DynamoServe: A Distributed Tiered Memory System for Multi-tenant LLM Serving

Diman Zad Tootaghaj[1], *Khaled Diab*[1], *Bob Lantz*[1], *Hanjiang Wu*[2], *K. K. Ramakrishnan*[4],
Md Ashfaqur Rahaman[3], Ryan Stutsman[3], Puneet Sharma[1], Tushar Krishna[2]
[1] HPE Labs, [2] Georgia Tech, [3] University of Utah, [4] UC Riverside

Abstract

The rapid adoption of large language models (LLMs) has increased the need for efficient multi-tenant inference systems that maximize GPU utilization. However, existing frameworks struggle to scale due to the high memory demands of model weights and key-value (KV) caches. We present *DynamoServe*, a multi-tenant LLM serving framework that addresses these challenges through three key innovations: (1) leveraging stranded GPU memory to offload model weights and KV caches, (2) mitigating resource fragmentation in multi-workload environments, and (3) improving memory locality through coordinated data placement and demand-driven weight migration across GPUs. Together, these techniques enable high-throughput, low-latency inference. Experiments on state-of-the-art models show that *DynamoServe* significantly improves memory efficiency without sacrificing latency.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Planning and scheduling**.

Keywords

Large Language Models, Machine learning workload, Memory Management

ACM Reference Format:
Diman Zad Tootaghaj[1], *Khaled Diab*[1], *Bob Lantz*[1], *Hanjiang Wu*[2], *K. K. Ramakrishnan*[4],, Md Ashfaqur Rahaman[3], Ryan Stutsman[3], Puneet Sharma[1], Tushar Krishna[2], [1] HPE Labs, [2] Georgia Tech, [3] University of Utah, [4] UC Riverside . 2026. *DynamoServe*: A Distributed Tiered Memory System for Multi-tenant LLM Serving. In *1st Workshop on Memory, Systems and Interconnect Co-design for AI (MOSAIC '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3789240.3829347

1 Introduction

Large language models (LLMs) continue to scale rapidly in size, driving significant improvements in performance and reasoning [4, 5]. However, the memory footprint of LLM inference comprising model weights, KV caches, and intermediate state often exceeds the high-bandwidth memory (HBM) capacity of a single GPU. Existing approaches such as model parallelism statically partition model state across devices but fail to adapt to dynamic workload demands. Alternatively, offloading to host memory or NVMe increases capacity but introduces high latency and bandwidth bottlenecks, degrading inference performance.

These challenges are amplified in modern multi-tenant GPU clusters, where heterogeneous workloads with asymmetric resource demands must coexist. Compute-intensive CNN workloads often underutilize GPU memory, while memory-intensive LLM workloads are constrained by capacity. As a result, clusters exhibit significant *stranded GPU memory*, where unused HBM on some devices coexists with memory pressure on others [1, 15, 16]. Current systems largely rely on static allocation or device-local memory management. Although recent approaches support selective memory sharing or offloading, efficiently leveraging unused GPU memory across devices remains challenging, often leading to underutilized capacity and memory bottlenecks.

Recent efforts have explored memory disaggregation and tiered memory for ML workloads. FlexGen increases effective capacity by offloading LLM state to CPU memory or NVMe, but suffers from high data movement overheads due to limited interconnect bandwidth [12]. Aqua supports peer-GPU memory offloading within a node, but expands memory through a single designated peer GPU rather than a distributed memory pool spanning multiple devices [13]. Consequently, available HBM capacity across the cluster may not be fully utilized. These limitations motivate systems that can dynamically aggregate and manage memory across many GPUs while preserving low-latency access.

Modern GPU clusters increasingly provide high-speed interconnects, ranging from intra-node fabrics such as NVLink to inter-node technologies such as RDMA. In this work, we focus on leveraging unused HBM accessible through NVLink-connected GPUs, which offers substantially lower access latency than conventional CPU-memory offloading. However, effectively utilizing this resource requires addressing several challenges: (i) how to aggregate and allocate stranded memory across GPUs, (ii) how to decide which LLM state to place on remote GPUs versus slower memory tiers, and (iii) how to account for heterogeneous interconnect costs when moving data across devices.

This paper argues that LLM inference systems should treat GPU memory as a cluster-wide resource rather than a device-local constraint. By dynamically pooling memory across GPUs, a serving system can decouple memory capacity from compute placement and improve utilization without sacrificing performance.

To this end, we present ***DynamoServe***, a multi-tenant LLM serving system that enables opportunistic GPU memory pooling across peer devices. *DynamoServe* transparently offloads LLM state, including model weights and KV caches, to unused HBM on peer GPUs using high-bandwidth interconnects. It incorporates interconnect-aware placement and migration policies to minimize data movement overheads while adapting to workload dynamics.

MOSAIC '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829347

In addition, *DynamoServe* supports fine-grained co-location of compute-intensive CNN workloads and memory-intensive LLM workloads using GPU sharing mechanisms such as NVIDIA MPS, enabling simultaneous utilization of compute and memory resources.

Contributions. This paper makes the following contributions:

- We identify *stranded GPU memory* as a key inefficiency in multi-tenant ML clusters with heterogeneous workloads.
- We design and implement *DynamoServe*, a system that dynamically pools GPU memory across devices and offloads LLM state to peer GPUs using high-bandwidth interconnects.
- We develop interconnect-aware placement and migration strategies that account for heterogeneous data transfer costs.
- We evaluate *DynamoServe* on representative multi-tenant workloads and demonstrate improved memory utilization and LLM inference performance over existing approaches.

2 Background and Motivation

LLM inference is fundamentally memory-bound. Model weights can occupy tens of GBs, while KV-cache grows linearly with sequence length and request concurrency, quickly exhausting GPU HBM capacity. For example, KV-cache storage can reach tens of GBs for long-context inference, as shown in Figure 1 using the Mooncake trace [10]. Even on modern GPUs such as the NVIDIA H200 (144GB HBM), we observe sustained near-100% memory utilization under realistic workloads, and in higher-load settings, requests are rejected due to insufficient KV-cache capacity.

Table 1 further highlights that even moderately sized models (e.g., OPT-13B) require tens of GBs for weights, with KV-cache introducing substantial additional overhead at longer context lengths. As a result, LLM inference systems frequently operate near memory capacity, leading to queuing delays, batching inefficiencies, or request rejection under load.

Existing systems address these constraints using model parallelism or offloading to host memory. Model parallelism introduces significant communication overhead and reduces flexibility in multi-tenant environments, while CPU/NVMe offloading is constrained by lower bandwidth and higher latency [6, 12]. These approaches are particularly inefficient in multi-tenant GPU clusters, where multiple models compete for shared memory and compute resources.

This motivates a design point where models that exceed single-GPU memory capacity can still be executed efficiently by leveraging unused high-bandwidth memory across peer GPUs within the same NVLink-connected domain, reducing reliance on both model parallelism and slow host-based offloading.

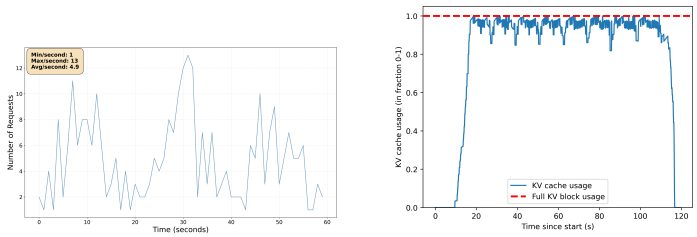
2.1 Leveraging Stranded GPU Memory

Multi-tenant GPU clusters exhibit significant *memory fragmentation*, where memory pressure and underutilization coexist across devices [1, 15, 16]. While this fragmented capacity exists in modern systems, it is not exposed as a first-class abstraction in current LLM serving frameworks.

DynamoServe leverages this opportunity by using unused high-bandwidth memory (HBM) on peer GPUs within the same NVLink-connected server as a transient extension of GPU memory for LLM inference. This approach prioritizes intra-node GPU interconnects, which offer substantially lower latency and higher bandwidth than alternative memory tiers such as CPU DRAM, NVMe, or scale-out RDMA-based access.

Model	Weights (GB)	KV@2K	KV@4K	KV@8K
OPT-6.7B	13.4	0.58	1.15	2.30
OPT-13B	26	1.10	2.20	4.40
OPT-30B	60	2.64	5.28	10.56

Table 1: LLM memory footprint (weights + KV-cache growth).



(a) Input trace distribution **(b) KV cache usage**

Figure 1: KV-cache pressure under bursty workloads [10].

Within this scope, *DynamoServe* dynamically aggregates available peer-GPU HBM and offloads LLM state, including model weights and KV caches, across GPUs. It employs coordinated placement and migration policies to adapt to workload dynamics and interconnect topology, enabling efficient utilization of otherwise stranded memory resources.

2.2 Modern multi-tenant LLM serving systems

3 *DynamoServe* Design

DynamoServe executes LLM inference on a designated GPU while dynamically offloading portions of model state to unused HBM on peer GPUs within the same NVLink-connected domain. This design expands effective memory capacity without immediately resorting to model parallelism, thereby decoupling memory availability from compute placement while keeping execution local to a single GPU. Compared to CPU or NVMe offloading, peer GPU memory offers significantly lower latency and higher bandwidth, enabling efficient execution of models that exceed a single GPU's memory capacity while remaining within the aggregated intra-node HBM budget. However, this design is not intended to replace model parallelism entirely. Instead, *DynamoServe* operates in a regime where memory expansion is sufficient to avoid or delay model parallel execution, which typically introduces additional communication and compute overhead. When memory demands exceed available peer-GPU capacity, model parallelism remains a necessary fallback mechanism.

The primary objective of *DynamoServe* is to minimize end-to-end inference latency under SLO constraints while improving cluster-wide GPU memory utilization. By prioritizing memory expansion over model parallelism when feasible, *DynamoServe* preserves compute resources on other GPUs for concurrent workloads, making it well-suited for multi-tenant environments.

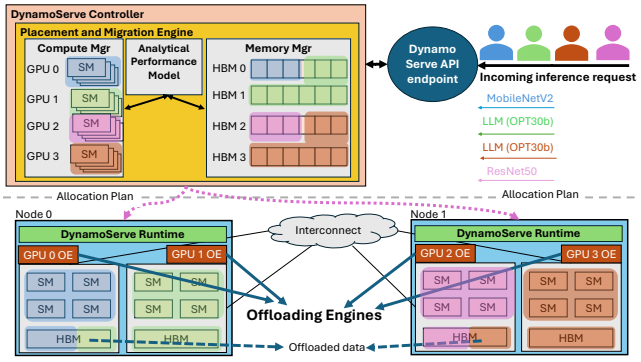


Figure 2: System Architecture.

Design Challenges. Key challenges include: (i) limited local memory due to KV-cache growth, (ii) transfer overheads for remote memory access, (iii) heterogeneous interconnect costs (e.g., NVLink vs. PCIe), and (iv) interference among co-located workloads. These challenges motivate a controller-driven design that jointly manages memory placement and compute allocation.

3.1 DynamoServe Architecture

DynamoServe consists of (1) a centralized *controller* responsible for global decisions, and (2) distributed *offloading engines* (OE) on each GPU that execute these decisions.

Controller. The controller maintains a cluster-wide view of GPU memory capacity, compute utilization, and interconnect topology. It identifies opportunities for leveraging underutilized GPU memory when available, but does not assume the existence of stranded memory or heterogeneous workload imbalance. When such opportunities arise, the controller determines placement of LLM state across memory tiers (local GPU, peer GPU, CPU, and NVMe) to minimize latency while satisfying SLO constraints.

Upon memory pressure, the controller evaluates placement options and selects targets based on available capacity and estimated data movement cost. These decisions are made off the critical path and are periodically updated as workload and resource conditions evolve.

Offloading Engine. Each GPU hosts an OE that manages local memory and data movement. Upon memory pressure (e.g., KV cache growth), the OE reports demand to the controller and executes placement decisions via direct GPU-to-GPU transfers. Transfers prioritize high-bandwidth links (e.g., NVLink) and avoid CPU involvement.

A key mechanism is *demand-driven weight migration.* The OE tracks layer execution order and access patterns to determine which weights should reside locally versus remotely. Frequently accessed layers are kept close to the executing GPU, while less critical states are evicted to peer GPUs with available capacity. Data transfers are performed asynchronously using separate CUDA streams to overlap communication with computation, reducing stall time.

By keeping data movement within GPUs and avoiding host or cross-node communication, DynamoServe reduces latency compared to CPU- or NVMe-based offloading and efficiently utilizes available intra-node GPU memory capacity across devices.

3.2 Memory Management Policy

The controller determines how to place model state across heterogeneous memory tiers to minimize latency while satisfying SLO constraints.

We consider a stream of inference requests R, which are processed online as they arrive. At each decision epoch, the controller uses the set of active requests within the current scheduling window, along with system state, to determine memory placement decisions. We define the memory tiers $M = \{\text{local GPU, peer GPU, CPU, NVMe}\}$.

This optimization is solved periodically over a sliding scheduling window t, and decisions are updated as requests arrive and complete. The system does not assume knowledge of future requests, and no global re-optimization or request migration across completed placements is performed beyond the active window.

Each request $r \in R$ consists of model weights (W_r), KV cache (K_r), and activations (A_r), with corresponding sizes S_r^W, S_r^K, and S_r^A.

We define decision variables:

$$x_{r,m}^c \in [0,1],$$

representing the fraction of component $c \in \{W, K, A\}$ for request r placed in memory tier m, subject to:

$$\sum_{m \in M} x_{r,m}^c = 1 \quad \forall r, c.$$

Each memory tier has capacity constraints:

$$\sum_{r \in R_t} \sum_{c \in \{W,K,A\}} x_{r,m}^c \cdot S_r^c \le C_m \quad \forall m \in M.$$

where R_t denotes the set of active requests observed within a scheduling window t, reflecting the online nature of the system.

We model transfer cost using a latency–bandwidth formulation:

$$TransferCost_r = \sum_{c \in \{W,K,A\}} \sum_{m \in M} x_{r,m}^c \cdot f_r^c \cdot \left(D_m + \frac{S_r^c}{B_m} \right),$$

where D_m and B_m denote the latency and bandwidth of memory tier m, and f_r^c is the observed access frequency of component c during execution.

We estimate compute time $Compute_r$ using either (i) offline profiling of the model under representative workloads or (ii) runtime measurements from prior executions of the same model under similar batch configurations.

Total latency is:

$$Latency_r = Compute_r + TransferCost_r,$$

and must satisfy:

$$Latency_r \le SLO_r \quad \forall r.$$

The controller minimizes aggregate latency:

$$\min \sum_{r \in R} w_r \cdot Latency_r.$$

where w_r is a priority or fairness weight assigned to request r. This optimization is solved periodically over a sliding scheduling window t, and decisions are updated as requests arrive and complete. The system does not assume knowledge of future requests, and no global re-optimization or request migration across completed placements is performed beyond the active window.

3.3 Allocation of Compute with Co-location

In addition to memory placement, *DynamoServe* allocates GPU compute across concurrent workloads using mechanisms such as NVIDIA MPS. Each workload is assigned a fraction of GPU compute, enabling safe co-location.

Rather than solving a complex optimization online, *DynamoServe* uses profiling-guided allocation. Specifically, we observe that each model exhibits a saturation point beyond which additional compute yields diminishing improvements in throughput or latency. This saturation point is estimated either through offline profiling or via short online calibration runs when a model is first deployed. *DynamoServe* allocates GPU compute up to this estimated saturation point, avoiding over-provisioning while enabling efficient sharing of GPU resources across workloads. This allocation is periodically updated as workload characteristics evolve. This approach is particularly effective for CNN–LLM co-location: CNNs are compute-intensive, while LLMs are primarily memory-bound. By allocating compute based on observed scaling behavior, *DynamoServe* allows both workloads to operate near peak efficiency while fully utilizing GPU resources.

4 Evaluation

To evaluate the performance of our proposed approach, we developed a prototype and deployed it on a cluster consisting of eight V100 GPUs. For our experiments, we use the OPT-30B model in a multi-GPU setup. The system configuration is as follows: (1) **CPU**: 2 x Intel(R) Xeon(R) Gold 6248 CPUs @ 2.50GHz, (2) **Memory**: 768 GB RAM, (3) **GPUs**: 8 x NVIDIA V100-SXM2-32GB (32GB HBM2) GPUs connected via NVLinks (both NV1 and NV2) shown in Figure 3. The offloading engine of *DynamoServe* is implemented both on top of the FlexGen [12] and vLLM [14] inference serving engines. While FlexGen and vLLM enable the swapping of weights and token states to CPU DRAM during LLM inference when GPU memory is a constraint, it is limited to a single-node setup and lacks dynamic memory management. To address this gap, we extended FlexGen and vLLM by integrating the ability to offload data to peer GPU memory on the same node.

For our experimental evaluation, the resource allocation and offloading decisions are derived analytically and implemented manually rather than using a general-purpose LP solver. This design choice is intentional, as our testbed is small and fixed-scale, allowing us to focus on system behavior and performance impacts without introducing solver overhead. Scaling the formulation to larger clusters and automated solvers is left for future work.

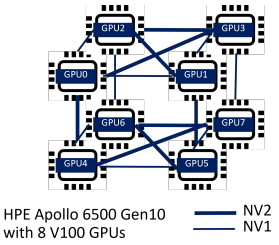


Figure 3: System testbed.

4.1 Profiling CNN Scaling Behavior under MPS

In this set of experiments, we profile the compute scaling behavior of representative workloads under NVIDIA MPS to identify their saturation points. We first consider three widely used CNN models spanning a range of computational intensities: MobileNetV2 (small), ResNet-50 (medium), and ResNet-152 (large). For each model, we vary the MPS active thread percentage from 10% to 100% and measure steady-state inference throughput. Figure 4 shows that different CNN models exhibit distinct scaling behaviors. MobileNetV2 saturates early, achieving near-peak throughput at approximately 30% of SMs, whereas ResNet-50 continues to scale up to around 60% SM allocation, and ResNet-152 requires roughly 70–80% of SMs to approach saturation. Beyond these knee points, additional compute provides diminishing returns, highlighting opportunities for safe co-location. We extend this analysis to large language models by profiling the OPT-30B model under MPS. Figure 5 shows the knee points for both latency and throughput. Similarly to CNNs, OPT-30B also shows clear saturation points where allocating SMs beyond a model's knee point leads to wasted compute capacity that can be reclaimed for co-located workloads without significantly impacting CNN performance. *DynamoServe* leverages this observation by using profiling-derived saturation points as *recommended compute budgets* for each workload class. Specifically, each workload is assigned a model-specific SM budget corresponding to its knee point, which serves as a target operating point rather than a strict upper bound. This budget is used in the compute allocation policy to avoid over-provisioning compute to workloads already operating in the saturated regime, while preserving headroom for co-located workloads. Importantly, workloads are not forced to operate beyond their knee point; instead, allocations beyond this point are considered low marginal utility and are deprioritized in favor of improving overall GPU utilization and co-location efficiency.

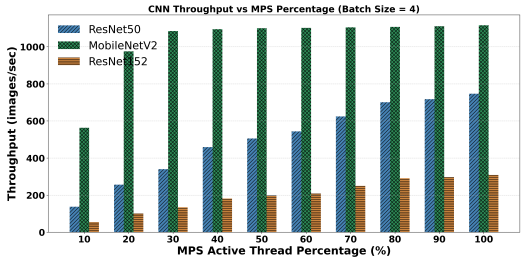
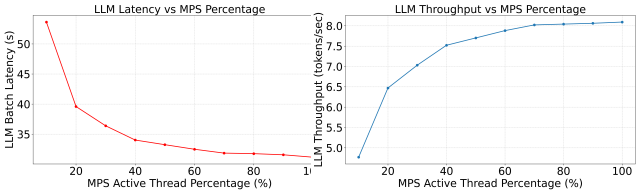


Figure 4: Throughput scaling and knee points for CNN inference workloads under MPS.



(a) Latency. **(b) Throughput.**

Figure 5: Knee points for latency and throughput for the OPT-30B LLM under MPS.

4.2 LLM-CNN Co-location

We evaluate *DynamoServe* 's ability to co-locate compute-intensive CNN workloads and memory-intensive LLM inference using profiling-derived saturation points and MPS-based SM partitioning. We compare against two baselines: (i) a time-sharing (TS) baseline, where workloads are executed sequentially on the GPU via temporal multiplexing, and (ii) an isolated execution baseline, where each workload runs independently on a dedicated GPU without any co-location or resource sharing (Solo). Figure 6 summarizes the results.

Under time sharing, LLM inference experiences increase in latency and a corresponding reduction in throughput compared to solo execution. While the absolute degradation is limited, it reflects the inherent inefficiency of coarse-grained scheduling, where execution is periodically interrupted and forward progress is uneven. In contrast, *DynamoServe* 's MPS-based co-location preserves near-solo LLM performance. When SMs are allocated according to the LLM's profiling-derived knee point, both latency and throughput closely match solo execution. This demonstrates that *DynamoServe* provides sufficient compute resources to the LLM while avoiding unnecessary interference from the co-located CNN. CNN throughput exhibits a similar trend. Time sharing introduces small but consistent overheads due to reduced effective GPU utilization. With MPS-based sharing, CNN throughput remains close to its solo baseline, indicating that *DynamoServe* avoids over-allocating SMs beyond the CNN's saturation point while maintaining high efficiency. Although the performance differences between time sharing and MPS-based co-location are small, they are consistent across workloads. More importantly, MPS-based allocation provides predictable, near-solo performance for both workloads while enabling concurrent execution. These results validate *DynamoServe* 's design choice of using profiling-derived knee points to guide SM allocation, enabling efficient and interference-aware co-location without sacrificing performance stability.

4.3 Peer-GPU Offloading for Insufficient Single-GPU Capacity Scenarios

We next evaluate *DynamoServe* against FlexGen and Aqua in a scenario where the LLM's model weights and KV cache cannot fit within the memory of any single GPU, but can fit when aggregating the unused memory of multiple peer GPUs. This experiment highlights the limitations of existing offloading mechanisms and the benefits of *DynamoServe* 's fine-grained peer-GPU memory pooling. In this setup, all three systems are configured to offload OPT-30B's parameter model state when local GPU memory is insufficient. FlexGen and Aqua both require that the offloaded model state fit entirely within a single remote memory tier. As a result, when no individual peer GPU has sufficient free memory to hold the full model state, both systems fall back to host CPU memory. In contrast, *DynamoServe* can distribute LLM state across multiple peer GPUs, leveraging their aggregated stranded HBM capacity. Figure 7 summarizes the performance of the three systems. *DynamoServe* significantly outperforms both baselines in terms of both throughput and latency. Compared to FlexGen and Aqua, *DynamoServe* achieves substantially lower inference latency and more than doubles token throughput. Moreover, *DynamoServe* attains markedly higher GPU utilization, indicating that it is able to effectively exploit available GPU resources rather than relying on slow host memory. Aqua improves upon FlexGen by reducing CPU offloading overhead when a suitable peer GPU is available. However, in this experiment, Aqua is unable to utilize peer GPU memory because no single GPU has enough capacity to accommodate the full model weights and KV cache. Despite sufficient aggregate GPU memory across the node, Aqua's single-producer–single-consumer offloading abstraction forces it to spill excess state to CPU memory, incurring high latency and limiting throughput.

When KV cache size is small (e.g., short contexts), *DynamoServe* keeps the entire KV cache and activations in local GPU memory and offloads only model weights to peer GPUs. As sequence length grows, KV cache is gradually spilled to peer GPUs.

5 Related Work

The widening gap between accelerator compute throughput and on-device memory capacity has made memory management a central challenge for large model training and inference. Prior work addresses this through model parallelism, memory offloading, memory-centric scheduling, and disaggregated memory systems.

CPU and NVMe Offloading. Systems such as FlexGen [12], ZeRO-Offload [11], and HeteGen [18] extend memory capacity by offloading to CPU or storage. Although they enable larger models, they incur significant latency and bandwidth penalties due to slower memory tiers and are typically limited to single-node execution. In contrast, *DynamoServe* leverages peer GPU memory as a lower-latency alternative.

Memory-Centric Scheduling. Aqua [13] improves utilization under oversubscription via memory-aware scheduling. However, each model is primarily associated with a single GPU, with limited use of peer-GPU memory and CPU fallback under pressure. *DynamoServe* instead aggregates idle GPU memory across devices within a node to enable more flexible placement of model state.

Memory Disaggregation. Memory disaggregation has been explored in distributed systems and enabled by high-speed interconnects such as NVLink and CXL [2, 3, 7, 8]. These systems demonstrate remote memory access across devices, but do not address the latency-sensitive access patterns of LLM inference. *DynamoServe* adapts these ideas to GPU-to-GPU memory sharing for inference workloads.

LLM Inference Systems. LLM serving systems focus on batching, scheduling, and KV-cache optimization [9, 17], typically assuming model state resides in local GPU memory. Our work is complementary, targeting memory capacity limitations via intra-node distributed GPU memory.

In summary, prior work has explored offloading to slow memory tiers and memory-centric scheduling, but none fully exploit stranded peer GPU memory as a low-latency, cluster-local offloading tier for LLM inference. *DynamoServe* fills this gap by dynamically aggregating GPU memory across devices, enabling scalable and efficient inference under multi-tenant workloads.

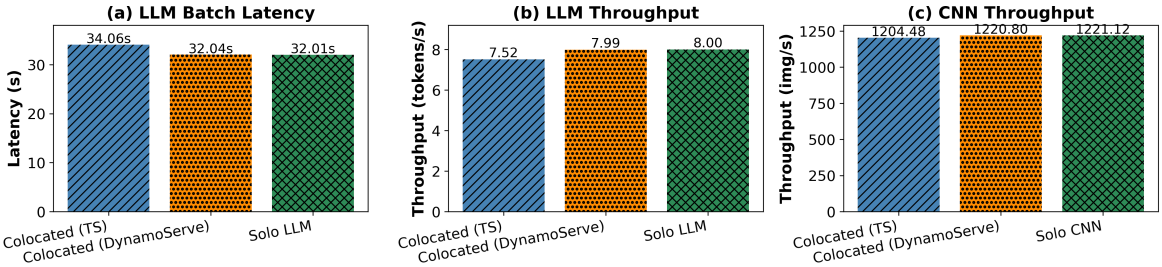


Figure 6: *DynamoServe*'s memory offloading has a negligible effect on the peer GPU's CNN, and only slightly reduces the performance of the LLM whose memory is being offloaded (vs. "solo" execution on a single GPU with sufficient memory). MPS multiplexing with fixed (knee) GPU percentage nearly eliminates the overhead vs. coarser-grained time slicing.

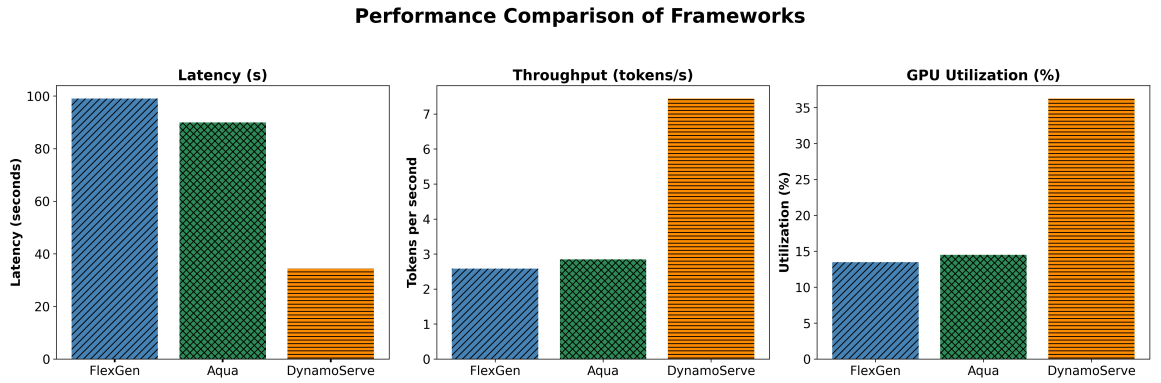


Figure 7: By offloading to the memory of multiple peer GPUs, *DynamoServe* improves latency, throughput, and utilization vs. FlexGen. Aqua is unable to achieve a similar benefit in this experiment: no single peer GPU has enough free memory, so Aqua offloads to DRAM over PCIe.

6 Conclusion

The rapid growth of LLM inference workloads has outpaced the memory capacity of modern accelerators, especially in multi-tenant GPU clusters where fragmentation and interference are inherent. This paper presented *DynamoServe*, a peer-GPU offloading framework that treats remote GPU memory as an elastic extension of local memory. By leveraging stranded memory on co-located GPUs and using high-bandwidth NVLink for data transfers, *DynamoServe* enables large-model inference without the overheads of CPU or NVMe offloading. Our dynamic weight migration and locality-aware prefetching policies reduce communication stalls, while our disaggregated prefill/decode execution model improves utilization and fairness compared to rigid tensor-parallel approaches. Across a range of models and workloads, *DynamoServe* delivers substantial gains in memory efficiency and scheduling flexibility while maintaining competitive end-to-end latency.

References

[1] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[2] CXL Consortium. 2022. Compute Express Link (CXL): Enabling Memory Disaggregation. https://www.computeexpresslink.org.

[3] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[4] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).

[5] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[6] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[7] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[8] NVIDIA Corporation. 2018. NVLink High-Speed GPU Interconnect. https://www.nvidia.com/en-us/products/workstations/nvlink-bridges/.

[9] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of machine learning and systems* 5 (2023), 606–624.

[10] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[11] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical*

Conference (USENIX ATC 21). 551–564.

[12] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[13] Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. 2025. Aqua: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 48–62.

[14] vllm Github. [n. d.]. vllm Github,. https://github.com/vllm-project/vllm. [Online; accessed 2025].

[15] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 995–1008.

[16] Tong Wu, Fuchun Yang, Wencong Zhang, Xiaohui Liu, et al. 2023. AntMan: Dynamic scaling on GPU clusters with guarantees. In *USENIX NSDI*.

[17] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[18] Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. *Proceedings of Machine Learning and Systems* 6 (2024), 162–172.