasks because the batch needs to wait for all its tasks to finish. Our MILP-based task decomposition resolves the conflict between throughput and latency by achieving a throughput like large task size and latency like small task size. This is because task decomposition can use different task sizes for different embedding tables, and it applies smaller task sizes for tables with heavy workloads (i.e., tables with larger pooling factors require more embedding reads) to eliminate long-running tasks.

7 Related Work

ERM serving systems. As embedding-based recommendation models (ERMs) are widely adopted [17, 26, 28, 53], many systems are designed to serve them efficiently [18, 24, 25, 27, 30, 31, 38, 39, 57] with embedding lookup as the main focus. Most systems (e.g., TorchRec [30], TFRS [24], and DeepRecSys [27]) store the embedding tables in DRAM for low access latency and adopt intra-table parallelism during execution. To reduce the data transfer overhead between embedding storage and computation accelerators (i.e., GPU or FPGA), some systems cache the hot embeddings in GPU memory [67, 71, 79, 81] or FPGA HBM [34, 37, 78]. Since GPU memory is much smaller than DRAM, Fleche [81] and UGache [71] design cache replacement and eviction policies. To handle large embedding tables, some systems use slow persistent storage like PMEM [16, 20, 82] and SSD [69, 72, 80]. RM-SSD [72] divides embedding lookup into two stages to leverage the memory controller that addresses the read amplification caused by the 4KB access granularity of SSD. To reduce slow disk access, EVStore [47] conducts approximate embedding serving by applying lossy compression to store more embeddings in DRAM and replacing a disk-resident embedding with a similar DRAM-resident embedding. Some systems scale ERM serving to distributed servers [25, 31, 39, 57, 67]. For instance, Guo et al. [25] reduce the data movement overhead by utilizing smartNICs to conduct embedding pooling. Production deployments (e.g., Kraken [83], DisaggRec [40] and FleetRec [35]) usually disaggregate embedding serving and model computation into separate services for flexibility, scalability, and resource utilization [35, 40, 56, 83, 89]. However, ERM training systems like FAE [9] and FEC [58] still utilize the aggregated design, where embedding lookup and model computation co-locate on the same set of servers.

Carina utilizes CXL memory to store the embeddings since CXL provides high performance and good scalability. Existing systems consider either a single type of storage device (i.e., DRAM) or two storage devices with a large performance gap (i.e., SSD vs. DRAM [72, 80], DRAM vs. GPU memory [71, 81]) while the performance gap between CXL and DRAM is much smaller. As such, Carina introduces fine-grained optimizations for embedding placement and task execution to fully utilize all memory devices while avoiding bottlenecks. Moreover, RecShard [67] and AutoShard [87] balance workload across distributed GPU servers by generating a static table placement. Carina consider the balance problem in a different scenario in which the threads are available to access cross-NUMA data, and the placement can also be adjusted to adopt access pattern changes. Some systems observe that some groups of embeddings are commonly utilized by the requests and store their aggregated results to reduce data access [37, 49, 84]. This optimization is orthogonal to Carina and can be easily incorporated. The partially aggregated results can be treated as high-frequency embeddings and organized according to our balanced placement policy.

More recently, some systems [55, 86] have also shown that CXL environments are well-suited for ERM workloads. CLAY [86] proposes a CXL-based NDP architecture for the embedding layer by hardware modification. ReCXL [55] focuses on efficient ERM training, introducing optimizations such as model update scheduling and hardware prefetching. However, these studies conduct

experiments in simulated CXL environments (e.g., ReCXL utilizes DRAM in remote-NUMA node to mimic CXL memory). Carina focuses on the role of NUMA within CXL architecture, drawing insights from evaluations performed on real CXL-enabled servers.

CXL systems. CXL [4] enables servers to connect to memory devices with PCIe lanes and scale beyond single-machine memory with high performance. It has been shown that directly utilizing CXL memory or interleaving CXL memory with DRAM as the underlying storage can yield DRAM-like performance for workloads in OLTP database [10, 73], KV-store [48, 75], and graph processing [66]. These applications are not affected because they conduct small data accesses and hence are not bandwidth-bound. To better utilize fast DRAM, TPP [60], SMT [44], NeoMem [93] and FreqTier [70] explicitly manage page replacement to put the hot pages in DRAM instead of simply interleaving DRAM and CXL. Some research optimizes specific applications on heterogeneous memory platforms with CXL. For instance, CXL-ANNS [32, 33] targets vector similarity search and builds a CXL prototype that can compute vector similarity for computation push-down. CXL-PNM [43, 64] builds a new CXL memory architecture with high bandwidth and software stack to accelerate large language model inference. Pond [14, 51] builds a CXL-based memory pool for cloud platforms to reduce DRAM costs. Carina targets ERM embedding serving, which is a read-only workload and bandwidth-hungry. As CXL memory's bandwidth is notably lower than DRAM, we conduct careful designs to fully utilize CXL while avoiding saturating its bandwidth.

8 Conclusion

We presented Carina, an efficient ERM embedding serving system for heterogeneous memory platforms with CXL. We identified the key challenge of CXL-oriented embedding serving as CXL congestion, which is caused by the limited bandwidth of CXL and makes CXL the bottleneck. To address this problem, we proposed balanced embedding placement with embedding-wise tiering and table-wise sharding to balance both traffic load and space consumption for the memory devices. We also adopted bandwidth-aware task execution with dedicated runtime task scheduling and task decomposition algorithms to control the real-time utilization of CXL bandwidth. Our experimental results show that Carina significantly outperforms a CXL-oblivious baseline in both system throughput and request latency, and that Carina's designs are effective.

Acknowledgments

We thank the reviewers for their valuable comments. This work was supported by Alibaba Innovative Research Program. This work was partially supported by Hong Kong General Research Fund (14208023), Hong Kong AoE/P-404/18, and the Center for Perceptual and Interactive Intelligence (CPII) Ltd under InnoHK supported by the Innovation and Technology Commission. This work was supported by Research Grants 8601116, 8601594, and 8601625 from the UGC of Hong Kong. We thank Han Long from HKUST for the mathematical analysis support.

References

[1] 2003. Linux numactl command. https://linux.die.net/man/8/numactl.
[2] 2012. NUMA Balancing (AutoNUMA). https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.
[3] 2020. Compute Express Link® 2.0 White Paper. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL2.0_White_Paper_November-2020_FINAL.pdf.
[4] 2022. CXL 3.0 Specification. https://www.computeexpresslink.org/download-the-specification/.
[5] 2022. Intel Optane DC PMM. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.
[6] 2023. Compute Express Link™(CXL™): Supporting Persistent Memory. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-2.0-Presentation-Persistent-Memory-20210615_FINAL.pdf.

[7] 2023. Micron Launches Memory Expansion Module Portfolio to Accelerate CXL 2.0 Adoption. https://investors.micron.com/news-releases/news-release-details/micron-launches-memory-expansion-module-portfolio-accelerate-cxl.
[8] 2023. Xconn Technologies: CXL 2.0 Memory Pooling (Sharing) Using Xconn Switch. https://www.xconn-tech.com/product.
[9] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating Recommendation System Training by Leveraging Popular Choices. *Proc. VLDB Endow.* 15, 1 (2021), 127–140. doi:10.14778/3485450.3485462
[10] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. ACM, 8:1–8:5. doi:10.1145/3533737.3535090
[11] Qingyao Ai, Yongfeng Zhang, Keping Bi, Xu Chen, and W. Bruce Croft. 2017. Learning a Hierarchical Embedding Model for Personalized Product Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*. ACM, 645–654. doi:10.1145/3077136.3080813
[12] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. 2023. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, HotNets 2023, Cambridge, MA, USA, November 28-29, 2023*. ACM, 25–32. doi:10.1145/3626111.3628201
[13] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2023. WiscSort: External Sorting For Byte-Addressable Storage. *Proc. VLDB Endow.* 16, 9 (2023), 2103–2116. doi:10.14778/3598581.3598585
[14] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38. doi:10.1109/MM.2023.3241586
[15] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*. ACM, 392–404. doi:10.1145/3572848.3577528
[16] Cheng Chen, Yilin Wang, Jun Yang, Yiming Liu, Mian Lu, Zhao Zheng, Bingsheng He, Weng-Fai Wong, Liang You, Penghao Sun, Yuping Zhao, Fenghua Hu, and Andy Rudoff. 2023. OpenEmbedding: A Distributed Parameter Server for Deep Learning Recommendation Models using Persistent Memory. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2976–2987. doi:10.1109/ICDE55515.2023.00228
[17] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*. ACM, 7–10. doi:10.1145/2988450.2988454
[18] Yujeong Choi, Jiin Kim, and Minsoo Rhu. 2024. ElasticRec: A Microservice-based Model Serving Architecture Enabling Elastic Resource Scaling for Recommendation Models. *CoRR* abs/2406.06955 (2024). doi:10.48550/ARXIV.2406.06955 arXiv:2406.06955
[19] CriteoLabs. 2014. Criteo display ad challenge. https://www.kaggle.com/c/criteo-display-ad-challenge.
[20] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org. https://proceedings.mlsys.org/book/277.pdf
[21] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
[22] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. In *Emerging theory, methods, and applications*. INFORMS, 257–277.
[23] Zhabiz Gharibshah and Xingquan Zhu. 2022. User Response Prediction in Online Advertising. *ACM Comput. Surv.* 54, 3 (2022), 64:1–64:43. doi:10.1145/3446662
[24] Google. 2021. TensorFlow Recommenders. https://github.com/tensorflow/recommenders.
[25] Anqi Guo, Yuchen Hao, Chunshu Wu, Pouya Haghi, Zhenyu Pan, Min Si, Dingwen Tao, Ang Li, Martin C. Herbordt, and Tong Geng. 2023. Software-Hardware Co-design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th International Conference on Supercomputing, ICS 2023, Orlando, FL, USA, June 21-23, 2023*. ACM, 336–347. doi:10.1145/3577193.3593724

[26] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. ijcai.org, 1725–1731. doi:10.24963/IJCAI.2017/239

[27] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 982–995. doi:10.1109/ISCA45697.2020.00084

[28] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 488–501. doi:10.1109/HPCA47549.2020.00047

[29] Hyunwoo Hwangbo, Yang Sok Kim, and Kyung Jin Cha. 2018. Recommendation system development for fashion retail e-commerce. *Electron. Commer. Res. Appl.* 28 (2018), 94–101. doi:10.1016/J.ELERAP.2018.01.012

[30] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. 2022. TorchRec: a PyTorch Domain Library for Recommendation Systems. In *RecSys '22: Sixteenth ACM Conference on Recommender Systems, Seattle, WA, USA, September 18 - 23, 2022*. ACM, 482–483. doi:10.1145/3523227.3547387

[31] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. 2023. Optimizing CPU Performance for Recommendation Systems At-Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*. ACM, 77:1–77:15. doi:10.1145/3579371.3589112

[32] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[33] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2024. Bridging Software-Hardware for CXL Memory Disaggregation in Billion-Scale Nearest Neighbor Search. *ACM Transactions on Storage* (2024). https://dl.acm.org/doi/10.1145/3639471

[34] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2021/hash/9e9a5486cb2f8e44d5b5fedd2a9e5fcd-Abstract.html

[35] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*. ACM, 3097–3105. doi:10.1145/3447548.3467139

[36] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *CoRR* abs/2404.12457 (2024). doi:10.48550/ARXIV.2404.12457 arXiv:2404.12457

[37] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo Ro. 2021. SPACE: Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 679–691. doi:10.1109/ISCA52012.2021.00059

[38] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*. IEEE, 790–803. doi:10.1109/ISCA45697.2020.00070

[39] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S. Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-Aware Inference Serving for At-Scale Personalized Recommendation. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 141–154. doi:10.1109/HPCA53966.2022.00019

[40] Liu Ke, Xuan Zhang, Benjamin Lee, G. Edward Suh, and Hsien-Hsin S. Lee. 2022. DisaggRec: Architecting Disaggregated Systems for Large-Scale Personalized Recommendation. *CoRR* abs/2212.00939 (2022). doi:10.48550/ARXIV.2212.00939 arXiv:2212.00939

[41] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News* 20, 2 (1992), 13–21.

[42] Daya Shanker Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. 2021. FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference. *CoRR* abs/2101.05615 (2021). arXiv:2101.05615 https://arxiv.org/abs/2101.05615

[43] Byeongho Kim, Sanghoon Cha, Sangsoo Park, Jieun Lee, Sukhan Lee, Shinhaeng Kang, Jinin So, Kyungsoo Kim, Jin Jung, Jong-Geon Lee, Sunjung Lee, Yoonah Paik, Hyeonsu Kim, Jin-Seong Kim, Won-Jo Lee, Yuhwan Ro, Yeongon Cho, Jin Hyun Kim, Joon-Ho Song, Jaehoon Yu, Seungwon Lee, Jeonghyeon Cho, and Kyomin Sohn. 2024. The Breakthrough Memory Solutions for Improved Performance on LLM Inference. *IEEE Micro* 44, 3 (2024), 40–48. doi:10.1109/MM.2024.3375352

[44] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander. *IEEE Micro* 43, 2 (2023), 20–29. doi:10.1109/MM.2023.3240774

[45] Petr Kobalicek. 2010. AsmJit Library. asmjit.com.

[46] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[47] Daniar Heri Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. 2023. EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 281–294. doi:10.1145/3575693.3575718

[48] KyungSoo Lee, Sohyun Kim, Joohee Lee, Donguk Moon, Rakie Kim, Honggyu Kim, Hyeongtak Ji, Yunjeong Mun, and Youngpyo Joo. 2024. Improving key-value cache performance with heterogeneous memory tiering: A case study of CXL-based memory expansion. *IEEE Micro* (2024).

[49] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: efficient embedding reduction on commodity hardware via sub-query memoization. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 302–313. doi:10.1145/3445814.3446717

[50] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of the Second Conference on Machine Learning and Systems, SysML 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2019/hash/1eb34d662b67a14e3511d0dfd78669be-Abstract.html

[51] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 574–587. doi:10.1145/3575693.3578835

[52] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based Product Retrieval in Taobao Search. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*. ACM, 3181–3189. doi:10.1145/3447548.3467101

[53] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 1754–1763. doi:10.1145/3219819.3220023

[54] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 3288–3298. doi:10.1145/3534678.3539070

[55] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. 2024. Enabling Efficient Large Recommendation Model Training with Near CXL Memory Processing. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. IEEE, 382–395. doi:10.1109/ISCA59077.2024.00036

[56] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. 2022. Monolith: Real Time Recommendation System with Collisionless Embedding Table. In *Proceedings of the 5th Workshop on Online Recommender Systems and User Modeling co-located with the 16th ACM*

Conference on Recommender Systems, ORSUM@RecSys 2022, Seattle, WA, USA, September 23rd, 2022 (CEUR Workshop Proceedings, Vol. 3303). CEUR-WS.org. https://ceur-ws.org/Vol-3303/paper8.pdf

[57] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. 2021. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 162–171. doi:10.1109/ISPASS51385.2021.00033

[58] Kaihao Ma, Xiao Yan, Zhenkun Cai, Yuzhen Huang, Yidi Wu, and James Cheng. 2023. FEC: Efficient Deep Recommendation Model Training with Flexible Embedding Communication. *Proc. ACM Manag. Data* 1, 2 (2023), 165:1–165:21. doi:10.1145/3589310

[59] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473

[60] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. 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, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 742–755. doi:10.1145/3582016.3582063

[61] Meta. 2022. Facebook DLRM datasets. https://github.com/facebookresearch/dlrm_datasets.

[62] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F. Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-Throughput Vector Similarity Search in Knowledge Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 197:1–197:25. doi:10.1145/3589777

[63] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 533–549. https://www.usenix.org/conference/osdi21/presentation/mohoney

[64] Sangsoo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. 2024. An LPDDR-based CXL-PNM Platform for TCO-efficient Inference of Transformer-based Large Language Models. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 970–982. doi:10.1109/HPCA57654.2024.00078

[65] Hongyu Ren and Jure Leskovec. 2020. Beta embeddings for multi-hop logical reasoning in knowledge graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1654, 11 pages.

[66] Shintaro Sano, Yosuke Bando, Kazuhiro Hiwada, Hirotsugu Kajihara, Tomoya Suzuki, Yu Nakanishi, Daisuke Taki, Akiyuki Kaneko, and Tatsuo Shiozawa. 2023. GPU Graph Processing on CXL-Based Microsecond-Latency External Memory. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 961–972. doi:10.1145/3624062.3624173

[67] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 344–358. doi:10.1145/3503222.3507777

[68] Devashish Shankar, Sujay Narumanchi, H. A. Ananya, Pramod Kompalli, and Krishnendu Chaudhury. 2017. Deep Learning based Large Scale Visual Recommendation and Search for E-Commerce. *CoRR* abs/1703.02344 (2017). arXiv:1703.02344 http://arxiv.org/abs/1703.02344

[69] Mohammadreza Soltaniyeh, Veronica Lagrange Moutinho dos Reis, Matthew Bryson, Xuebin Yao, Richard P. Martin, and Santosh Nagarakatte. 2022. Near-Storage Processing for Solid State Drive Based Recommendation Inference with SmartSSDs®. In *ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022*. ACM, 177–186. doi:10.1145/3489525.3511672

[70] Kevin Song, Jiacheng Yang, Sihang Liu, and Gennady Pekhimenko. 2023. Lightweight Frequency-Based Tiering for CXL Memory Systems. *CoRR* abs/2312.04789 (2023). doi:10.48550/ARXIV.2312.04789 arXiv:2312.04789

[71] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. 2023. UGACHE: A Unified GPU Cache for Embedding-based Deep Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 627–641. doi:10.1145/3600006.3613169

[72] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2022. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 1056–1070. doi:10.1109/HPCA53966.2022.00081

[73] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 105–121. doi:10.1145/3613424.3614256

[74] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter R. Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *CoRR* abs/2305.10863 (2023). doi:10.48550/ARXIV.2305.10863 arXiv:2305.10863

[75] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 818–833. doi:10.1145/3627703.3650061

[76] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[77] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, and Yuanyuan Dong. 2022. PATS: Taming Bandwidth Contention between Persistent and Dynamic Memories. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*. IEEE, 885–890. doi:10.23919/DATE54114.2022.9774762

[78] Yitu Wang, Shiyu Li, Qilin Zheng, Andrew Chang, Hai Li, and Yiran Chen. 2023. EMS-i: An Efficient Memory System Design with Specialized Caching Mechanism for Recommendation Inference. *ACM Trans. Embed. Comput. Syst.* 22, 5s (2023), 100:1–100:22. doi:10.1145/3609384

[79] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. 2022. A GPU-specialized Inference Parameter Server for Large-Scale Deep Recommendation Models. In *RecSys '22: Sixteenth ACM Conference on Recommender Systems, Seattle, WA, USA, September 18 - 23, 2022*. ACM, 408–419. doi:10.1145/3523227.3546765

[80] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 717–729. doi:10.1145/3445814.3446763

[81] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. 2022. Fleche: an efficient GPU embedding cache for personalized recommendations. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 402–416. doi:10.1145/3492321.3519554

[82] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. 2023. PetPS: Supporting Huge Embedding Models with Persistent Memory. *Proc. VLDB Endow.* 16, 5 (2023), 1013–1022. doi:10.14778/3579075.3579077

[83] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 21. doi:10.1109/SC41405.2020.00025

[84] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 282–301. doi:10.1145/3582016.3582029

[85] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. ACM, 5439–5450. doi:10.1145/3580305.3599805

[86] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. 2024. CLAY: CXL-based Scalable NDP Architecture Accelerating Embedding Layers. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*. ACM, 338–351. doi:10.1145/3650200.3656595

[87] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. 2022. AutoShard: Automated Embedding Table Sharding for Recommender Systems. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 4461–4471. doi:10.1145/3534678.3539034

[88] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*.

ACM, 658–674. doi:10.1145/3600006.3613135

[89] Wei Zhang, Dai Li, Chen Liang, Fang Zhou, Zhongke Zhang, Xuewei Wang, Ru Li, Yi Zhou, Yaning Huang, Dong Liang, Kai Wang, Zhangyuan Wang, Zhengxing Chen, Min Li, Fenggang Wu, Minghai Chen, Huayu Li, Yunnan Wu, Zhan Shu, Mindi Yuan, and Sri Reddy. 2023. Scaling User Modeling: Large-scale Online User Representations for Ads Personalization in Meta. *CoRR* abs/2311.09544 (2023). doi:10.48550/ARXIV.2311.09544 arXiv:2311.09544

[90] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org. https://proceedings.mlsys.org/book/315.pdf

[91] Chenguang Zheng, Guanxian Jiang, Xiao Yan, Peiqi Yin, Qihui Zhou, and James Cheng. 2024. GE2: A General and Efficient Knowledge Graph Embedding Learning System. *Proc. ACM Manag. Data* 2, 3, Article 183 (May 2024), 27 pages. doi:10.1145/3654986

[92] Qihui Zhou, Peiqi Yin, Xiao Yan, Changji Li, Guanxian Jiang, and James Cheng. 2024. Atom: An Efficient Query Serving System for Embedding-based Knowledge Graph Reasoning with Operator-level Batching. *Proc. ACM Manag. Data* 2, 4, Article 193 (Sept. 2024), 29 pages. doi:10.1145/3677129

[93] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, and Guangyu Sun. 2024. Toward CXL-Native Memory Tiering via Device-Side Profiling. *CoRR* abs/2403.18702 (2024). doi:10.48550/ARXIV.2403.18702 arXiv:2403.18702

Received October 2024; revised January 2025; accepted February 2025