GPU-Disaggregated Serving for Deep Learning Recommendation Models at Scale

Lingyun Yang[†], Yongchen Wang, Yinghao Yu, Qizhen Weng[†], Jianbo Dong, Kan Liu,
Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi,
Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, Wei Wang[†]
[†]*Hong Kong University of Science and Technology* *Alibaba Group*

Abstract

Online recommender systems use deep learning recommendation models (DLRMs) to provide accurate, personalized recommendations to improve customer experience. However, efficiently provisioning DLRM services at scale is challenging. DLRMs exhibit distinct resource usage patterns: they require a large number of CPU cores and a tremendous amount of memory, but only a small number of GPUs. Running them in multi-GPU servers quickly exhausts the servers' CPU and memory resources, leaving a large number of unallocated GPUs stranded, unable to utilize by other tasks.

This paper describes Prism, a production DLRM serving system that eliminates GPU fragmentation by means of resource disaggregation. In Prism, a fleet of CPU nodes (CNs) interconnect with a cluster of heterogeneous GPU nodes (HNs) through RDMA, leading to two disaggregated resource pools that can independently scale. Prism automatically divides DLRMs into CPU- and GPU-intensive subgraphs and schedules them on CNs and HNs for disaggregated serving. Prism employs various techniques to minimize the latency overhead caused by disaggregation, including optimal graph partitioning, topology-aware resource management, and SLO-aware communication scheduling. Evaluations show that Prism effectively reduces CPU and GPU fragmentation by 53% and 27% in a crowded GPU cluster. During seasonal promotion events, it efficiently enables capacity loaning from training clusters, saving over 90% of GPUs. Prism has been deployed in production clusters for over two years and now runs on over 10k GPUs.

1 Introduction

Personalized recommender systems are the key infrastructure for many user-facing, revenue-generating web services, such as content streaming, e-commerce, social networks, and web search [12, 21, 47, 58]. These systems use deep learning recommendation models (DLRMs) to provide accurate, personalized recommendations to improve customer experience and increase user engagement. DLRM serving consumes the majority of the inference resources in today's AI cloud, with top recommendation models account for more than 79% of AI cycles, according to Meta [26].

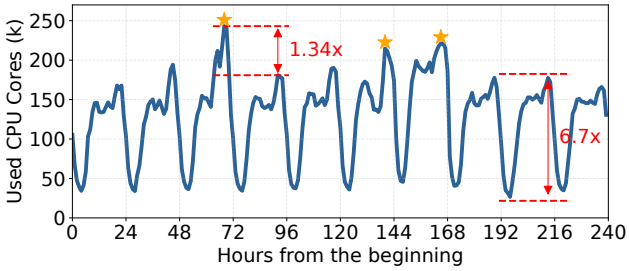


Figure 1: The CPU demands of DLRM services exhibit *daily* and *seasonal* variations; GPU demands follow the same trend. Trace collected from a production cluster, including three (starred) e-commerce promotion events.

However, serving DLRMs at scale faces aggravating challenges. DLRM serving has a stringent latency service-level objective (SLO), usually on the scale of tens of millisecond per request. In the meantime, DLRM serving needs to handle frequent spikes in demand. Meeting the latency SLOs often means provisioning for the peak load, which can be significantly higher than the average [28, 39]. Figure 1 illustrates the resource demands of Alibaba's DLRM services in a production cluster. We observe a distinct diurnal pattern with the peak-to-valley ratio over 6×; during seasonal promotion events, the peak load can be 1.3× higher than the regular peaks, which is in line with the previous report [68]. Provisioning for the peak load at such scale results in significant underutilization, making it economically unviable.

To reduce overprovisioning, a better strategy is to provision for the average load and enable *capacity loaning* during load spikes. Large companies like Alibaba own multiple purpose-specific infrastructures: some for training and the others for inference. When DLRM serving is in peak hours, it can temporarily loan GPU servers from training clusters as training jobs are less latency-sensitive and can tolerate interruptions. However, there is a mismatch between server configuration and DLRM's resource demand. Unlike training tasks that demand extensive GPU cycles, recommendation models exhibit low compute-intensity and are not bottlenecked on GPU. Instead, they perform sparse computations such as embedding [36, 37], which requires abundant memory for storing embedding tables and many CPU cores for table look-up and

pooling operations [26, 32]. As a result, running recommendation models in training servers quickly exhausts the servers' CPU and memory resources, leaving a large number of unallocated GPUs stranded. In our cluster, a typical DLRM service requests 48 CPUs and 1 GPU, while training servers commonly have ⟨96 CPUs, 8 GPUs⟩ (Figure 4). Deploying two DLRM inference instances occupies all CPUs on the host, leaving 6 unallocated GPUs unable to utilize by other tasks.

This paper presents Prism, a production DLRM serving system that addresses the resource mismatch problem with *GPU disaggregation*. Prism runs on a shared infrastructure where a fleet of CPU nodes (CNs) interconnect with a cluster of heterogeneous GPU nodes (HNs) through a high-speed RDMA network. Each CN has a large number of CPU cores and high memory but no GPU, while each HN is a typical training server with multiple GPUs but only a modest amount of CPU and memory resources. This infrastructure breaks down a cluster of monolithic servers with fixed configurations into two *disaggregated resource pools*, where CNs provide rich CPU and memory resources while HNs provide abundant GPUs. The two resource pools can be independently scaled to match the changing demands of dynamic workloads. Given a DLRM, Prism automatically divides its compute graph into two subgraphs, one containing CPU- and memory-intensive operators and the other GPU-intensive. The system then schedules the two subgraphs on the selected CN and HN for disaggregated serving and returns the results to users.

This paper describes the challenges, techniques, and lessons learned in building a disaggregated DLRM system at a production scale. First, the disaggregation needs to be *transparent to model owners*. Manually re-architecting models to a disaggregated version raises the risk of accuracy degradation and requires extra efforts from their owners, thus undesirable. Second, the system should *scale to thousands of servers* to handle excessive load spikes. Given the surging traffic, it should promptly schedule workloads to a large fleet of servers to achieve dramatic total throughput in a short period of time. Third, the system should *meet the stringent latency SLOs* of DLRM serving, in the presence of non-trivial communication overhead between CNs and HNs due to GPU disaggregation. Prism tackles these challenges with three major components: a disaggregation-optimized real-time prediction (RTP) framework that optimally partitions compute graphs between CNs and HNs (§4.1), a topology-aware resource manager that minimizes inter- and intra-server communication overhead (§4.2), and SLO-aware communication scheduling that ensures disaggregated serving within the target latency SLOs (§4.3).

Prism has been pilot deployed in late 2022 as the underlying infrastructure for a small number of production online recommendation services, with serving scale continuously increasing over the past two years. As of January 2025, Prism runs on over 10k GPUs, successfully decoupling the CPU and GPU computation demands of DLRMs without impacting service performance. Evaluations show that in daily high-allocation GPU clusters, Prism effectively reduces CPU fragmentation by 53% and GPU fragmentation by 27%; during seasonal promotion events, Prism efficiently enables capacity loaning from training clusters, saving over 90% of GPUs (§5). Our primary contributions are as follows.

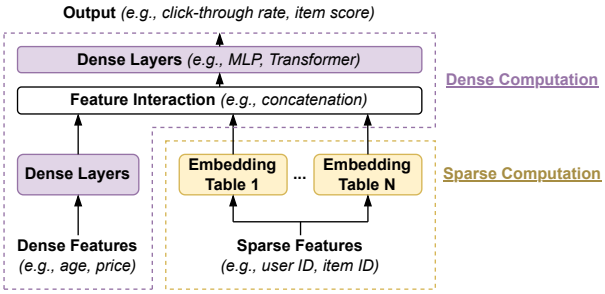


Figure 2: An architecture overview of DLRM.

- We identify the challenges in resource provisioning when deploying elastic DLRM services at a production scale and motivate the need for GPU-disaggregated serving.
- We design and implement Prism to harvest resources from CPU nodes and heterogeneous GPU nodes by means of disaggregated serving, alleviating the resource mismatch between the server configurations and DLRMs' resource demands, while still meeting latency SLOs.
- We evaluate Prism in production and demonstrate that it can effectively reduce resource fragmentation without compromising service performance, enabling efficient capacity loaning during promotion events.

We have released a production DLRM serving trace[1] as part of the Alibaba cluster trace program [3].

2 Background and Motivation

2.1 A Primer on DLRM

Modern recommendation models have a large feature set, separated into dense and sparse categories. Dense features, such as vectors and matrices, are processed by typical deep neural network layers, while sparse features are processed by indexing large embedding tables. Figure 2 illustrates a typical architecture of a DLRM [12, 47, 69]. The input comprises dense continuous features (e.g., user age and item price) and sparse categorical features (e.g., user ID and item ID). Sparse features are transformed into continuous embedding representations via a lookup operation in the embedding tables. These features are then fused and fed into a densely connected deep network, such as a multi-layer perceptron [12] or transformer [57], for prediction, eventually generating the model output, such as click-through rate or item score.

[1] https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2025

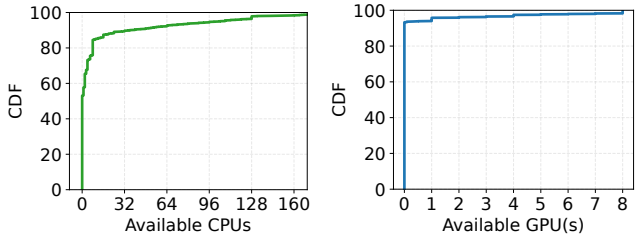


Figure 3: Resource allocation snapshot of a large-scale GPU cluster H with over 4k nodes, 640k CPU cores, and 11k GPUs.

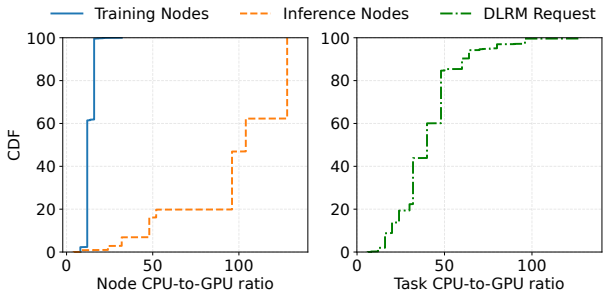


Figure 4: CDF of CPU-to-GPU ratio. The DLRM trace was collected over a period of one week from our production clusters, comprising of more than 25k instances.

DLRMs exhibit distinct computational and memory consumption characteristics compared to convolutional and recurrent neural networks [26, 30]. The embedding operations dominate the run-time of recommendation models. These operations are characterized by large embedding tables (typically on the order of tens to hundreds of gigabytes), low compute-intensity and little to no regularity, making them ill-suited to run on GPUs. In contrast, the model's dense network component is better executed on GPUs, which can achieve 10–44× speedup compared to running on CPUs, as shown in our experiments. Therefore, current DLRM serving systems decouple the sparse embedding lookup operations from dense computations. The former runs on a large number of CPUs, and the latter runs on a GPU.

2.2 Challenges in DLRM Provisioning

DLRM services serve a massive volume of requests with stringent latency SLOs and account for the majority of inference cycles in today's AI cloud [26]. These services feature dynamically changing requests with large *daily* and *seasonal* variations as shown in Figure 1. Instead of serving DLRMs in a dedicated cluster, which requires provisioning for the peak load and is economically unviable, at Alibaba we serve DLRMs along with other machine learning models in *shared clusters* in normal days; we additionally *loan* servers from training clusters to handle large load spikes during seasonal promotion events. However, this approach faces two challenges in practice.

C1: Resource fragmentation for daily DLRM serving. In shared clusters, frequently scaling DLRM provisioning based on the changing load results in severe resource fragmentation [64]. Figure 3 illustrates the distribution of available CPUs and GPUs of servers in a large GPU cluster, where both CPU and GPU allocation rates reach 90%. Despite having sufficient CPU and GPU quotas, DLRM service owners observe prolonged scheduling time or even scheduling failures [64]. From their perspective, the cluster has over 30k fragmented CPUs and more than 200 fragmented GPUs (detailed explanation in §5.4). The severe fragmentation is primarily due to the high CPU-to-GPU ratio of these inference instances (Figure 4, right), which cannot be accommodated using the servers' remaining resources. In practice, scheduling new DLRM instances often requires manually relocating running jobs to reduce fragmentation, which is time-consuming.

C2: Ineffective capacity loaning during seasonal load spikes. Throughout a year, Alibaba hosts several large-scale seasonal shopping festivals [68], during which DLRM services experience significant load spikes, exceeding the daily peak by more than 1.3× (Figure 1). To handle these transient, excessive load spikes, *capacity loaning* is often needed, which we elaborate as follows.

Production AI clouds are built from a large quantity of *heterogeneous servers* with configurations meant to support either training or inference tasks. Figure 4 (left) shows the distribution of the CPU-to-GPU ratio of the server nodes used for training and inference in a production cluster at Alibaba. In general, nodes suitable for training tasks are configured to have multiple advanced GPUs (e.g., 4 or 8 H800 GPUs) with high-speed interconnect (e.g., NVLink) but only a limited number of CPUs (e.g., 16 to 32 cores), thus having a low CPU-to-GPU ratio. In comparison, nodes optimized for inference tasks, especially DLRM serving, have a much higher CPU-to-GPU ratio, typically with a single GPU but a large number of CPU cores (e.g., 128 cores) to accelerate CPU-intensive operations, such as data preprocessing and embedding lookup. The presence of these heterogeneous servers essentially segregates a datacenter into two purpose-specific infrastructures, one for training and the other for inference. To improve utilization and reduce overprovisioning, capacity loaning between the two infrastructures should ideally be enabled. That is, when DLRM serving is in peak hours, it can temporarily loan training servers to handle excessive recommendation queries.

However, the mismatch between server configuration and resource demand renders capacity loaning ineffective. Figure 4 (right) shows the distribution of CPU-to-GPU ratio of the recommendation models deployed in our production clusters. Over 90% of DLRMs have CPU-to-GPU ratio greater

than 20, whereas *all* multi-GPU training nodes have a ratio below 20. As a result, running DLRMs on these training nodes quickly exhausts CPUs, leaving a large number of GPUs stranded, unable to utilize by other tasks.

2.3 GPU-Disaggregated DLRM Serving

Resource disaggregation [18, 23, 25, 53] holds tremendous promise in DLRM serving. This architecture allows decoupled independent scaling-out of CPUs and GPUs, thereby addressing the mismatch between server configuration and resource demands. Resource sharing is also made possible as GPU-intensive AI workloads, such as training, can run on the same infrastructure, together with recommendation models, significantly increasing the system utilization. Our design of a GPU-disaggregated DLRM inference service is motivated by two fundamental observations.

O1: Distinct characteristics of DLRMs. DLRMs exhibit a distinctive computational structure that naturally lends itself to graph partitioning based on operator-level resource affinity. Our analysis reveals that embedding table queries dominate CPU computation, while matrix multiplication accounts for the majority of GPU computational overhead (Figure 8). While DLRM services routinely update model parameters and embedding table values to reflect real-time user interaction, the underlying computational graph remains static. This architectural stability ensures that graph partitioning and disaggregation optimizations, once computed, remain optimal throughout the model's deployment lifecycle, eliminating the need for dynamic repartitioning.

O2: High-performance RDMA network. Modern RDMA network interface cards (RNICs) deliver bandwidth capabilities of 200–400 Gbps, matching or even surpassing that of PCIe 4.0 x16 interconnects (256 Gbps). RDMA's offload capabilities bypass the kernel networking stack, minimizing CPU overhead——a critical factor for inference workloads in disaggregated environments. Optimized RDMA networking can provide scalability, stability, and SLO guarantees for disaggregated inference scenarios (§5).

GPU disaggregation approaches. GPU disaggregation can be achieved using three approaches, as summarized in Figure 5. We compare these approaches and justify our choice.

1) API-level disaggregation: this approach intercepts program calls to CUDA APIs and redirects them to a remote GPU node for execution [2, 15, 16]. It requires only replacing the original CUDA library inside the container without any modification to the serving framework. However, its application agnostic nature offers little optimization opportunities. Future CUDA upgrades would also require considerable engineering effort in forward compatibility and performance tuning. Moreover, supporting heterogeneous AI accelerators (e.g., AMD GPUs) necessitates developing and maintaining distinct remoting layers for each platform.

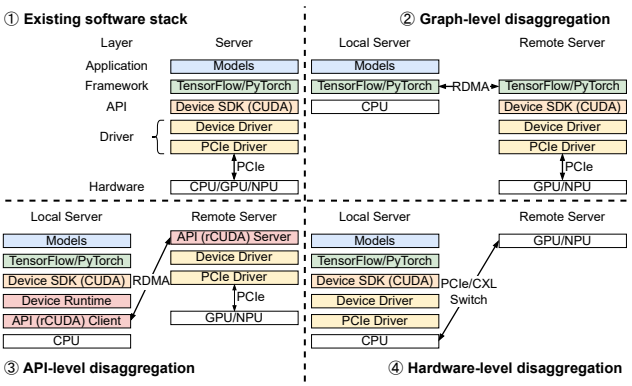


Figure 5: GPU disaggregation at different levels.

2) Hardware-level disaggregation: GPU disaggregation can be ideally enabled with specialized hardware, such as customized multi-hop PCIe switches [29, 42] and CXL technologies [1]. Hardware-level disaggregation requires no modification to software and incurs minimum I/O latency. However, it faces significant deployment barriers as it requires expensive infrastructure upgrade and customization. The disaggregation is also confined to a short distance (e.g., PCIe switches support disaggregation only within a rack).

3) Graph-level disaggregation: A DLRM is represented as a compute graph consisting of multiple operators running on CPU or GPU. Inspired by model parallelism, we can partition the compute graph into a CPU sub-graph and a GPU sub-graph (**O1**). The two sub-graphs can then be scheduled on selected CPU and GPU nodes for disaggregated execution, with communication over high-speed RDMA network (**O2**). Compared with the previous two approaches, graph-level disaggregation requires no specialized hardware and can achieve good performance with optimized partitioning, scheduling, and network transport. We hence consider it as a viable approach and base our system design on it.

3 Prism Overview

In this section, we present Prism, a large-scale DLRM system that enables GPU-disaggregated serving by means of graph partitioning. Prism has been deployed in our production clusters, serving as the underlying infrastructure for core recommendation services. As of January 2025, the system serves over 20k DLRM instances, utilizing more than 10k GPUs and 800k CPUs. As illustrated in Figure 6, Prism operates on a cluster where a fleet of heterogeneous GPU nodes (HNs) interconnects with a number of CPU nodes (CNs) via a high-speed RDMA network. A CN is configured with a large number of CPUs and high memory, while an HN is a multi-GPU node. This results in two disaggregated resource pools, a CPU pool provisioned by CNs and a GPU pool provisioned by HNs. Prism automatically partitions recommendation models for distributed inference on CNs and HNs.

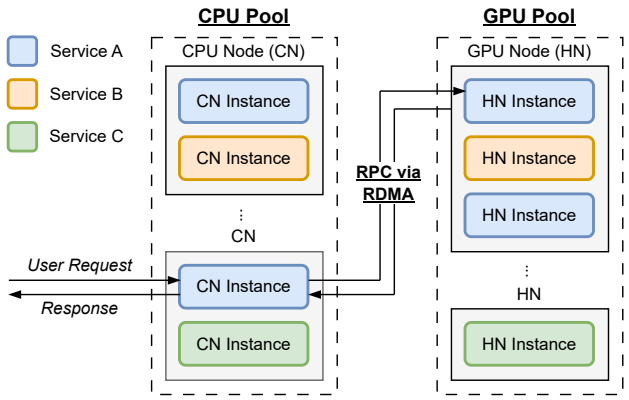


Figure 6: The cluster deployment of DLRM inference instances and request execution flow in Prism.

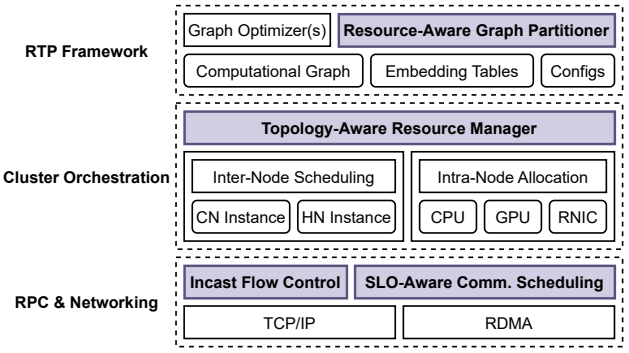
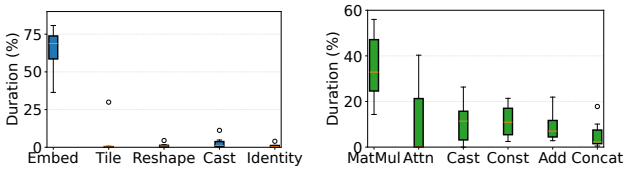


Figure 7: Prism overview and its key designs are in *purple*.

System architecture. Figure 7 provides an overview of Prism. The RTP framework (§4.1) takes input the compute graph, associated embedding tables, and configuration parameters (e.g., enabling/disabling custom optimizers). It divides a monolithic recommendation model into two subgraphs, one consisting of CPU-intensive operations and the other consisting of GPU-efficient operations. The resource manager (§4.2) then packages the two subgraphs into containerized CN and HN instances and places them onto the selected CN and HN nodes for disaggregated execution, by considering the network and node topology. The manager makes resource allocation decisions (e.g., CPUs, GPUs, RNICs) and dynamically scales CN and HN instances in response to the changing load. Once the CN and HN instances are placed on nodes, the communication scheduler establishes RDMA connections and performs incast control and SLO-aware request scheduling to meet latency SLOs (§4.3).

Execution flow. As illustrated in Figure 6, user requests are first routed to a frontend CN instances for embedding lookups and other CPU-intensive computations. Intermediate tensors are then transferred to the remote HN instance on a GPU node through RPCs over the RDMA network. After the HN instance returns results, the CN instance performs post-processing before sending the response to the user.



(a) Major CPU operators. (b) Major GPU operators.

Figure 8: Operator analysis of online DLRM services.

4 Prism Design

Prism is designed to meet three key requirements:

- *Transparency to model development and optimization:* For production deployment, it is important to make disaggregation transparent to model developers, without their cooperation to modify model architecture or implementation. It is hence essential to enable automated graph partitioning to support disaggregated inference for various recommendation models, without affecting users or invalidating original graph optimization strategies (§4.1).
- *Compliance to SLOs:* Disaggregation inevitably introduces performance overheads, necessitating a joint optimization approach across various system components to minimize the impact on service performance. For example, the compute graph must be judiciously partitioned and optimized to effectively separate the resource usage of different operators (§4.1). Furthermore, the node placement and resource allocation of instances require careful consideration to ensure good performance (§4.2).
- *Scalability*. In contrast to traditional recommendation services, GPU disaggregation introduces non-trivial communication overhead. Our system must ensure service performance remains unaffected, even under conditions of multiple server instances and high traffic loads in production environments (§4.3).

4.1 Resource-Aware Graph Partitioner

The most critical *first* step in decoupling CPU and GPU computation is to partition a model's original computation graph into two subgraphs, which are executed respectively by the CN and HN instances.

Retrofit for the existing workflow. In production DLRM systems, the recommendation framework provides model developers with a plethora of optional optimization techniques (e.g., JIT fusion, CUDA Graphs [22]) to enable or disable based on the model characteristics. These optimizers are typically applied in a *sequential manner*, iteratively rewriting the original computation graph and ultimately generating an optimized computation graph tailored for deployment. The

primary objective of our disaggregation approach is to minimize the impact on the service's inference performance. Thus, Prism introduces *graph partitioning* and *disaggregation optimization* as the final stages in the existing DLRM workflow, strategically waiting for all user-specified optimization strategies to be completed before meticulously reconstructing the computation graph for disaggregated serving.

Operator analysis of DLRMs. To gain deeper insights into the computational characteristics of DLRM services, we conducted a comprehensive profiling of the operators, as illustrated in Figure 8. The results unveil two key observations that serve as the foundation for our *resource-aware graph partitioning* algorithm: (1) Embedding lookup operators exhibit a dominant presence, accounting for over 70% of the CPU computation (Figure 8, left); consequently, we designate all embedding lookup-related operators as `CPU-intensive operators`. (2) Matrix multiplication computations emerge as the most significant contributor to GPU computation time. In more advanced models [11] incorporating transformers [57], the computation ratio of attention layers can even reach a staggering 40% (Figure 8, right); we categorize these operators as `GPU-efficient operators`.

Device placement. Leveraging the aforementioned observations, the RTP framework employs a heuristic approach to partition the GPU subgraph that necessitates computation on the HN instance: (1) `GPU-efficient operators` are selected as the initial seeds for the partitioning process. (2) Commencing from the seed operators, a Depth-First Search (DFS) coloring algorithm is simultaneously executed from both upstream and downstream directions, aiming to encompass the maximum number of operators feasible for GPU computation. (3) The DFS coloring process is terminated upon encountering `CPU-intensive operators`, ensuring an optimal balance between CPU and GPU utilization.

Optimization for disaggregation. To enhance the efficiency of distributed graph execution, we introduce two complementary strategies that achieve up to 50% reduction in inter-node data transfer volume: (1) We preserve and optimize constant subgraph execution. While constant operations constitute 10–20% of GPU computational workload (Figure 8, right), their optimization potential is often compromised when fragmented across different servers. By consolidating constant operations onto the HN instance, we maintain the structural integrity of constant subgraphs. This ensures that constant computations are executed exactly once and their results are cached for subsequent operations, eliminating redundant data transfers. (2) In cases where multiple derivative tensors are generated from a common source tensor and subsequently transmitted from the CN instance to the HN instance, we employ a size-aware transfer strategy. When derivative tensors' aggregate size exceeds the ancestor's magnitude, we optimize communication patterns by transmitting only the source tensor, thereby reducing the network bandwidth requirements.

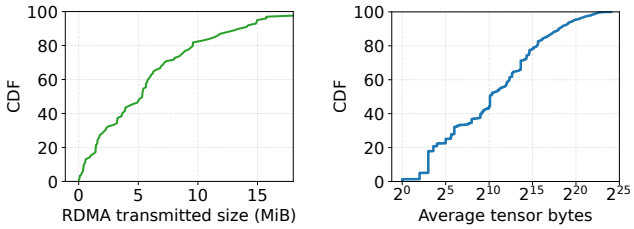


Figure 9: Statistical analysis of data transfer for top 100 disaggregated DLRM inference services. **Left**: Total volume of RDMA data transfers from a CN instance to a HN instance; **Right**: Distribution of transferred tensor sizes.

RPC for remote GPU execution. We develop a unified operator called `FusedGraphOp` for RPCs. This operator aggregates all tensors that require transmission from the CN instance and sends the requests to one of the remote HN instances. Upon completion of the remote GPU computation, the results are returned, marking the conclusion of the entire inference process. To minimize memory copies between the computational framework (e.g., TensorFlow) and the RPC system, we establish a unified memory pool where `FusedGraphOp` flags tensors requiring transmission and relinquishes control to the RPC system. By leveraging GPUDirect RDMA [7], these data are already stored in the RPC system's memory buffer upon creation, realizing defacto *zero-copy data transmission* [27, 67]. Experimental results demonstrate that this approach can yield performance improvements of 19–181%. The RPC system adaptively balances load using real-time processing latency of each HN instance [13].

Production deployment. We deploy Prism to evaluate DLRMs in production. In 80% of recommendation services, the data transfer size via the RDMA network is less than 10 MiB (Figure 9, left). Note that we focus on unidirectional data transfer from the CN instance to the HN instance, as the data sent back is typically *negligible* (e.g., approximately 100 KiB). The results presented in §5 demonstrate that, at this scale of transfer size, disaggregated inference can effectively separate resource requirements without compromising service performance. Interestingly, we observe a *bimodal* distribution in the size of transferred tensors. Only 11% of the tensors exceed 0.25 MiB, while a significant 44% of the tensors are smaller than 1 KiB (Figure 9, right). These findings motivate us to adopt a size-based approach when transferring tensors. For small tensors (e.g., $\leq$ 4 KiB), we directly employ RDMA send/recv operations, whereas for large tensors, we utilize RDMA write operations.

Implementation. Our implementation is built upon an optimized internal version of TensorFlow v1.12 [9], comprising approximately 2k lines of Python code for graph partitioning and disaggregation optimization, along with 3,500 lines of C++ code for the `FusedGraphOp` functionality. Prism rewrites

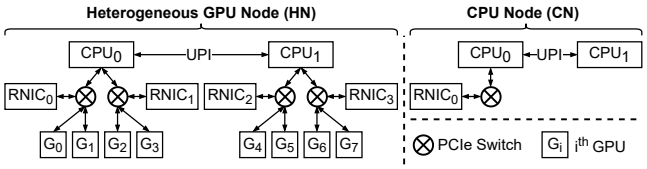


Figure 10: The hardware topology of a typical HN and CN.

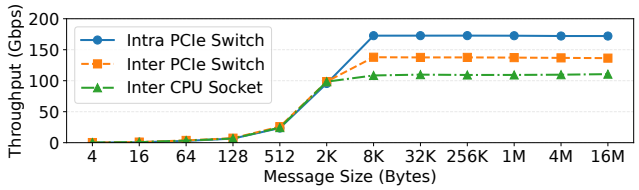


Figure 11: Performance comparison of GPUDirect RDMA across different GPU-RNIC interconnect topologies.

Algorithm 1: Topology-aware resource scheduling

Input: Cluster N, DLRM application id app, CN/HN instance $role$, resource requirements (cpu_num, gpu_num, mem_size, $disk\_size$), deployment density d
Output: Selected node $node\_id$, allocated resources (cpu_id, gpu_id, $rnic\_id$)

1 Initialize node set $S \leftarrow \varnothing$
2 $N_{pod} \leftarrow N$.filter($N.cluster\_topology, app$) ▷ Retain all nodes in the pod that runs the same app's existing instances
3 $N_{asw} \leftarrow$ get_asw_nodes(N_{pod}, app) ▷ Get nodes from ASWs with maximum instances
4 **parallel for** *node* $n \in N_{asw}$ **do**
5 **if** (*Insufficient resources* $\|$ $get\_density(n, app) \geq d$) **then**
6 Return ▷ Filter out unavailable nodes
7 `// Prefer the shortest CPU/GPU-to-RNIC path`
8 **if** $role$ = *HN* **then**
9 $(cpu\_id, gpu\_id, rnic\_id, topo\_score) \leftarrow$ calc($n.topology$, GPU, RNIC, gpu_num)
10 **else if** $role$ = *CN* **then**
11 $(cpu\_id, \varnothing, rnic\_id, topo\_score) \leftarrow$ calc($n.topology$, CPU, RNIC, cpu_num)
12 $S \leftarrow S \cup (n, cpu\_id, gpu\_id, rnic\_id, topo\_score)$
13 $node\_id$, cpu_id, gpu_id, $rnic\_id \leftarrow \arg\max_S topo\_score$

the user-submitted model graph, where the CN subgraph receives the original model inputs (e.g., user attributes, candidate item lists), and the HN subgraph receives tensors sent via `FusedGraphOp`. If a tensor requires transfer via the RPC library and subsequent operator consumption, Prism replicates a new operator and delegates control to the RPC library. We also provide support for the PyTorch framework [49]. The subsequent experiments (§5) focus on TensorFlow versions to evaluate overall system performance.

4.2 Topology-Aware Resource Manager

The RTP framework disaggregates the model onto CN and HN instances, with the resource manager selecting nodes and allocating resources. To ensure recommendation service SLOs, the resource manager adheres to *topology-aware* node scheduling and resource allocation principles (Algorithm 1).

Inter-node scheduling. The cluster consists of a fleet of CNs and HNs (Figure 6). These nodes are grouped in a cluster unit called a pod (point-of-delivery), where nodes have one or more dual-port high-performance RNICs. Intra-cluster communication is achieved through a classic two-tier clos network [50]. Prism follows two principles for scheduling all CN and HN instances of the same DLRM service. First, all instances are confined within the same pod (Algorithm 1, Line 2), as *cross-pod RDMA connections induce over 50% performance degradation* in our evaluation. Second, co-locating CN and HN instances within the same ASW (Access Switch) offers superior performance. Therefore, we prefer this affinity and endeavor to pack these instances under the same ASW. If instances are distributed across multiple ASWs, Prism schedules new instances to the ASW with the most existing instances (Algorithm 1, Line 3).

Intra-node resource allocation. The hardware topologies of a typical HN and CN are depicted in Figure 10. The interconnect between GPUs and RNICs has a substantial impact on performance. Figure 11 shows that arbitrary bindings of GPUs and RNICs can induce 21–36% performance loss. Therefore, the resource scheduler prefers to assign RNIC and GPU on the same PCIe switch for each HN instance, mitigating data movement. Regarding CPU and RNIC topology, Prism mainly considers CPU-intensive CN instances. CN typically has only one RNIC, so the scheduler prioritizes the allocation of CPUs under the same PCIe switch connected to the RNIC (Algorithm 1, Line 8–11).

Production deployment. To ensure fault tolerance, online inference services deploy instances across multiple data centers. Simultaneously, to mitigate the effects of traffic peaks, the resource scheduler limits the deployment density of DLRM instances (Algorithm 1, Line 5). This scheduling policy avoids bursty resource consumption on the same node. During peak seasonal traffic periods, the scheduler incorporates additional instance density constraints at the NIC switch layer. These constraints are derived from the aggregate bandwidth capacity of distributed switches and predetermined safety thresholds (e.g., 80%). The scheduler dynamically allocates bandwidth quotas for each instance based on these constraints, thereby preventing network saturation.

Implementation. We develop topology-aware scheduling capabilities as a scoring plugin integrated within Kubernetes [4]. The device plugin deployed on each node and the central scheduler each implemented approximately 1,000 lines of code in Golang. The device plugins on individual nodes manage heterogeneous hardware topologies, reporting available GPUs and virtualized RNICs to the scheduler.

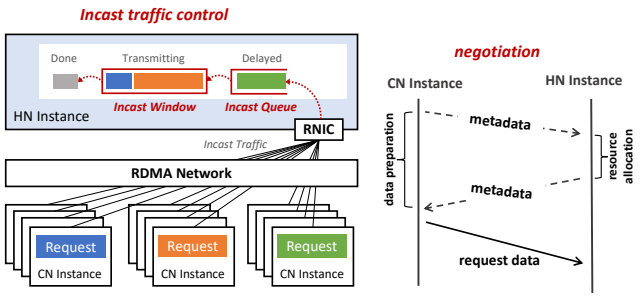


Figure 12: The incast control mechanism comprises of an incast window and queue. Only requests within the window can transmit parameters. The queue stores delayed requests.

4.3 SLO-Aware Communication Scheduler

Online recommendations demand stringent performance and stability guarantees. Empirical evaluations indicate that to adhere to stringent SLOs, the p50 and p99 latencies for transmitting 4 MiB model parameters must be below 3 ms and 6 ms, respectively. Notably, when a substantial number of CN instances concurrently transmit data to a limited number of HN instances, it generates a fan-in traffic pattern, also known as *incast*, as depicted in Figure 12. High-volume incast traffic can degrade overall performance, including long-tail latency and reduced throughput.

Incast flow control. As presented in Table 1, under various incast configurations, failing to impose restrictions on incast flows can result in request failures, which is unacceptable for a reliable online service. To meet the strict SLOs and mitigate the performance degradation caused by incast traffic, we propose an SLO-aware incast control mechanism. Incast control enhances communication performance by reducing the volume of incast traffic. As illustrated in Figure 12, incast control orchestrates the HN and CN instances to throttle incast traffic in a window-based manner. Initially, the CN instance sends a communication request before transmitting model data. Next, the HN instance verifies whether the incast window has the sufficient capacity to receive the model parameters. If space is available, the HN instance sends a communication response to initiate the data transmission. Otherwise, the HN instance defers processing the communication request in the incast queue until the incast window has the adequate capacity.

The *incast window size* and the *processing order of delayed requests* are crucial factors in determining the effectiveness of incast control in meeting SLOs, necessitating meticulous design. An undersized window leads to unnecessary delayed requests, while an oversized window renders incast control ineffective. Moreover, communication requests arrive with varying SLOs, and requests with more stringent SLOs might be impeded by other requests, resulting in SLO violations. We address these challenges through *adaptive incast window* and *SLO-aware communication request scheduling*.

Incast Size	Latency	% of Failed Requests
10	10 ms	≈ 33%
20	40 ms	≈ 50%
100	10 s	≈ 100%

Table 1: Incast statistics from our clusters. Incast size is the number of concurrent messages from different CN instances. Latency refers to the time for a HN instance to process 1,000 requests. The default message size is 8 MiB, typical for our services. Failed requests include timeouts or packet loss.

Algorithm 2: SLO-Aware Scheduling

Input: Newly-arrived communication request r' with its completion requirement $SLO_{r'}$ and message size $M_{r'}$, available network bandwidth B, requests in the incast queue R, current time t_{now}
Output: Scheduled communication requests R'

1 $d_{r'} \leftarrow t_{now} + SLO_{r'} - M_{r'}/B$ `/* calculate deadline */`
2 **foreach** r *in* R **do**
3 **if** $d_{r'} > d_r$ *and* r' *not in* R' **then**
4 R'.append(r')
5 R'.append(r)

Adaptive incast window. In Prism, the incast window size adapts to the congestion level of network links and PCIe links, as demonstrated in Figure 10. For PCIe link congestion, the traffic is stored in the RNIC receiver buffer. For network link congestion, the switch buffer stores the traffic. When the buffer occupancy of RNIC and switch surpasses a predefined threshold, the RNIC notifies the senders of congestion via Congestion Notification Packets (CNPs). Consequently, the number of CNPs serves as an estimator of the congestion level. The HN instance periodically collects the count of sent CNPs and calculate the congestion level by averaging CNP count by the number of f congestion requests. If the congestion level exceeds a predefined threshold T_{high}, the window sizes is increased. Conversely, if the congestion level falls below T_{low}, the window size is decreased. In Prism, the values of T_{high} and T_{low} are tunable based on the online workloads, preventing frequent fluctuations in the window size.

Deadline-aware request scheduling. Intuitively, Prism can process delayed requests held in the incast queue using a first-come-first-serve (FCFS) approach. Although straightforward to implement, FCFS is not SLO-friendly, as early-arriving requests with loose SLOs can obstruct requests that arrive later but have more stringent SLOs. To overcome this issue, the incast queue is designed to pop requests in a SLO-aware manner. Specifically, upon the arrival of a new communication request, Prism reorders the requests in the incast queue to maximize the number of requests meeting their SLOs. Based on this insight, we propose the early-deadline-first scheduling for delayed communication requests. The deadline of a communication request is defined as the latest time to initiate parameter transmission to meet the SLO. Prism calculates the deadline for the arrived request (Algorithm 2, Line 1) by

subtracting parameter transmission overhead ($M_{r'}/B$) from the sum of the current time and SLO. Then, r' is inserted after the requests with smaller deadline (Line 2–5).

Operational experience. Production clusters frequently employ a mixed deployment of online and offline tasks to improve efficiency. In our scenarios, the disaggregated DLRM serving introduces frequent RDMA network communication. We observe that even with exclusive access to the RNIC, RDMA transfer latency can experience a tenfold increase under intense resource contention. The root cause is that under the container overlay network [72], both RDMA and TCP rely on the overlay network scheme for communication. The TCP traffic from mixed workloads affects the flow table logic at the bottom layer of the network card, impacting RDMA traffic. Our current workaround involves monitoring node resource utilization and online service latency, triggering the eviction of offline tasks when the metrics become abnormal. We consider this as an open question for future research.

Implementation. We extend the native RoCEv2 [24] stack and implement an efficient middleware to fully leverage RDMA communication capabilities in a virtualized environment. The implementation consists of 40k lines of C++ code. Standard protocols [71] running in RNICs act as a black box for upper-level system components, which is not suitable for rapid testing and deployment. Thus, we implement an additional software-level control strategy to mitigate the negative impact of incast congestion. We design specialized meta-message packets for negotiation between senders and receivers to restrict the number of busy connections (Figure 12). This complementary software control mechanism is tuned with underlying congestion control protocols to ensure stable performance. To reduce overhead from registering memory regions for RDMA requests, Prism maintains a pre-allocated memory pool and several maps from buffer addresses to the related memory region objects. A memory allocator similar to slab is adopted to prevent buffer fragmentation.

5 Evaluation

In this section, we conduct extensive experiments to evaluate the performance impact and resource efficiency improvements of Prism. Our results demonstrate the following:

- Prism's disaggregated inference can maintain service performance under *high*-traffic scenarios (§5.2).
- Prism can effectively separate CPU and GPU computations, enhancing GPU efficiency in multi-GPU nodes by 5–9× (§5.3).
- Prism has been deployed in production clusters for over two years. In a daily GPU cluster with a high allocation rate (90%), it can reduce CPU fragmentation by 53% and GPU fragmentation by 27%. Additionally, during seasonal peak traffic, Prism can efficiently borrow training nodes to meet the increased demand, saving over 90% of GPUs (§5.4).

Model	Emb Size (Approximate)	RDMA TX (Per Req)	Dense Features
Model-XS	100 GiB	552.96 KiB	338.67 MiB
Model-S	450 GiB	6.84 MiB	57.20 MiB
Model-M	500 GiB	3.87 MiB	21.46 MiB
Model-L	600 GiB	3.69 MiB	20.79 MiB
Model-XL	700 GiB	9.03 MiB	8.73 GiB

Table 2: Models for evaluation.

5.1 Methodology

Production workloads. We use models from our industrial DLRM system, with test data originating from *real-world* user requests. Table 2 outlines the *high-level* characteristics of these models, including the size of their embedding tables, the communication volume per request transferred via RDMA after disaggregation, and memory space required for storing dense features. The models analyzed span a range of scales and use cases, processing *billions* of daily requests for tasks such as video click-through rate prediction, ad conversion rate prediction, and item ranking.

Machine specifications. The hardware topology of the HN and CN is illustrated in Figure 10. By default, model instances are deployed on HNs. Each HN contains 128 CPU cores, 8 A100 [5] GPUs with 80 GiB GPU memory each, and 4 RNICs with a 200 Gbps. After resource disaggregation using Prism, CN instances are deployed on CNs. Each CN is configured with 128 CPU cores and a single 200 Gbps RNIC. All nodes use Intel(R) Xeon(R) Platinum 8369B CPUs, with 1024 GiB memory, interconnected in a pod.

Baselines. To the best of our knowledge, Prism is the first GPU-disaggregated inference system for DLRMs. Our experiments feature three primary baselines for comparison: (i) *Baseline* refers to our highly optimized production DLRM inference system without resource disaggregation, as a comparison for performance overhead; (ii) *Local Disaggregation* splits the original DLRM instance, but the CN instance and HN instance co-locates on the same HN server; (iii) *Prism (Remote Disaggregation)* disaggregates the DLRM inference service by placing CN instance on the CN server and the HN instance on the HN server.

Metrics. We seek to evaluate the DLRM system's processing capabilities under peak traffic loads. To simulate such conditions, we initialize 30 workers to continuously send requests at a predefined frequency. Our key indicators are categorized into two groups:

Performance-level metrics: (i) *goodput*, defined as the number of requests that DLRM instances can process; (ii) *average latency*; and (iii) *p99 latency*.

Resource efficiency metrics: (iv) *CPU usage*: the overall CPU consumption of the DLRM service and the proportion of CPU usage on GPU nodes; (v) *GPU efficiency*: measured by the number of inference requests that can be processed on

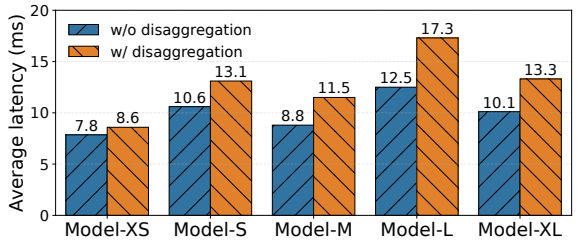


Figure 13: Latency of sequentially executing requests.

a single GPU node; and (vi) *resource fragments*: the amount of unallocated resources that are *expected* to be fragmented because they are too small to meet workload requirements or become stranded from other dimensional resources, quantified using the methodology proposed in [64].

5.2 Prism Performance: End-to-End Results

Overhead of serving a single request. First, we quantify the impact of model disaggregation on inference efficiency in the *worst-case* scenario. We send only *one* request at a time to the DLRM inference system for sequential execution, which indicates *low* resource utilization and throughput. Figure 13 records the average latency of sequentially processing requests in different models, showing a 10–38% increase. In this setup, the performance degradation of inference mainly stems from two aspects: (1) each request processing requires data transmission across the RDMA network, and (2) when executing requests sequentially, only one instance of Prism's CN instance or HN instance is computing at a time, whereas the baseline allows *asynchronous* executions of CPU and GPU operators on the same node.

Performance under varying traffic loads. Figure 14 shows the performance of various baselines under differing levels of requests per second (RPS). While Prism incurs minor overhead from cross-node communication during low RPS inference (e.g., p99 latency of Model-L at 200 RPS), it can *lower latency and increase goodput under higher RPS* compared to other baselines (e.g., p99 latency of Model-L at 800 RPS). Two main factors drive these improvements. First, modern GPUs' first-in-first-out scheduling can result in substantial queue head-of-line blocking [48], where at high RPS, concurrent requests *interleavingly* submit kernels to the GPU will drastically increase the latency. Prism mitigates this by leveraging `FusedGraphOp` to fuse each request's GPU operators, into a single sequential execution rather than interleaving across requests. Second, Prism decreases PCIe volume by converting host-to-device data transfers in DLRM computation to high-bandwidth device-to-device RDMA transfers. *Local disaggregation* also shows performance gains due to the graph optimizations mentioned above, but CPU contention between the CN instance and HN instance results in inferior performance compared to Prism. One special case is Model-XL, which has the largest communication volume (i.e., over 9 MiB) but very short GPU time ($<$ 1 ms, the lowest part of Figure 15). Even under this condition, Prism only suffers at most 6% performance loss. Overall, Prism is capable of maintaining service performance under *high* traffic scenarios in real-world deployments.

Latency breakdown. We perform additional breakdown analysis of elapsed time across stages, divided into CN computation time, data transmission latency via the RDMA network, and HN computation time. We discover that different DLRM inference services have distinct bottlenecks, which can be classified as *CPU-bounded*, *balanced*, or *GPU-bounded*. For example, the increased latency of Model-XS at high RPS is primarily attributed to the HN instance, indicating that the computational capacity (i.e., GPU) of the HN instance becomes the bottleneck. In contrast to the baseline, Prism decouples the resource usage of DLRM inference, allowing for *arbitrary* configuration of the number of CN and HN instances. This *flexibility* enables Prism to better accommodate the diverse resource requirements of different DLRM services (e.g., independently scale out HN instances to alleviate the computational bottleneck, in the case of GPU-bounded services like Model-XS, as shown in Figure 17).

5.3 Prism at Node: Resource Efficiency

Distinct resource separation. Figure 16 compares the CPU usage across different DLRM services before and after resource disaggregation. Prism maintains nearly identical CPU consumption when processing equivalent amounts of goodput, in comparison to the baseline. Notably, Prism effectively separates the computational resources of DLRM, resulting in a substantial reduction of CPU consumption on GPU nodes by 15–84×. These findings suggest that Prism possesses the capability to effectively mitigate GPU fragmentation issues that arise due to insufficient CPU resources.

Better resource efficiency on multi-GPU nodes. We deploy varying numbers of inference instances on a single HN to handle DLRM requests, limiting the service latency to 25 ms, and then compare the total goodput. Figure 17 demonstrates that Prism can deploy more DLRM instances on a single multi-GPU node, increasing inference throughput by 5–9×. Without disaggregation, the baseline can deploy a maximum of two Model-XS inference instances on the HN, fully saturating the CPUs while leaving 6 GPUs completely idle and *stranded*. For Model-XL, due to the memory limitations imposed by its massive embedding tables (i.e., 700 GiB), the baseline can deploy at most one instance. Prism supports flexible configuration of the ratio between CN and HN instances. For CPU-bounded Model-XL, only increasing the number of CN instances from 1 to 4 can yield a 36% improvement. For GPU-bounded Model-XS, if the MIG [8] feature

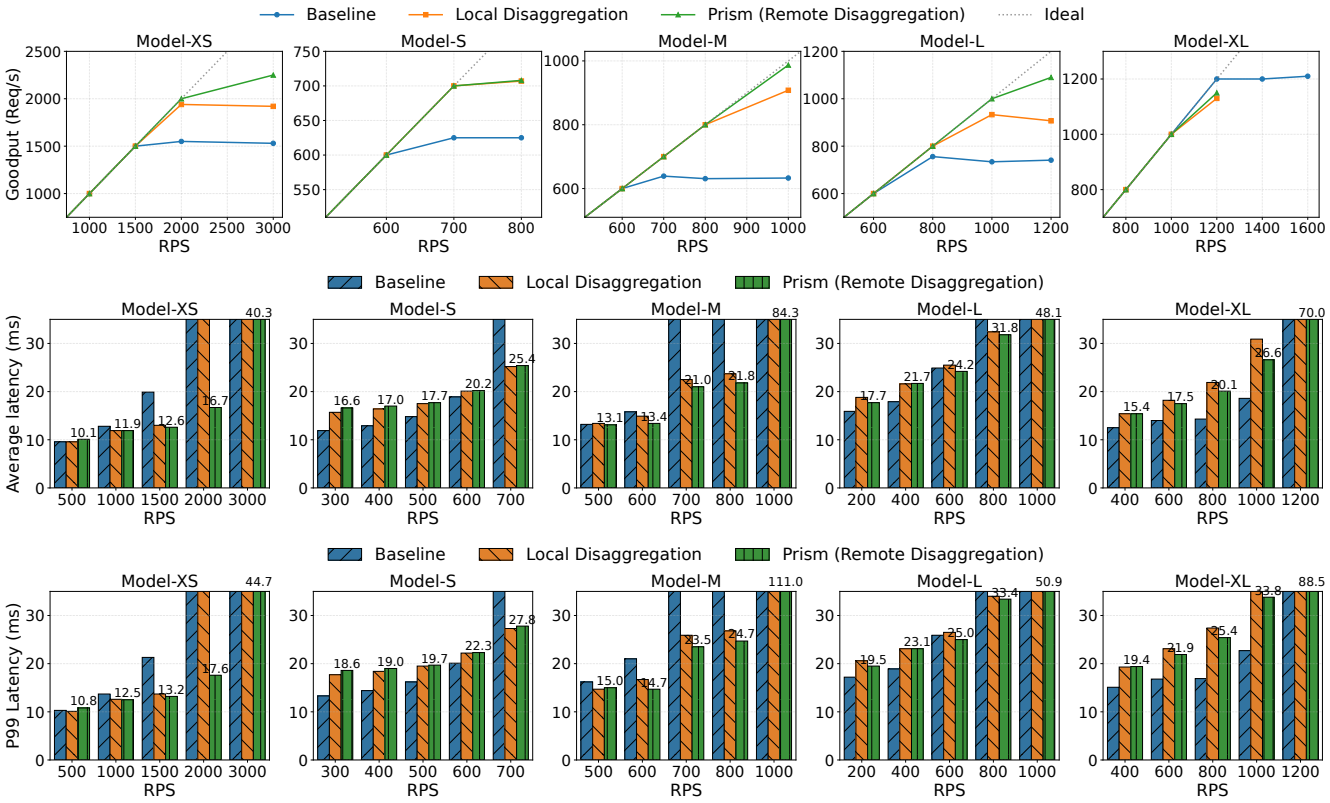


Figure 14: The end-to-end performance under varying traffic loads.

of A100 (i.e., fine-grained GPU allocation) is enabled, up to 24 HN instances can be deployed, boosting throughput to 9× compared to the baseline. This demonstrates that Prism's resource disaggregation allows for more efficient utilization of multi-GPU nodes customized for training workloads.

5.4 Prism at Scale: Production Deployment

Prism has been deployed in our production cluster for over two years, scaling to more than 10k GPUs to date. The system addresses two challenges: (1) resource allocation in GPU clusters with high allocation rates and (2) resource provisioning during seasonal traffic spikes. By leveraging a design that avoids costly infrastructure upgrades, Prism has enabled serving scale to grow continuously throughout this period.

Reduced resource fragmentation in GPU clusters. As discussed in [64], cluster resource fragmentation must consider the distribution of workloads. For tasks with different resource requirements, the cluster resource fragmentation from their perspective varies. Figure 18 compares the changes in resource requirements of DLRM instances before and after our deployment of Prism. Prism reduces the stringent resource requirements of the DLRM inference service. For HN instances that require GPU allocation, their CPU requests are less than 12 cores, and memory requests are below 24 GiB. In contrast, CN instances, which are CPU-intensive and need to load embedding tables, have CPU requests exceeding 48 cores and memory requests greater than 240 GiB. We simulate the deployment of DLRM instances before and after resource disaggregation to cluster H with a high allocation ratio. Figure 19 compares the changes in fragmented cluster resources. As Prism separates the resource requirements of DLRM instances, CN instances can run on nodes where GPUs are exhausted but CPUs remain, while HN instances can run on nodes with insufficient CPUs but available GPUs. Statistical analysis reveals that this approach significantly reduces the cluster's CPU fragments by 53% (18k cores) and GPU fragments by 27% (60 GPUs). These findings suggest that even in clusters with high allocation rates, the disaggregated DLRM service can efficiently utilize fragmented resources to meet deployment requirements.

Efficient resource loans for peak demand. During e-commerce promotional periods, we leverage Prism to borrow a portion of training nodes (equipped with A100 GPUs) to scale out DLRM inference services, meeting the *short-term* yet *high-throughput* traffic peaks. Table 3 outlines the resource requirements of three online services, with each HN instance requiring only one MIG GPU and minimal CPU resources (≤ 4), while the remaining CN instances handle the vast majority of CPU computations. These instances are

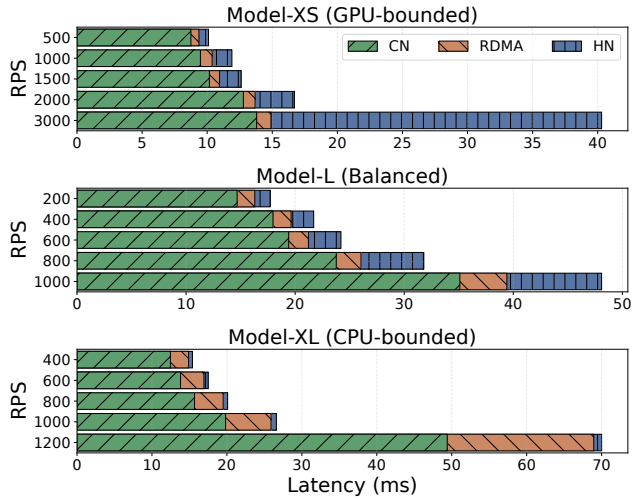


Figure 15: Latency breakdown of disaggregated DLRM inference services, which consists of CN computation time, RDMA transfer latency, and HN computation time. Model-XS, Model-S, and Model-M demonstrate consistent trends.

Figure 16: Comparison of CPU utilization. Prism (total) represents the total CPU consumption of the CN instance and HN instance in the disaggregated DLRM inference service, while Prism (HN) denotes the CPU consumed by the HN instance. The x-axis labels include the goodput in parentheses.

distributed across different nodes to ensure high availability. Figure 20 illustrates the QPS and latency of the three services during seasonal promotions, with the peak occurring at the 20th hour. Latency for the online service was not detrimentally impacted—overall latency remained within 25 ms and no request exceeded 35 ms, demonstrating the ability in handling production-level traffic loads. A rough estimation reveals that only 6 A100 nodes are required to satisfy GPU requirements of these services, whereas previously, an A100 node could deploy at most 2 inference instances, requiring up to 70 A100 nodes to meet the same demand. By decoupling resource requirements, Prism enables DLRM services to more efficiently borrow multi-GPU training nodes during promotional periods, saving over 90% of GPUs.

6 Discussion and Related Work

DLRM systems. Existing DLRM systems primarily focus on scaling embedding table capacity for enhanced model accuracy [34,36,46], optimizing embedding lookup operations for accelerated inference [55,59,63,65,66], maximizing resource efficiency during training [60,61], developing embedding table pruning techniques [37], and expediting model parameter updates [54]. Only a few works study DLRM provisioning at the datacenter scale. Hercules [33], developed for monolithic servers, efficiently searches the task scheduling space and dynamically provisions the best-matching heterogeneous resources in the presence of diurnally changing load. To our knowledge, DisaggRec [34] is the only work that advocates resource disaggregation for large-scale DLRM serving. Unlike Prism, DisaggRec is a memory-disaggregated system that decouples the deployment of compute and memory, with the aim of addressing the growing memory demands of large-scale DLRMs. DisaggRec is only a prototype evaluated in an emulated memory-disaggregated infrastructure.

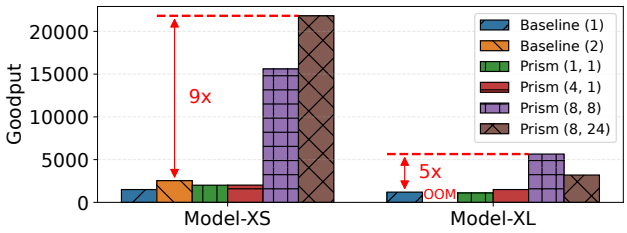


Figure 17: Comparison of HN GPU efficiency, measured by the total goodput. Baseline (count) represents the number of monolithic DLRM instances deployed on the HN, while Prism (CN count, HN count) denotes the number of CN instances and HN instances, respectively.

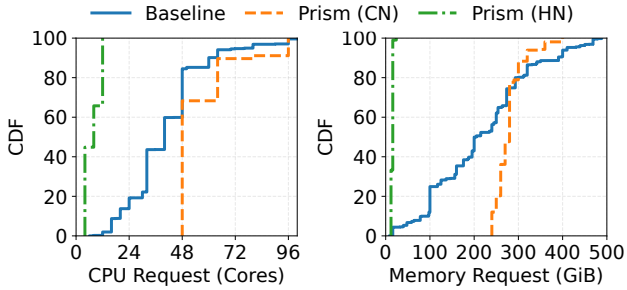


Figure 18: Comparison of resource requirements. Baseline, Prism (CN), and Prism (HN) denote the original monolithic DLRM instances, the disaggregated CN instances, and the disaggregated HN instances respectively. Both the baseline and Prism (HN) instances are allocated one GPU.

Colocation of training and inference workloads. Modern cluster management systems aspire to create a *unified* infrastructure that seamlessly integrate training and inference workloads, optimizing resource multiplexing while minimizing fragmentation. Specifically, Lyra [39] advances this paradigm by leveraging elastic training mechanisms to repurpose low-load inference servers. In this work, Prism reveals fundamen-

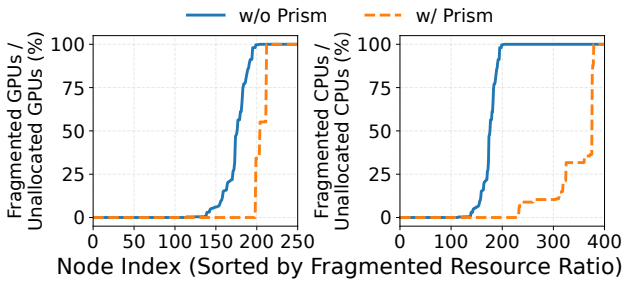


Figure 19: The change in resource fragments of a large-scale GPU cluster *H* before and after resource disaggregation. The original node residual resources are depicted in Figure 3.

Service	Role	# of Instances	CPU	GPU
Product-A	CN	25	48	-
	HN	45	4	MIG 2g.20gb
Product-B	CN	15	48	-
	HN	40	4	MIG 2g.20gb
Product-C	CN	15	48	-
	HN	55	2	MIG 2g.20gb

Table 3: The resource specifications for three online recommendation services. Each GPU mentioned here corresponds to one GPU instance on A100 [5], which includes two GPU compute slices and 20 GiB of GPU memory.

tal inefficiencies in repurposing training clusters for DLRM inference workloads, eliminating resource mismatches through resource disaggregation.

Resource disaggregation. Resource disaggregation holds tremendous promise in datacenters, with many prototype implementations demonstrating benefits of independent scaling of compute and memory resources, improved reliability, and cost-efficient hardware deployment. Existing works span multiple domains, encompassing API and framework innovations [15, 51, 70], operating system and network architectures [18, 53], and hardware design [25, 29, 38]. Prism specifically targets DLRMs, which are characterized by their substantial memory footprints and intensive CPU usage. By disaggregating CPU and GPU provisioning, Prism optimizes resource utilization in heterogeneous GPU clusters. This approach can potentially generalize to other models exhibiting distinct resource usage patterns, including graph neural networks [52] and large language models with retrieval augmented generation [20].

Model parallelism. The idea of model parallelism has been long introduced [35], yet the partitioning of computation graphs across available devices continues to evolve for different model architectures [17, 31, 44, 45, 56]. Prism focuses on DLRMs, which are distinct in their large embedding tables [36, 37]. Moreover, we aim to optimize the inference rather than the training, thus more concerned with the per-request latency rather than long-term throughput.

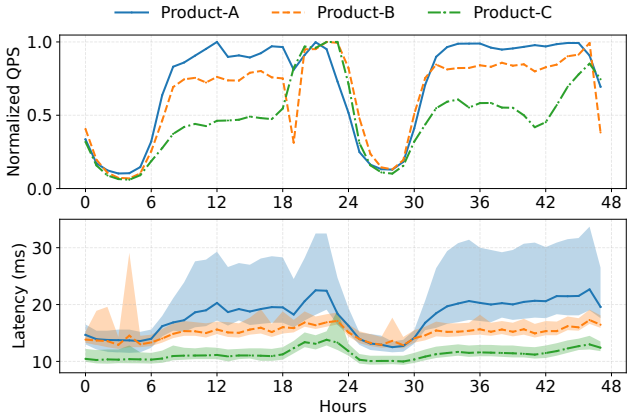


Figure 20: The statistics of three production services (Table 3) during a tremendous traffic promotional period.

Communication optimization with RDMA. RDMA technology has revolutionized datacenter applications across domains, including remote storage systems [10, 19, 43, 62], distributed training frameworks [6, 7, 14], and cluster monitoring solutions [41]. Specifically, previous work [24, 40, 71] attempts to implement high performance and stable RDMA communications from the protocol level. Prism adopts software-level flow control and scheduling strategies, which are more customizable for online services. Prism is complementary to the above RDMA-optimizing techniques.

7 Conclusion

In this work, we propose Prism, the first GPU-disaggregated DLRM serving system to efficiently provision resources at scale. Prism divides DLRMs into CPU- and GPU- intensive subgraph and offloads them to CPU and GPU servers. Prism employs various techniques to minimize the latency overhead, including optimal graph partitioning, topology-aware resource scheduling, and RDMA network optimization. Experimental results demonstrate that Prism effectively separates CPU and GPU computations while maintaining service performance. Prism effectively reduces CPU fragmentation by 53% and GPU fragmentation by 27% in a GPU cluster with a high allocation rate. During seasonal promotional events in e-commerce platforms, Prism can efficiently borrow GPU servers from training clusters to meet peak traffic demands, resulting in up to 90% of GPU savings.

Acknowledgement

We thank our shepherd Seo Jin Park and anonymous reviewers for their valuable comments. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC GRF Grants 16217124 and 16210822, RGC CRF Grants C6015-23G and C7004-22G, and NSFC/RGC Collaborative Research Scheme under the contract of CRS_HKUST601/24.

References

[1] Compute Express Link 3.0. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf, 2023.

[2] VMware vSphere Bitfusion. https://docs.vmware.com/en/VMware-vSphere-Bitfusion/index.html, 2023.

[3] Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2025.

[4] Kubernetes. https://kubernetes.io/, 2025.

[5] NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/, 2025.

[6] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl, 2025.

[7] NVIDIA GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2025.

[8] NVIDIA Multi-Instance GPU (MIG). https://docs.nvidia.com/datacenter/tesla/mig-user-guide, 2025.

[9] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[10] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure storage with RDMA. In *Proc. USENIX NSDI*, 2023.

[11] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for E-commerce recommendation in Alibaba. In *Proc. ACM DLP-KDD*, 2019.

[12] 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. Wide & deep learning for recommender systems. *arXiv preprint arXiv:1606.07792*, 2016.

[13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56:74–80, 2013.

[14] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, Shanyuan Gao, Xin Long, Jie Zhang, Yong Li, Zhisheng Xia, Liuyihan Song, Yingya Zhang, Pan Pan, Guohui Wang, and Xiaowei Jiang. ACCL: Architecting highly scalable distributed training systems with highly-efficient collective communication library. *IEEE Micro*, 41(5):85–92, 2021.

[15] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. HPCS*, 2010.

[16] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. DGSF: Disaggregated GPUs for serverless functions. In *Proc. IEEE IPDPS*, 2022.

[17] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proc. USENIX OSDI*, 2021.

[18] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proc. USENIX OSDI*, 2016.

[19] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *Proc. USENIX NSDI*, 2021.

[20] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2024.

[21] Carlos A. Gomez-Uribe and Neil Hunt. The Netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manag. Inf. Syst.*, 6(4), 2016.

[22] Alan Gray. Getting started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with Infiniswap. In *Proc. USENIX NSDI*, 2017.

[24] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proc. ACM SIGCOMM*, 2016.

[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proc. ACM ASPLOS*, 2022.

[26] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's DNN-based personalized recommendation. In *Proc. IEEE HPCA*, 2020.

[27] Walid A. Hanafy, Limin Wang, Hyunseok Chang, Sarit Mukherjee, T. V. Lakshman, and Prashant Shenoy. Understanding the benefits of hardware-accelerated communication in model-serving applications. In *Proc. IEEE/ACM IWQoS*, 2023.

[28] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA*, 2018.

[29] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, Qiang Liu, Dennis Cai, and Xiantao Zhang. DxPU: Large-scale disaggregated GPU pools in the datacenter. *ACM Trans. Archit. Code Optim.*, 20(4), 2023.

[30] 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. Optimizing CPU performance for recommendation systems at-scale. In *Proc. ACM/IEEE ISCA*, 2023.

[31] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proc. MLSys*, 2019.

[32] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim 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. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proc. ACM/IEEE ISCA*, 2020.

[33] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *Proc. IEEE HPCA*, 2022.

[34] Liu Ke, Xuan Zhang, Benjamin Lee, G. Edward Suh, and Hsien-Hsin S. Lee. DisaggRec: Architecting disaggregated systems for large-scale personalized recommendation. *arXiv preprint arXiv:2212.00939*, 2022.

[35] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[36] Daniar H. Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. EVStore: Storage and caching capabilities for scaling embedding tables in deep recommendation systems. In *Proc. ACM ASPLOS*, 2023.

[37] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, Chun-cheng Jason Chen, and Mosharaf Chowdhury. AdaEmbed: Adaptive embedding for large-scale recommendation models. In *Proc. USENIX OSDI*, 2023.

[38] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proc. ACM ASPLOS*, 2023.

[39] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. ACM EuroSys*, 2023.

[40] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *Proc. ACM SIGCOMM*, 2019.

[41] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *Proc. USENIX NSDI*, 2023.

[42] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. SmartIO: Zero-overhead device sharing through PCIe networking. *ACM Trans. Comput. Syst.*, 38(1–2), 2021.

[43] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proc. ACM SIGCOMM*, 2022.

[44] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *Proc. ICLR*, 2018.

[45] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proc. ICML*, 2017.

[46] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proc. ACM/IEEE ISCA*, 2022.

[47] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[48] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined GPU scheduling. In *Proc. ACM SOSP*, 2023.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.

[50] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proc. ACM SIGCOMM*, 2022.

[51] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proc. USENIX OSDI*, 2020.

[52] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Netw. Learn. Syst.*, 20(1):61–80, 2009.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.

[54] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In *Proc. USENIX OSDI*, 2022.

[55] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. UGache: A unified GPU cache for embedding-based deep learning. In *Proc. ACM SOSP*, 2023.

[56] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu.

Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *Proc. USENIX OSDI*, 2021.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017.

[58] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In *Proc. ACM KDD*, 2018.

[59] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. Merlin HugeCTR: GPU-accelerated recommender system training and inference. In *Proc. ACM RecSys*, 2022.

[60] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. RAP: Resource-aware automated GPU sharing for multi-GPU recommendation model training and input preprocessing. In *Proc. ACM ASPLOS*, 2024.

[61] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. EL-Rec: Efficient large-scale recommendation model training via tensor-train embedding table. In *Proc. ACM/IEEE SC*, 2022.

[62] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *Proc. USENIX OSDI*, 2020.

[63] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A GPU-specialized inference parameter server for large-scale deep recommendation models. In *Proc. ACM RecSys*, 2022.

[64] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-sharing workloads with fragmentation gradient descent. In *Proc. USENIX ATC*, 2023.

[65] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient GPU embedding cache for personalized recommendations. In *Proc. ACM EuroSys*, 2022.

[66] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex Bronstein, Ronald Dreslinski, Trevor Mudge, and Nishil Talati. GRACE: A scalable graph-based approach to accelerating recommendation model inference. In *Proc. ACM ASPLOS*, 2023.

[67] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using RDMA. In *Proc. ACM SIGCOMM Posters and Demos*, 2017.

[68] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. Workload consolidation in Alibaba clusters: The good, the bad, and the ugly. In *Proc. ACM SoCC*, 2022.

[69] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM KDD*, 2018.

[70] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *Proc. USENIX OSDI*, 2022.

[71] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proc. ACM SIGCOMM*, 2015.

[72] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *Proc. USENIX NSDI*, 2019.