The Journal of Supercomputing (2025) 81:1232
https://doi.org/10.1007/s11227-025-07646-4

FlowLoader: an efficient large language model loader via parallel transmissions and local multi-tier caching

Jun Yin[1] · **Fei Wu**[1] · **Shihao Li**[2] · **Peng Huang**[2]

Accepted: 1 July 2025
© The Author(s) 2025

Abstract
The checkpoint of large language models (LLMs) typically range from tens of gigabytes to terabytes in storage size, with loading processes often requiring several seconds to minutes. This substantial latency issue significantly impacts service level objective compliance for providers of large-scale model services. To address the LLM loading cold-start problem, we present LLM FlowLoader, a model loader featuring multi-tier local caching and parallel communication pipeline optimization. By strategically leveraging GPU servers' heterogeneous network resources and storage hierarchies, our system accelerates checkpoint loading through two key innovations: (1) An asynchronous pipeline transporter based on task stealing and affinity domain awareness, which transforms complex network transmission into simplified chain-structured transmission by introducing affinity domains, and adaptively enhances I/O bandwidth through a task-stealing algorithm. (2) A multi-tier caching system co-designed with the transporter that exploits GPU servers' multi-tier memory/storage architecture to accelerate data accessibility and reduce cold-start latency. Experimental results demonstrate that LLM FlowLoader not only achieves significant loading speed improvements but also provides novel insights for optimizing GPU resource utilization.

Keywords Large language models · Cold-start · Parallel transmission · Multi-tier caching · Task stealing

1 Introduction

In recent years, large language models (LLMs) with hundred-billion-scale parameters have revolutionized natural language processing. However, their exponentially increasing computational and storage requirements create dual challenges for modern inference serving systems [1]. A typical LLM deployment demands

Fei Wu and Shihao Li contributed equally to this work.

Extended author information available on the last page of the article

Published online: 12 August 2025

persistent storage ranging from tens of gigabytes to terabytes, coupled with coordinated computation across multiple high-end GPUs [2]. Consider the widely used LLaMa2-70B [3] model: When implementing tensor parallelism with parameter sharding optimization, it still requires at least eight NVIDIA A100 GPUs (40 GB VRAM) to satisfy the video memory requirements for a single inference instance. Most model vendors currently provide inference capabilities as a service, yet the cold-start latency during model initialization poses a severe challenge to the SLO (service level objective) of large model services [4–6].

While existing approaches like TrIMS [7], Clockwork [8], and Computron [9] have demonstrated success in mitigating cold-start issues for smaller models(e.g., maintaining ResNet-50's 100MB parameters in memory [10]) through GPU-as-cache strategies with pre-warmed instances, these solutions prove inadequate for modern LLM deployments. The prohibitive GPU memory footprint of LLM (typically ranging from tens to hundreds of gigabytes) and constrained server resources make persistent model retention in memory fundamentally impractical. This necessitates on-demand loading strategies as the only viable solution, thus making the design of efficient model loaders a critical systems challenge. In cloud service scenarios, current large model loaders primarily face three key technical barriers:

1. Bursty Traffic: When burst traffic exceeds the capacity of existing service instances, the system must complete cold starts for new instances within seconds to meet SLO requirements. However, loading the model parameters of LLaMA-2-70B across 8 GPUs necessitates an 84-second cold-start process [5, 11], resulting in a Time to First Token (TTFT) that severely exceeds acceptable operational thresholds.
2. Preemption: In multi-tenant shared GPU cluster environments, resource schedulers must dynamically adjust model instance deployments based on priority [12]. For example, preemptive tasks from high-priority tenants may forcibly evict low-priority LLM instances, requiring cold-start processes to be reinitiated when the original tenant resumes service.
3. Resource Costs: The substantial GPU costs of LLM services compel providers to adopt aggressive resource reclamation strategies. Statistics show that idle instances are typically reclaimed after just 120 s of inactivity to control costs [13]. However, short request intervals cause instances to enter a "start-stop oscillation" state, where cold-start overhead accounts for over 60% of the service cycle [14].

Currently, service providers face the dual challenges of escalating costs and prohibitive startup latency, which equally persists in on-premises or offline deployment scenarios.

While cold-start direct loading appears to offer the most straightforward initialization strategy for LLM deployment, this approach fundamentally exposes systemic inefficiencies in contemporary parameter loading paradigms. State-of-the-art loading frameworks (e.g., PyTorch [15], Safetensors [16]) employ redundant data transfer mechanisms that significantly degrade loading performance.

Specifically, these implementations adopt a suboptimal two-stage materialization process: Model checkpoints are first deserialized into CPU host memory through conventional I/O pipelines, followed by subsequent host-to-device memory transfers to GPU accelerators. This dual-copy architecture, originally designed for conventional deep learning models, fails to address the unique challenges of LLM parameter orchestration where single-node memory constraints necessitate multi-GPU distribution across heterogeneous memory hierarchies [5]. Moreover, current loaders lack essential optimizations for modern LLM deployment scenarios, including parameter sharding-aware prefetching, zero-copy direct device initialization, and tensor parallelism-aware placement strategies.

Modern GPU servers present heterogeneous memory subsystems where computational resources (GPU's High Bandwidth Memory, HBM), host memory (Dynamic Random-Access Memory, DRAM), and persistent storage (Nonvolatile Memory Express, NVMe SSDs) form tiered hierarchies interconnected through high-bandwidth interfaces (PCIe 4.0/5.0, NVLink 3/4) [17, 18]. Conventional loading methods fail to exploit these parallelized data pathways due to their rigid serialized execution model. To overcome this architectural mismatch, we propose LLM FlowLoader, which strategically partitions and caches model checkpoints across local storage tiers based on their capacity and bandwidth characteristics. Our key innovations include:

- Efficient Asynchronous Pipelining Mechanism: By introducing the concept of affinity domains, LLM FlowLoader transforms complex network transmissions into streamlined chain-based transfers while ensuring transmission determinism. The pipeline employs a hierarchical scheduling strategy that combines static partitioning with dynamic adjustment. Specifically, the system first performs static segmentation based on the topological structure of GPU clusters and link bandwidth, followed by dynamic load balancing through a task-stealing algorithm.
- Multi-Tier Local Cache Architecture: LLM FlowLoader leverages multi-level storage resources in GPU servers to partition and distribute model checkpoints across different storage tiers based on access frequency and size. This approach enables rapid access to frequently used model parameters, thereby reducing the time overhead of model cold starts.

The remaining sections of this paper are structured as follows. Section 2 elaborates on our fundamental observations and the derived research motivations. Section 3 then presents the architectural framework and design principles of our proposed system. Finally, Sect. 5 demonstrates the experimental results through comprehensive evaluations, validating the effectiveness of our approach.

2 Motivation

In modern GPU servers, a single PCIe device typically occupies only a portion of the server's PCIe lanes. For instance, AMD EPYC™ Rome processors provide 128 PCIe 4.0 lanes [19], where a single GPU is typically configured with 16 PCIe lanes.

According to PCIe 4.0 specifications, a x16 configuration theoretically provides up to 32 GB/s of unidirectional bandwidth [17, 18]. Meanwhile, an SSD using PCIe 3.0 x4 interface can achieve approximately 4 GB/s of unidirectional bandwidth [20]. Furthermore, DRAM bandwidth scales with channel configurations: A DDR4 channel delivers 17−25.6 GB/s, quadrupling to 68−102.4 GB/s in quad-channel setups [21]. For GPU-to-GPU communication, NVIDIA's NVLink 3.0 provides approximately 300 GB/s of bandwidth [17]. However, even the most advanced model loading frameworks fail to fully utilize these high-speed bandwidths. These inference frameworks typically employ synchronous I/O approaches for model loading, which can only utilize the bandwidth of a single channel. Even when using multiple links, they operate in synchronous blocking calls, resulting in significant underutilization of the high-speed bandwidth provided by GPU servers.

2.1 Hardware

In this section, we investigate the time consumption of data loading from SSD, CPU memory, and GPU memory under different scenarios, as well as the efficiency of parallel multi-task loading. Our experiments were conducted on a server equipped with four A800 GPUs interconnected via NVLink 3.0 x8, where each GPU is indirectly connected to the CPU through independent PCIe switches. The server features dual AMD EPYC^{TM} Rome processors with 16-channel DDR4 3200 MHz memory modules, each channel theoretically providing 25 GB/s bandwidth, and is configured with a 1TB SSD supporting PCIe 3.0 x4.

2.2 Case studies

In this section, we conduct comprehensive bandwidth testing and phenomenon analysis under various scenarios on the aforementioned GPU system. We perform real-world bandwidth measurements for major storage components including CPU memory, SSD, and even GPU memory. Since model checkpoint files typically range from 2GB to over 100GB [3, 22], our experiments specifically focus on large-file/memory transfer scenarios and exclusively examine read operations, as model checkpoint loading primarily involves read access. Our test cases employ frequently used communication primitives(e.g., broadcast, gather [23, 24]) to demonstrate the parallel transmission capabilities of the GPU server's communication links.

2.2.1 P2P

Figure 1a presents the measured maximum peer-to-peer (p2p) data transfer bandwidth between SSD, CPU memory, and GPUs in our GPU server configuration. Our measurements show that transferring data from SSD to CPU memory achieves a peak bandwidth of approximately 3.0 GB/s using 16 threads, while CPU-to-GPU memory transfers reach up to 25 GB/s. For inter-GPU communication, NVLink 3.0 delivers 174 GB/s, whereas PCIe-based transfers achieve 25 GB/s. Due to architectural constraints and protocol overhead, all measured bandwidths fall below their

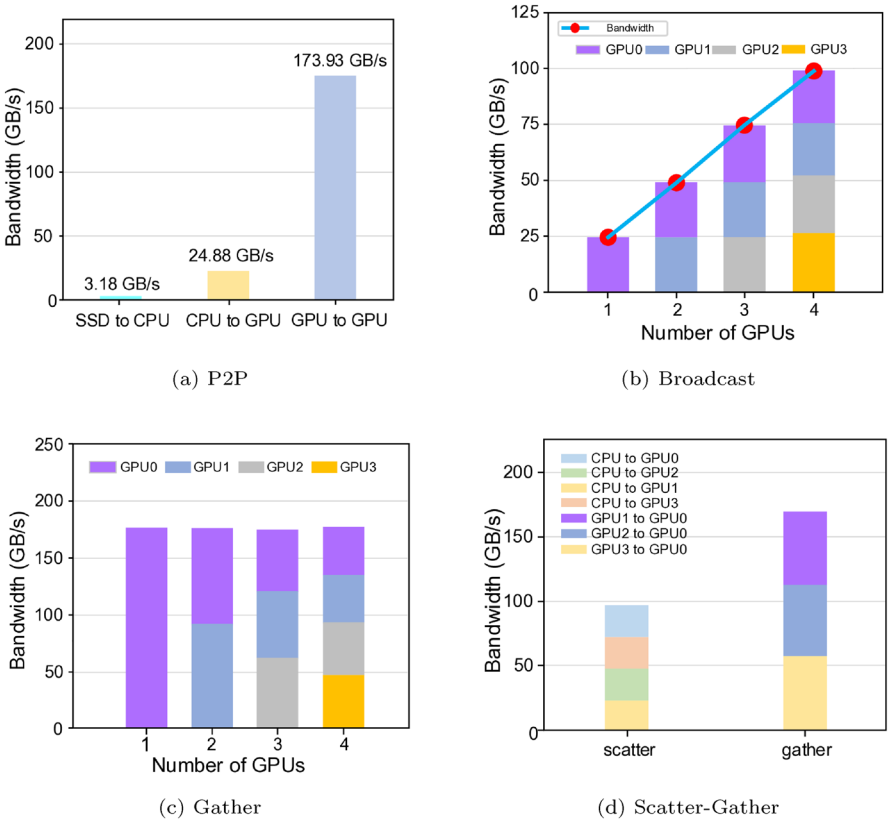


Fig. 1 Collective Communication Bandwidth Test

theoretical maximums. These P2P baseline measurements will guide subsequent bandwidth analysis of various parallel communication operations.

2.2.2 Broadcast

Figure 1b demonstrates the aggregate bandwidth of broadcast operations (simultaneously copying data from the CPU to multiple GPUs) as we progressively increased the number of GPUs. In this test, we accounted for the impact of the NUMA (Nonuniform Memory Access) architecture [25] by binding each GPU to an independent process and using the `numactl` command to pin both memory and processes to their corresponding NUMA nodes. Our measurements showed that as we incrementally added GPUs for broadcast operations, each GPU consistently achieved peak bandwidth (approximately 25 GB/s), aligning with the server's configuration of providing dedicated PCIe lanes for each of the four GPUs. These results indicate that GPUs generally possess the capability to execute large-scale parallel transmission tasks efficiently.

2.2.3 Gather

Figure 1c illustrates the bandwidth utilization when gathering data from other GPUs to GPU0. It can be observed that as we incrementally increase the number of GPUs participating in the gather operation, the aggregated data loading bandwidth on GPU0 remains stable at 174 GB/s, with the bandwidth allocation among source GPUs exhibiting near-equitable distribution. Given that NVLink bandwidth substantially exceeds PCIe capabilities, during single-node CPU-GPU data transfers, we can leverage idle PCIe channels from auxiliary GPUs for initial data reception, followed by NVLink-enabled GPU-GPU parallel transmission. This indirect pathway effectively circumvents the bandwidth limitations inherent to direct single-channel transfers.

2.2.4 Scatter–gather

Figure 1d illustrates the bandwidth performance of the scatter+gather collective communication pattern. Using a pipelined transmission approach, we partitioned the data into blocks for multi-stage transfers. First, data in CPU memory was scattered across four GPUs, followed by a concurrent gather operation to consolidate data from all four GPUs onto GPU0. The aggregate communication bandwidth between GPUs reached approximately 174 GB/s, while the total bandwidth between CPU memory and GPUs measured about 100 GB/s. Through the scatter and gather operations, we expanded the p2p communication bandwidth between CPU memory and GPUs from approximately 25 to 100 GB/s.

2.3 Analysis and insights

PCIe and NVLink deliver high-speed communication bandwidth for modern GPU-based high-performance computing servers. However, due to device bandwidth limitations and system design constraints, these resources remain underutilized. For instance, in the aforementioned GPU server, while the GPUs demonstrated a measured bandwidth of 25 GB/s, the process of loading data from SSDs to GPUs was bottlenecked by the SSD's PCIe 3.0 x4(3 GB/s) interface. To address this, preloading partial data into CPU memory or leveraging operating system file caching techniques could fully utilize GPU bandwidth. Additionally, for NVLink-interconnected GPUs, neighboring GPUs could act as data transfer relays, combined with parallel channels in CPU memory for data movement. In the server's PCIe architecture, by utilizing an NVLink bypass, we achieved data transfers from CPU memory to GPUs at a bandwidth of 100 GB/s, compared to the original single PCIe lane's 25 GB/s. Conversely, when performing peer-to-peer GPU transfers via NVLink, introducing a PCIe bypass through CPU mediation could theoretically add 32 GB/s of bandwidth, representing a 10% improvement over standalone NVLink transfer speeds.

These experiments and analyses highlight that even state-of-the-art model loaders fail to fully exploit the rich communication links in GPU servers. By strategically

leveraging the hierarchical storage architecture of GPU servers and designing asynchronous pipelined transmission schemes, we can parallelize communication pathways to maximize aggregate bandwidth utilization.

3 System design

Next, we introduce LLM FlowLoader: a model loader specifically designed for large language models. The core motivation of this design stems from an in-depth observation of modern GPU communication architectures: Although mainstream GPUs (such as the A100/H800 series) support multimodal communication interfaces (PCIe, NvLink), existing model loading systems (e.g., PyTorch, Safetensors) predominantly rely on single-link serial transmission modes. This results in underutilized parallel bandwidth of GPU topology networks. LLM FlowLoader aims to integrate the potential communication bandwidth of these channels to accelerate large model loading and deliver higher SLO performance.

3.1 Transmitter design

3.1.1 Pipeline transmission

We draw inspiration from the actor communication model [26] to abstract hardware devices and communication topology networks at the software level: Storage devices in GPU servers (including SSDs, DRAM, and GPUs) are abstracted as actors, while physical communication links are abstracted as bidirectional transports in the software layer. Each actor communicates with others via transports, and an actor may have multiple transports to establish point-to-point connections with other actors. If the transport is interconnected through PCIe switch or NVSwitch on the physical hardware, it will share the same task queue (discussed in Sect. 3.1.3). By defining the concepts of actors and transports, we construct a software-level communication topology model for GPU servers.

To uniquely identify communication endpoints, each actor is assigned an ID composed of DeviceType, id. Additionally, we provide a send primitive for communication invocation. The parameters of the send primitive include:

- Source information: Address and length of the source data
- Source actor ID: ID of the device where the source data resides
- Destination information: Address of the target location
- Destination actor ID: ID of the target device

By explicitly specifying source and destination addresses, we abstract away the complex underlying communication processes and simplify the invocation workflow. Since transmissions are asynchronous, completion notifications are delivered via callback functions. To accelerate communication speed and increase I/O bandwidth, we implement the following measures:

Data Sharding. Since data transmission may involve multi-stage transfers (e.g., loading data from SSDs to GPUs typically requires CPU intermediation), we employ data sharding to increase I/O bandwidth and reduce latency. The source data is divided into multiple chunks for pipelined transmission. Each chunk contains transmission metadata with sequential offsets that map one-to-one between the source and destination addresses. Actors in the pipeline are equipped with a ring buffer. The ring buffer is implemented as a circular queue with head and tail pointers, enabling first-in-first-out lock-free access. Its slots consist of memory units matched to the storage type (e.g., SSD DMA buffers, CUDA pinned memory).

Out-of-Order Transmission. By splitting transmission tasks into chunks, we leverage parallel communication links in the topology network to accelerate transfers. Due to varying speeds across links, chunks may arrive at the destination out of order. However, since the source and destination addresses of each chunk are strictly mapped during initialization, the final data sequence remains intact despite disordered transmission, ensuring correctness of the copied results.

Two-Tiered Pipeline Transfer. Data copying between actors occurs via preestablished transports. Upon receiving a task, the source actor sends the transmission request to the corresponding transport using the method described in Sect. 3.1.2. The transport then notifies the destination actor to provide a target address. If the destination actor is the final target, it directly supplies the output address. For intermediate buffer actors, the transport allocates a memory element from the destination's ring buffer as a copy buffer. This intermediate actor then acts as the new source actor, propagating the chunk until it reaches the final destination. This multi-stage transfer constitutes the first-tier pipeline. Each transport maintains two task queues: (1) Main task queue; (2) Task-stealing queue (detailed in 3.1.3). A dedicated event-loop thread in the transport continuously polls for pending tasks, notifies destination actors, and executes copy operations. Since notifying actors incurs inter-actor communication overhead (introducing pipeline bubbles), we implement a second-tier pipeline within the transport. By overlapping I/O operations with communication latency, this design effectively masks idle time and minimizes transmission bubbles (Fig. 2).

In addition, a `send` communication primitive is provided to enable invocation. The parameters of the `send` primitive include source information (comprising the source data address, data length, and the ID of the actor residing on the source

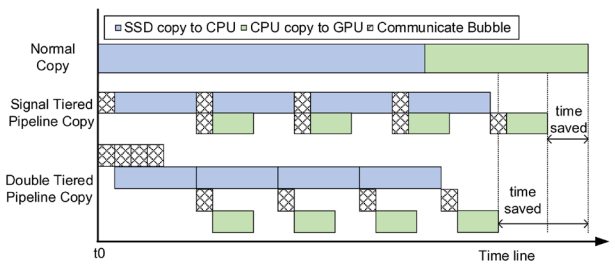


Fig. 2 Example of two-tiered pipelined transfer

device) along with the destination address and the ID of the actor on the destination device. By explicitly specifying both source and destination addresses, our design abstracts away underlying communication complexities and simplifies the invocation process for pipelined transmission mechanisms. As the transmission operates asynchronously, completion notifications are implemented through `callback` functions to signal the conclusion of data transfers. We employ a copy completion counter to verify full data transfer execution. Specifically, upon each successful transfer of an individual partitioned data chunk, the corresponding copy counter is atomically incremented. When the accumulated count equals the predetermined total number of complete data blocks, the transmission pipeline automatically triggers the callback function to signal global task completion, thereby implementing an asynchronously synchronized notification mechanism.

The architecture of our designed pipelined transmission mechanism is illustrated in Fig. 3. In this figure, we present the pipeline transmission workflow using data transfer from SSD to GPU as a case study. The diagram employs distinct color schemes to differentiate components affiliated with different actors: Brown tones represent SSD actor domains (encompassing data, transport mechanisms and actor entities), blue tones denote CPU actor domains, while green tones indicate GPU actor domains. Blue arrows annotate task execution sequences, whereas red arrows demarcate data transmission paths. The pipeline operates through eight sequential phases: (1) Invoke the send interface exposed by the flowloader module; (2) The source SSD actor partitions both source data and destination addresses into mutually corresponding blocks based on fixed offset alignment; (3) Submit copy tasks to the SSD-to-CPU transport channel; (4) The transport module acquires memory allocation from the ring buffer managed by its connected CPU actor; (5) Perform data copy operations from a partitioned source block to the allocated ring buffer; (6) The CPU actor dispatches copy tasks to the target GPU actor through hardware-accelerated queuing mechanisms; (7) The destination GPU actor executes final data

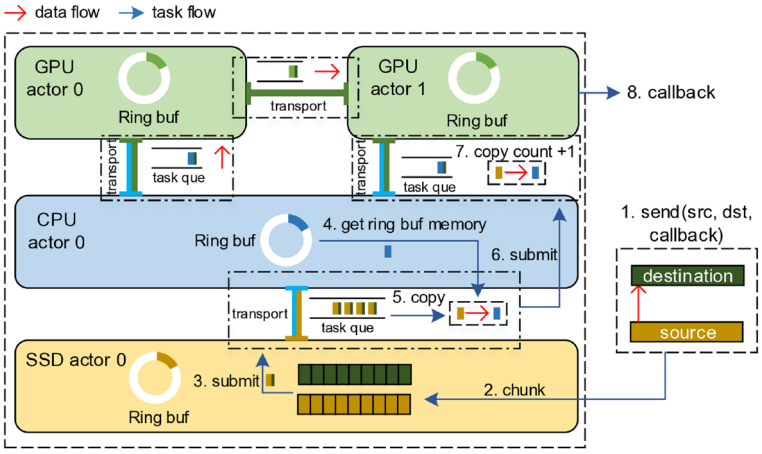


Fig. 3 Schematic diagram of the pipelined transmission architecture

copy operations to device memory and increments the copy completion counter; (8) When the counter matches the predetermined total block count, the system invokes callback functions to signal global transfer completion, ensuring asynchronous operation synchronization.

3.1.2 Transmission paths

The communication topology in GPU servers typically exhibits a hierarchical structure: Links closer to GPUs deliver higher bandwidth [17, 18, 27]. Modern servers also feature symmetry in communication topology through NUMA architectures [17, 18, 25], enabling the modeling of parallel data transmission across hierarchical links as a multi-source multi-sink maximum flow problem. However, classical maximum flow algorithms [28, 29] (e.g., Ford-Fulkerson [30], Edmonds-Karp [31]) demonstrate significant limitations in dynamic network environments. Their implementation complexity grows exponentially with network scale and requires global topological information for repeated augmenting path computations, rendering them unsuitable for scenarios with real-time link load fluctuations. Moreover, the parallel transmission paths generated by such algorithms often span multiple topological layers, resulting in frequent cross-layer device switching that induces computational resource contention and additional scheduling overhead. To address the maximum flow problem in GPU servers, the designed algorithm should therefore satisfy two critical requirements: minimizing cross-layer device switching and prioritizing the selection of adjacent links for data transmission, thereby optimizing transfer efficiency through localized routing decisions. Inspired by greedy algorithms, we propose an affinity domain-aware dynamic parallel path selection algorithm to tackle this challenge.

Definition 1 (Affinity Domain) For a connected graph $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is the vertex set and $\boldsymbol{E}$ is the edge set, given a destination vertex $a \in \boldsymbol{V}$, the affinity domain $\boldsymbol{\Phi}_a^d$ is defined as the set of all vertices at a distance d from a:

$$\boldsymbol{\Phi}_a^d = \{v \in \boldsymbol{V} \mid d(a, v) = d\} \tag{1}$$

where $d(a, v)$ denotes the shortest path length between vertices a and v in $\boldsymbol{G}$.

Definition 2 (Affinity Distance) For a connected graph $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$, given a destination vertex $a \in \boldsymbol{V}$, if a vertex $b \in \boldsymbol{\Phi}_a^d$, the affinity distance $\rho(\boldsymbol{\Phi}_a^d)$ of the affinity domain $\boldsymbol{\Phi}_a^d$ to a is defined as $d(a, b)$.

Theorem 1 (*Existence and Uniqueness*) *For any connected graph* $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$, *given a destination vertex* $a \in \boldsymbol{V}$ *and an arbitrary starting vertex* $b \in \boldsymbol{V}$, *the affinity distance* $\rho(\boldsymbol{\Phi}_a^d)$ *of the affinity domain containing* b *must exist and is unique*.

Proof Since $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$ is connected, for any two vertices $a, b \in \boldsymbol{V}$, there exists a minimal path $p_{\min}$ with length $\ell(p_{\min})$. By the definition of affinity distance,

$d(a,b) = \ell(p_{\min})$. Thus, the affinity distance $\rho(\mathbf{\Phi}_a^d)$ for any $b \in \boldsymbol{V}$ relative to a must exist and is uniquely determined. □

Definition 3 (Adjacency) For a connected graph $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$ and a destination vertex $a \in \boldsymbol{V}$, two affinity domains $\mathbf{\Phi}_a^{d_1}$ and $\mathbf{\Phi}_a^{d_2}$ are termed adjacent if their affinity distances differ by 1, i.e., $|\rho(\mathbf{\Phi}_a^{d_1}) - \rho(\mathbf{\Phi}_a^{d_2})| = 1$.

Theorem 2 (*Adjacent Affinity Domain Direct Connectivity*) *For any connected graph* $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$, *destination vertex* $a \in \boldsymbol{V}$, *and adjacent affinity domains* $\mathbf{\Phi}_a^{d_1}$ *and* $\mathbf{\Phi}_a^{d_2}$ (*assuming* $\rho(\mathbf{\Phi}_a^{d_2}) > \rho(\mathbf{\Phi}_a^{d_1})$), *any vertex* $c \in \mathbf{\Phi}_a^{d_2}$ *must have a direct edge to at least one vertex* $b \in \mathbf{\Phi}_a^{d_1}$.

Proof By the connectivity of $\boldsymbol{G}$, the shortest path from $c \in \mathbf{\Phi}_a^{d_2}$ to a must include a vertex b such that $d(a,b) = d(a,c) - 1$. By Theorem 1 and the definition of affinity domains, $b \in \mathbf{\Phi}_a^{d_1}$. Therefore, b and c are directly connected by an edge in $\boldsymbol{E}$. □

Theorem 3 (*Adjacent Affinity Domain Direct Connectivity*) *For a connected graph* $\boldsymbol{G}(\boldsymbol{V},\boldsymbol{E})$ *and a destination vertex* $a \in \boldsymbol{V}$, *if there exist affinity domains* $\mathbf{\Phi}_a^{d_1}$ *and* $\mathbf{\Phi}_a^{d_2}$ *adjacent to* $\mathbf{\Phi}_a^{d}$ *then* $\mathbf{\Phi}_a^{d_1}$ *and* $\mathbf{\Phi}_a^{d_2}$ *cannot be adjacent.*

Proof Assume, for contradiction, that $\mathbf{\Phi}_a^{d_1}$ and $\mathbf{\Phi}_a^{d_2}$ are adjacent. By Definition 3, their affinity distances differ by 1. Without loss of generality, suppose $\rho(\mathbf{\Phi}_a^{d_2}) < \rho(\mathbf{\Phi}_a^{d}) < \rho(\mathbf{\Phi}_a^{d_1})$. By Theorem 2, there exists a vertex $c \in \rho(\mathbf{\Phi}_a^{d_2})$ directly connected to a vertex $b \in \rho(\mathbf{\Phi}_a^{d_1})$. This implies $d(a,b) = \rho(\mathbf{\Phi}_a^{d_1})$ and $d(a,c) = \rho(\mathbf{\Phi}_a^{d_2})$. However, since b and c are adjacent, the shortest path from a and c could pass through b, leading to $\rho(\mathbf{\Phi}_a^{d_2}) = \rho(\mathbf{\Phi}_a^{d_1}) - 1$, which contradicts $\rho(\mathbf{\Phi}_a^{d_2}) < \rho(\mathbf{\Phi}_a^{d}) < \rho(\mathbf{\Phi}_a^{d_1})$. Hence, the assumption is invalid (Fig. 4). □

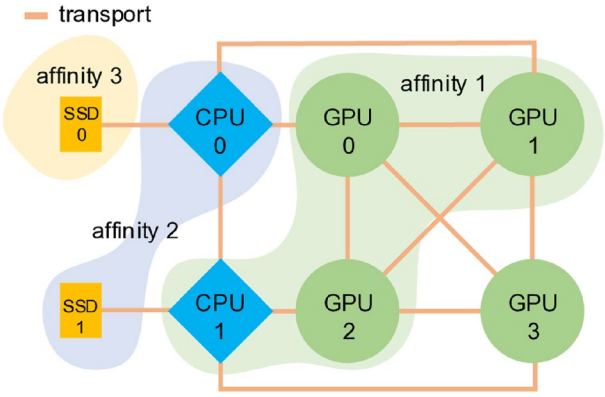


Fig. 4 Schematic Diagram of GPU3's Affinity Domain

The communication topology among GPU servers exhibits connectivity. For any destination actor b, Theorem 1 (Existence and Uniqueness) ensures that all other vertices belong to unique affinity domains. When a node in actor a distant affinity domain needs to transmit tasks to the destination actor, it only needs to find a path to an actor node in the immediately adjacent higher-level affinity domain at each step. According to Theorem 3 (Monotonic Adjacency), the transmission path formed by affinity domains is inherently loop-free, thus guaranteeing eventual delivery to the destination actor node.

Specifically, the affinity distance from any actor to the start actor a can obtain by Breadth-First Search (BFS) algorithm [32], which systematically explores nodes in increasing order of their topological distance. Since BFS inherently constructs a tree-like hierarchical structure centered at actor a, where each traversal layer corresponds to an affinity domain level, this algorithm naturally satisfies the requirement of partitioning nodes into unique affinity domains based on topological proximity. Consequently, BFS is employed to compute both affinity distances and affinity domains. Then, for any starting actor a, the transmitted data is forwarded by sequentially choosing the next hop from adjacent actors within progressively closer affinity domains toward destination actor b. This mechanism transforms complex topology-aware routing into a unidirectional pipelined transmission, significantly simplifying the problem.

Furthermore, if the subgraph $\boldsymbol{G'}$ formed by actors within the same affinity domain is connected, parallel transmission can be employed: Partial data from one actor can be offloaded to another connected actor within $\boldsymbol{G'}$, leveraging parallel channels to achieve load balancing and accelerate transmission speed.

We formalize this approach in Algorithm 1 for path selection.

Algorithm 1 Transport Selection Algorithm

Input: starting actor a, destination actor e, communication topology network of servers G
Output: best transport

if $s == e$ **then**
  return *null* ▷ Already reached the end actor
end if
$current_distance \leftarrow affinity_domains[e][s]$ ▷ Calculate the affinity distance between node s and node e using precomputed BFS results.
$target_distance \leftarrow current_distance - 1$
$best_transport \leftarrow null$
$min_load \leftarrow \infty$
for each $neighbor$ in $G.neighbors(s)$ **do**
  if $affinity_domains[e][neighbor] == target_distance$ **then**
    $transport \leftarrow G.get_transport(s, neighbor)$
    $bandwidth \leftarrow get_bandwidth(transport)$ ▷ get transport's bandwidth
    $tasks \leftarrow get_pending_tasks(transport)$ ▷ get pending tasks for the transport
    $load_value \leftarrow bandwidth/tasks$
    if $load_value < min_load$ **then**
      $best_transport \leftarrow transport$
      $min_load \leftarrow load_value$
    end if
  end if
end for
return $best_transport$

3.1.3 Task stealing

To enhance data transmission efficiency in GPU servers during large-scale parallel processing, we propose a task-stealing methodology that dynamically exploits idle bandwidth resources across peer and secondary affinity domains. This approach enables adaptive load redistribution by opportunistically rerouting data flows through underutilized links when network congestion or load imbalance occurs. The proposed mechanism not only optimizes resource utilization by leveraging dormant transmission pathways but also ensures uninterrupted data delivery through real-time task rescheduling. By implementing this approach, we achieve significant improvements in both transmission velocity and system throughput for GPU-accelerated parallel computing tasks.

Structural Components. The transport primarily consists of a local task queue, a stealable task queue, and a task loop thread. When an actor submits a transmission task to the transport, the task is first directed to the local task queue. To increase transmission bandwidth, the system allows additional parallel links to steal tasks from this transport's stealable task queue. However, since the stealing

path is not the shortest transmission route (requiring multiple task hops), introducing parallel paths may degrade performance when local tasks are scarce. Therefore, we set a threshold for the local task queue: Only when the number of local tasks exceeds these thresholds are newly submitted transmission tasks placed into the stealable task queue, thereby reducing unnecessary steal operations involving multiple transfers during low local task loads. When tasks exist in the stealable task queue, other stealable transports may initiate stealing. Each steal operation retrieves only one transmission task, and this process continues until no processable tasks remain in the stealable task queue of the targeted transport(Fig. 5).

Stealing Process. The task loop thread continuously checks the local task queue for pending transfer tasks. If no executable task exists, the thread subsequently examines its own work-stealing task queue. When both queues are empty, the thread initiates a traversal of work-stealing queues belonging to adjacent transports within the same actor to locate stealable transfer tasks. In the absence of executable tasks throughout this hierarchical search, the cyclic thread enters a dormant state until subsequent activation. Upon awakening, it resumes the aforementioned inspection and execution cycle until successfully completing a transfer task. This operational workflow persists iteratively across all cyclic threads in each transport of the transmission system, forming a perpetual execution mechanism.

Stealable Transport. The determination of whether a neighboring transport is eligible for task stealing follows these criteria:

- Same Minimal Ring and Path Constraint: The stealing transport and the target transport must reside within the same minimal ring, and the path between actors in this minimal ring must not traverse the same PCIe switch or NvSwitch.
- Task Availability and Path Idleness: The work-stealing queue of the target transport must contain pending transfer tasks, and the transmission path in the opposite direction within the same minimal ring must be idle.

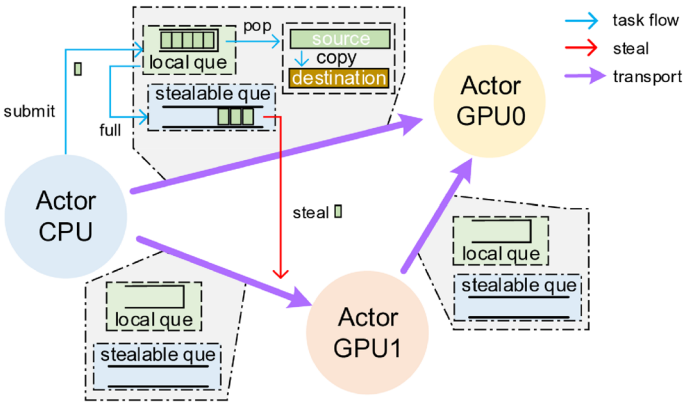


Fig. 5 Task-Stealing Schematic Diagram

- Affinity Domain Proximity: The actor connected to the stealing transport must reside in the same level or an adjacent affinity domain as the current actor.

If all the above conditions are satisfied, tasks in the target transport's work-stealing queue may be stolen.

3.2 Cache design

As mentioned in [6, 27, 33], caching model parameters closer to the GPU can significantly reduce the time required to load them. Although advanced GPUs now offer up to 80GB of VRAM [34], this capacity remains insufficient for models exceeding 30B parameters (which require at least 60GB of VRAM). Furthermore, host memory (DRAM), which often scales to terabyte (TB) levels, is well suited for caching model weights to accelerate loading times [17, 21]. Beyond host memory, storage systems composed of multiple SSDs can provide hundreds of TBs of capacity [20, 35, 36], while their combined parallel bandwidth rivals that of dedicated high-speed network storage servers. While proximity to the GPU enables higher communication bandwidth, it comes with limited capacity. To effectively utilize the multi-tier storage resources of GPU servers and accelerate the loading process of large model checkpoints, we have redesigned an efficient caching system tailored to the characteristics of large model usage scenarios.

3.2.1 Cache tiering

We have constructed a hierarchical caching architecture based on the access speed differentials of storage media: GPU VRAM constitutes the tier-1 high-speed cache, CPU memory serves as the tier-2 buffer layer, while SSDs and HDDs collectively form the tier-3 persistent storage tier (Fig. 6). Building upon this framework, the system achieves vertical coordination optimization across heterogeneous storage systems through parallelized communication channels in GPU servers, with the hierarchical caching architecture exhibiting the following key characteristics:

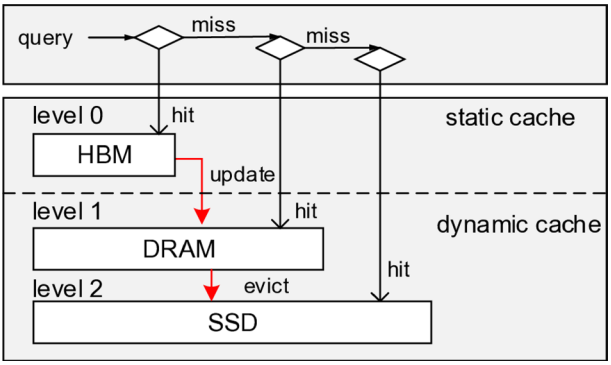


Fig. 6 Multi-tier cache system

Dynamic and Static Caching. Large language models frequently exhaust GPU memory resources during inference tasks. Under these conditions, the GPU is clearly unsuitable for managing dynamic caching. Once the GPU successfully loads a model checkpoint, we allow this portion of GPU memory to be registered within the caching system as static cache. Conversely, SSD storage and CPU main memory serve as the dynamic cache. Due to the temporal locality characteristics exhibited by LLM inference requests and the need for performance optimization, we implemented the caching tier using LevelDB [37] (managed via an LRU policy).

Cache Update Mechanism. When the GPU executes a model loading task, two primary actions occur:

- Static Cache Update: The static cache first decrements the reference count (if applicable) of any previously cached layer instance. Subsequently, it increments the reference count of the newly loaded layer. Crucially, unloading a layer does not decrement its reference count. This means that an unloaded model layer is not immediately freed from GPU memory but persists as cached data.
- Dynamic Cache Update: After a layer is loaded, it triggers an update task for the dynamic cache. If the layer is not found (i.e., a cache miss) in the nearest available dynamic cache level (e.g., CPU RAM before SSD), its reference count is incremented and its data is propagated to that nearest dynamic cache level. Upon completion of this update, the reference count is decremented.

Layer data with a reference count of zero is automatically evicted from GPU memory. Importantly, the reference counts for layers involved in an inference task are not decremented upon task completion. Consequently, these layers remain resident in GPU memory as static cache until the next model loading operation commences. Data evicted from any cache level is inserted into the subsequent lower level of the caching hierarchy (e.g., from GPU to CPU RAM, or from CPU RAM to SSD).

Cache Chunk and Consistency. Considering the layered architecture of Transformer-based LLMs, we adopt model layers as fundamental units for chunk granularity partitioning. Each cache chunk corresponds to an independently loadable parameter set of model layers. The system maintains a key-value structure using <model name, version number, model layer index> triplets, with each cache level employing an exclusive LevelDB instance to manage its hash-based index. Global mapping relationships are established through multi-level cascading traversal. During cache queries, the cache manager prioritizes locating target chunks via hash keys in the current level. Cache misses trigger recursive queries in lower-level storage. For model updates, we implement atomic switching through version number increments to ensure parameter consistency. Legacy version chunks will be automatically evicted when no access references exist.

3.2.2 Cache query strategy

Our system implements a hierarchical cache query protocol with asynchronous pipeline optimization. During model loading operations, the transporter architecture sequentially queries multiple cache tiers: (1) local static cache, (2) remote

static caches, and (3) dynamic cache pools. Upon any cache hit detection, the system immediately dispatches an asynchronous tensor loading task to the processing pipeline while continuing subsequent cache queries. This design effectively overlaps the data transfer latency with cache traversal operations through parallel execution. For tensors unresolved through all cache layers, the system initiates direct loading from persistent repositories.

4 Implementation details

We implemented the entire software using C++ and Python: The core components were developed in C++, while Python was used to provide external Python API interfaces. To achieve high loading efficiency, we adopted the following structural optimizations:

Efficient Model Loading Point. We implemented the fast model checkpoint strategy from literature [5]:

- Block-based sequential storage: Tensors on each GPU are organized as raw binary data files using chunk partitioning, avoiding metadata storage to enable large contiguous reads.
- Memory-aligned index mapping: We constructed tensor index files implementing <name, gpu_id, offset, size> tuple mappings. The offset values are aligned to memory word size boundaries, enabling rapid tensor localization and recovery through direct physical address calculation.

I/O Optimization. We employed strategies consistent with literature [5] to avoid redundant copying in Linux kernel or CUDA kernel spaces. Techniques like direct I/O and pinned memory were adopted to minimize I/O copy operations, maximizing bandwidth utilization across all PCIe hierarchy levels.

Lock-free Queue. Multiple queue structures (e.g., task queues and event-loop queues) were implemented in the transmission architecture. Given the frequent operations like queue length queries and element push or pop, high-speed lock-free queues were deployed. This design leverages atomic operations for queue management, effectively eliminating thread suspension issues caused by lock contention.

Memory Management. We implemented a unified memory management framework for heterogeneous memory devices using an abstract pointer with a <handle, offset> structure. This design enables cross-device addressing through handle-offset combinations and incorporates the memkind [38] framework to construct storage-adaptive memory layers. The implementation workflow for SSD devices proceeds as follows: (1) Acquire a file descriptor via the `open` system call, and then pap memory regions using `mmap` to allocate cache space and obtain the base address; (2) Initialize the memkind memory manager with this base address for subsequent memory allocation; (3) Derive file offsets by calculating the difference between memkind-allocated addresses and the base address; (4) Finally, execute data I/O operations on specified SSD regions through `pread`/`pwrite` system calls using computed <handle, offset> pairs.

Event Notification Mechanism. In our code implementation, an event-driven mechanism has been adopted to establish an efficient asynchronous programming paradigm. The specific implementations are structured as follows:

- Transmission Events: We employ a hybrid approach combining condition variables with lock-free queues to implement event notifications for task queues, as elaborated in Sect. 3.1. This design ensures that when tasks are enqueued, the condition variable strategically notifies waiting threads, thereby minimizing frequent accesses to the lock-free queue while maintaining thread synchronization.
- Stealing Events: Upon reaching a predefined saturation threshold in the local task queue of an individual transport module, the system activates a conditional variable broadcast mechanism. This broadcast initiates task-stealing events, signaling other available transport modules to execute work-stealing operations. This threshold-driven approach effectively balances workload distribution across parallel processing units.

Precomputation. We further enhance system efficiency through precomputation and cached results, which manifests in two aspects: (1) During the initialization phase, each actor precomputes the affinity distances of other actors to generate a global matrix. A hash table is employed to cache the transport set, which contains the nearest affinity distances to all destination actors. This design allows Algorithm 1 to efficiently query and compare cached transport sets via atomic operations during execution, ultimately selecting the least busy transport in the cached transport set. Benefiting from the limited number of channels, this operation completes at nanosecond-level granularity; (2) As illustrated in Fig. 5, we pre-label stealable task transport combinations based on the precomputed affinity distance matrix. For instance, when the CPU's transport CPU-GPU0 forms a minimal loop with its transport CPU-GPU1 and GPU1's transport GPU1-GPU0, the latter transport CPU-GPU1 is marked as capable of stealing tasks from the former transport CPU-GPU0. Subsequent stealing operations are implemented through atomic fetches of elements in lock-free queues, enabling highly efficient runtime execution.

Optimal Buffer Configuration. As demonstrated in Fig. 2, each transport requires two buffers to facilitate concurrent data operations: one buffer for immediate data copying and another buffer allocated to the subsequent actor for downstream data transfers. This design necessitates configuring the ring buffer length of each actor to twice the number of its associated transports to ensure seamless pipeline execution. For example, given a buffer block size of 16 MB, a CPU actor with 4 PCIe channels would require memory allocation as follows:

$$4(\text{transports}) \times 2\,(\text{buffers}) \times 16\,\text{MB} = 128\,\text{MB}. \tag{2}$$

This multiplicative approach guarantees sufficient memory provisioning for overlapping data transfers, minimizing pipeline stalls under full utilization.

The overall software architecture of our implementation is illustrated in Fig. 7.

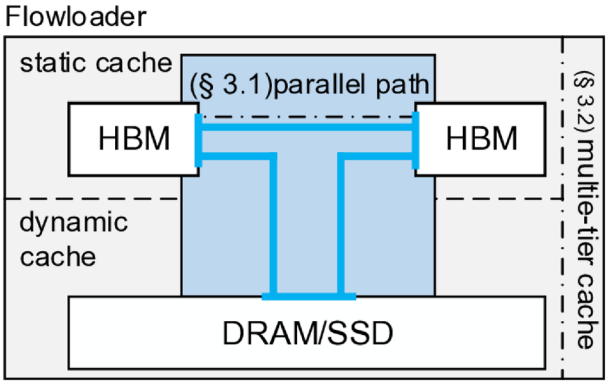


Fig. 7 The architecture diagram of Flowloader

5 Evaluation

In this section, we comprehensively evaluate the model loading speed of FlowLoader (FL) across various scenarios and conduct a comparative analysis with state-of-the-art systems. The PyTorch [15] framework is selected as a baseline due to its status as the most widely adopted deep learning framework, though its synchronous I/O operations during model loading often result in suboptimal speed performance. For advanced comparison, we choose Serverless LLM (SLLM) [5] as it pioneers a meticulously designed model loader that implements asynchronous pipeline techniques. Notably, SLLM achieves $10-200\times$ acceleration over PyTorch through its asynchronous model loading architecture.

5.1 Experimental setup

We validate our tests on a GPU server equipped with 4 H800 GPUs (interconnected via NVSwitch with NVLink 3.0 x8). The hardware configuration includes: dual-socket AMD EPY^{TM} Rome CPUs (3 groups of Xgm interconnects at 18 GT/s maximum), 1 TB DDR4 memory, and 2 TB PCIe 3.0 x4 SSD. The communication topology of the test server is shown in Fig. 8.

Based on the network topology of device interconnections, the generated affinity distance matrix is depicted in Fig. 9. This matrix provides a topology-aware foundation for subsequent transmission path optimization. The test models cover parameter scales ranging from 7B to 32B, with detailed specifications listed in Table 1.

5.2 Data transmission performance evaluation

This section evaluates the data transmission efficiency of FlowLoader. In Sect. 5.2.1, we validate the effectiveness of FlowLoader's NUMA-aware affinity strategy and

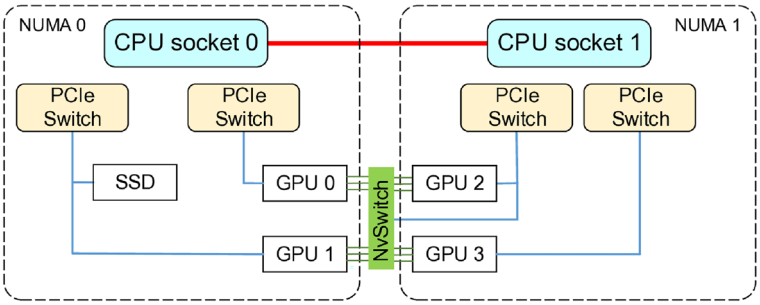


Fig. 8 Server communication topology

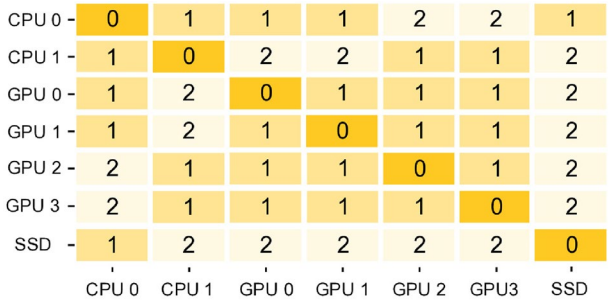


Fig. 9 Affinity distance matrix

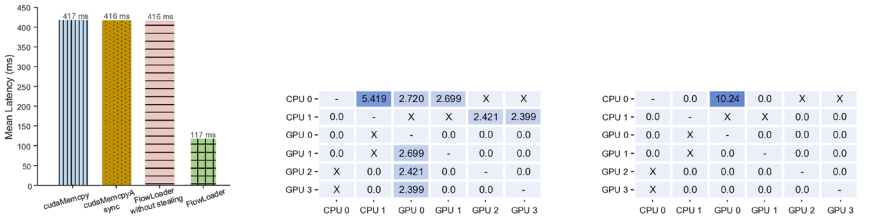
Table 1 Model Size and GPU Requirements for Deploying

Model	Size (GB)	GPUs required
Qwen2-1.5B	3	1
Qwen2-7B	14	1
Qwen1.5-14B	28	1
Qwen1.5-14B	28	2
Qwen1.5-32B	64	2
Qwen2.5-72B	144	4

task-stealing mechanism by comparing cross-NUMA-node and local-NUMA-node transmission experiments. Section 5.2.2 further assesses its load-balancing capability in multi-concurrent transmission scenarios.

5.2.1 Single-point transmission performance analysis

Local-NUMA-Node Transmission. Figure 10a shows the latency of data transmission from CPU 0 to GPU 0 within the same NUMA node. FlowLoader reduces latency by approximately 300 ms compared to the cuda memcpy APIs, achieving a

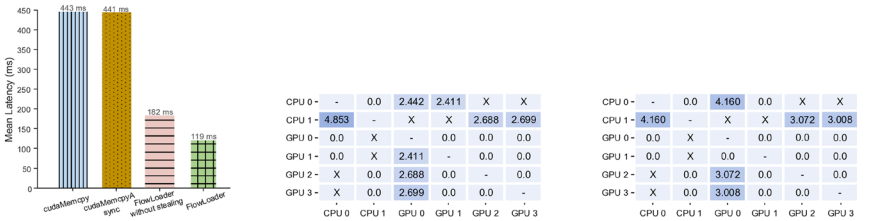


(a) Mean latency (b) Channel traffic(GB) with task-stealing (c) Channel traffic(GB) without task-stealing

Fig. 10 Local-NUMA transmission

3.6× efficiency improvement. Figure 10b details the traffic distribution across transmission channels in this experiment. By leveraging idle PCIe lanes from unused GPUs, FlowLoader expands single-channel transmission into four parallel channels. After GPUs 1-3 complete data stealing, pipelined transmission to GPU 0 occurs via NVLink. When the task-stealing functionality is disabled, transmission reverts to single PCIe channel mode (Fig. 10c), exhibiting latency comparable to cuda memcpy APIs.

Cross-NUMA Node Transmission. In the cross-NUMA node experiment (CPU 1 to GPU 0), FlowLoader reduces latency by 330 ms compared to cuda memcpy APIs, achieving a 3.8× efficiency gain (Fig. 11a). In this setup, GPUs $2-3\times$ under NUMA node 1 handle approximately half of the traffic, subsequently transferring data to GPU 0 via NVLink. Additionally, the pipeline mechanism masks cross-NUMA read latency, while the higher bandwidth of UPI (compared to direct CPU-GPU connections) enables CPU 0 to handle 50% of the total traffic, with GPU 1 sharing another 50% of CPU 0's load (Fig. 11b). When task-stealing is disabled (Fig. 11c), the affinity domain algorithm provides only three transmission channels. However, due to CPU 0's 40% traffic burden and single PCIe channel limitation, latency decreases to 182 ms, yielding 2.3× efficiency over cuda memcpy APIs. Notably, FlowLoader maintains comparable latency in cross-NUMA scenarios to local transmissions, while cuda memcpy APIs exhibits a 26 ms latency increase due to cross-node access overhead.



(a) Mean latency (b) Channel traffic(GB) with task-stealing (c) Channel traffic(GB) without task-stealing

Fig. 11 Cross-NUMA transmission

5.2.2 Concurrent transmission performance analysis

GPU Gather Transmission Test. Figure 12a compares the latency when GPUs 1-3 concurrently transmit data to GPU 0. cuda's memcpy requires 172 ms due to NVLink bandwidth contention on GPU 0 via NVSwitch. FlowLoader reduces total latency to 154 ms (16% efficiency gain) by offloading 2.57 GB of traffic to idle PCIe parallel links (Fig. 12b), demonstrating adaptive load balancing and full channel utilization under concurrent contention.

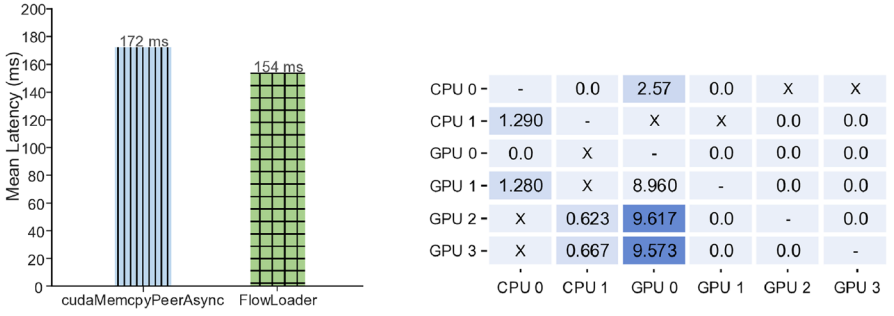
CPU-GPU Concurrent Transmission Test. Figure 13 presents the results of concurrent data transfers from CPU to GPU. Figure 13a shows the comparison between FlowLoader and the cuda memcpy API for two concurrent parallel channels: CPU 0 to GPU 0 and CPU 1 to GPU 2. Figure 13b displays the individual throughput of each channel within the two concurrent CPU-to-GPU transfers when using FlowLoader. Figure 13c shows concurrent four-channel transfers (GPU 0 to GPU 0-1 and CPU 1 to GPU 2-3), and Fig. 13d displays the throughput of each channel under four concurrent transfers using FlowLoader. Comparing with Sect 5.2.1 reveals that FlowLoader's transfer time increases linearly as more CPU-to-GPU channels become occupied. This is due to FlowLoader's ability to leverage idle parallel transfer links. When the upstream and downstream paths of the transmission links are saturated, FlowLoader cannot access additional transfer channels, thus failing to accelerate the transfer.

5.3 Model loading

5.3.1 Initial loading

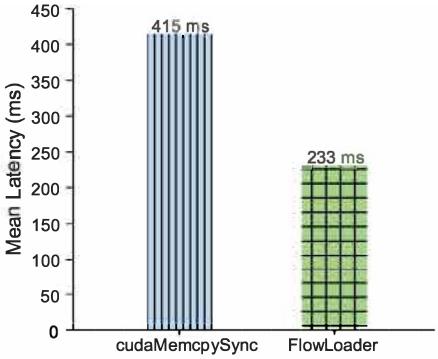
Using a 2 TB PCIe 3.0 x4 SSD (3.0 GB/s bandwidth) as the loading source, we compare FlowLoader against PyTorch and SLLM baselines. The average latency across multiple model loading instances (with system cache cleared each time) is shown in Fig. 14.

Compared with PyTorch, FlowLoader and SLLM demonstrate significant optimizations in model loading efficiency. This stems from PyTorch's paradigm requiring



(a) Mean latency (b) Channel traffic(GB)

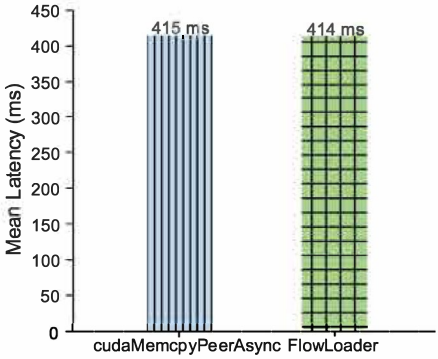
Fig. 12 GPUs gather transmission



(a) 2 CPU-GPU Parallel Transmission's latency

	CPU 0	CPU 1	GPU 0	GPU 1	GPU 2	GPU 3
CPU 0	-	0.0	0.54	0.48	X	X
CPU 1	0.0	-	X	X	0.55	0.47
GPU 0	0.0	X	-	0.0	0.0	0.0
GPU 1	0.0	X	0.48	-	0.0	0.0
GPU 2	X	0.0	0.0	0.0	-	0.0
GPU 3	X	0.0	0.0	0.0	0.47	-

(b) 2 CPU-GPU Parallel Transmission's Channel traffic(GB)



(c) 4 CPU-GPU Parallel Transmission's latency

	CPU 0	CPU 1	GPU 0	GPU 1	GPU 2	GPU 3
CPU 0	-	0.0	10.24	10.24	X	X
CPU 1	0.0	-	X	X	10.24	10.24
GPU 0	0.0	X	-	0.0	0.0	0.0
GPU 1	0.0	X	0.0	-	0.0	0.0
GPU 2	X	0.0	0.0	0.0	-	0.0
GPU 3	X	0.0	0.0	0.0	0.0	-

(d) 4 CPU-GPU Parallel Transmission's Channel traffic(GB)

Fig. 13 CPU-GPU concurrent transmission

complete model architecture definition, which typically necessitates CPU-side deserialization operations during loading. Particularly in parallel computing scenarios (such as tensor parallelism or pipeline parallelism), explicit tensor partitioning operations must first be performed on CPU memory before transferring to GPU memory. This multi-stage workflow inherently introduces sequential synchronization overhead: Parameters must be instantiated in CPU memory, then partitioned, and finally transmitted to corresponding GPU devices. Such conventional methodology incurs redundant computational costs. In contrast, FlowLoader adopts SLLM's optimized GPU-native persistence strategy via pre-serialization. This strategy uses a memory word size-aligned block-sequential storage format by capturing precise snapshots of inter-GPU tensor distributions and maintaining a dedicated tensor index file (mapping logical tensor identifiers to physical GPU memory addresses, including device IDs, offsets, and sizes). This enables the system to directly transfer data into GPU memory space without requiring architecture reconstruction or deserialization.

In terms of data transmission mechanisms, both FlowLoader and SLLM employ pipeline technology to mitigate multi-level transfer latency. However, FlowLoader's

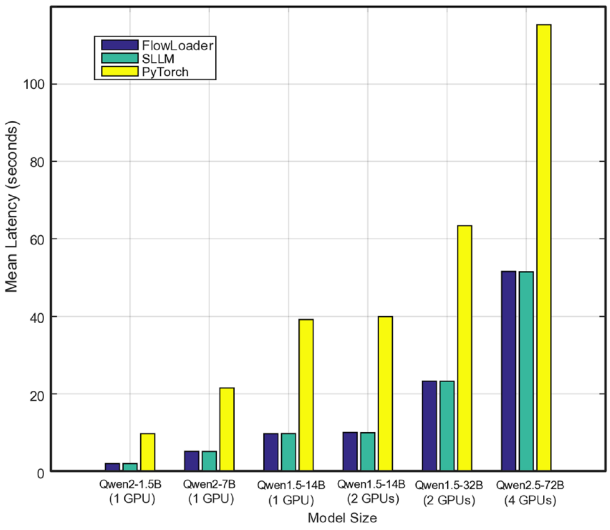
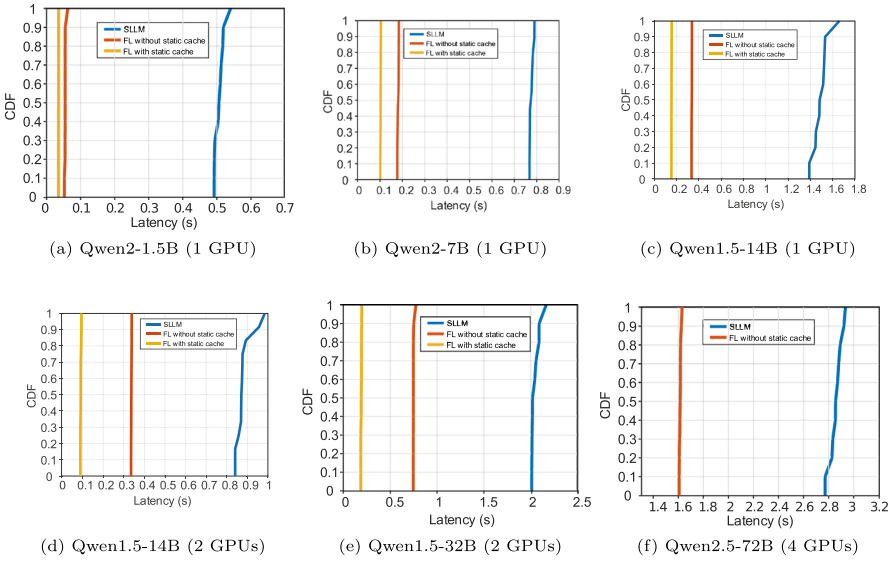


Fig. 14 First-time loading latency

unique parallel path architecture and dynamic task scheduling mechanism fail to fully realize their theoretical advantages due to SSD bandwidth constraints (limited to 3 GB/s). Notably, the single PCIe channel bandwidth between CPU and GPU (typically exceeding 15.75 GB/s) far surpasses the maximum throughput capability of current SSD devices, rendering FlowLoader's multi-PCIe-channel-based parallel transmission design ineffective under existing hardware configurations. Experimental data demonstrates that by adopting SLLM's direct I/O access technology, both FlowLoader and SLLM successfully achieve stable SSD read bandwidth at the theoretical peak level (3 GB/s). Leveraging efficient pipeline architecture design, their model loading efficiency improves by $2.7 - 4\times$ compared to the PyTorch baseline, with actual time consumption aligning closely with mathematically calculated values derived from SSD maximum bandwidth.

5.3.2 Cache preheating

Load Speed Across Different Cache Levels. Under cache-available scenarios, FlowLoader utilizes a multi-level caching system, while SLLM employs pinned memory to provide a CPU-managed cache. In this test, we first loaded all data onto their respective cache layers. Subsequently, multiple repeated loading iterations were performed to obtain the cumulative distribution function (CDF) graph of loading times for each model across the different cache layers, as shown in Fig. 15. SLLM improves loading speed by $6 - 11\times$ over SSD-initiated loads via strategies like chunked multi-stage pipeline loading. For Qwen1.5-14B deployed on two GPUs, SLLM achieves $1.8\times$ speedup over single-GPU setups by leveraging dual PCIe channels. FlowLoader outperforms SLLM by $2.8 - 4\times$ without static caching through global CPU-GPU PCIe channel scheduling, and further improves to $7.3 - 10\times$ with static caching enabled.



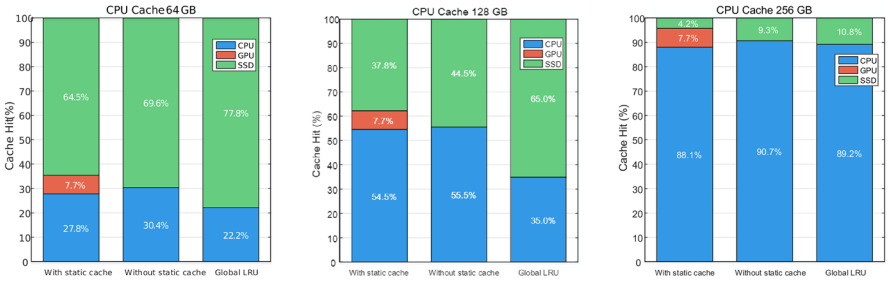
(a) Qwen2-1.5B (1 GPU) (b) Qwen2-7B (1 GPU) (c) Qwen1.5-14B (1 GPU)

(d) Qwen1.5-14B (2 GPUs) (e) Qwen1.5-32B (2 GPUs) (f) Qwen2.5-72B (4 GPUs)

Fig. 15 CDF of cache-enabled loading latency

Impact of Different CPU Cache Capacities. Since CPU cache offers high-speed data access, it significantly influences model loading speed. In this experiment, we evaluated the impact of CPU cache capacity by setting it to 64 GB, 128 GB, and 256 GB, while maintaining a fixed SSD cache size of 512 GB. To demonstrate the effectiveness of the multi-layer caching system, we compared the Flowloader tiered caching system against two baselines: (1) a system with GPU static caching disabled (using only CPU and SSD), and (2) a global LRU unified memory management strategy (using memkind to manage SSD and DRAM memory, prioritizing DRAM). Prior to testing, the cache systems were warmed up using 1000 randomly generated loading requests for the models listed in Table 1. Given the limited resources of the test server (only 4 GPUs), when GPU resources were exhausted, previously loaded models were unloaded to accommodate new ones. Subsequently, 100 randomly generated access sequences were used to measure cache hit rates. The warm-up and test request sequences remained consistent across all experiments.

As shown in Fig. 16, enabling static caching resulted in approximately 7.7% of requests hitting the GPU cache. This allows the system to leverage the faster GPU cache instead of CPU cache, thereby increasing transfer speed. Due to the consistent request sequences, GPU static caching consistently provided this 7.7% hit rate across different CPU cache capacities. While enabling static caching slightly reduced CPU cache hit rates, the combined hit rate of the high-speed GPU and CPU caches was 5.1% to 6.7% higher than using dynamic caching alone. Compared to the global LRU strategy, the tiered caching system achieved a 6.4% higher high-speed cache hit rate under the 256 GB CPU cache condition. This performance gap occurs because the global LRU strategy fails to effectively distinguish between cache tiers. Its ability to utilize faster cache tiers degrades significantly as CPU cache capacity

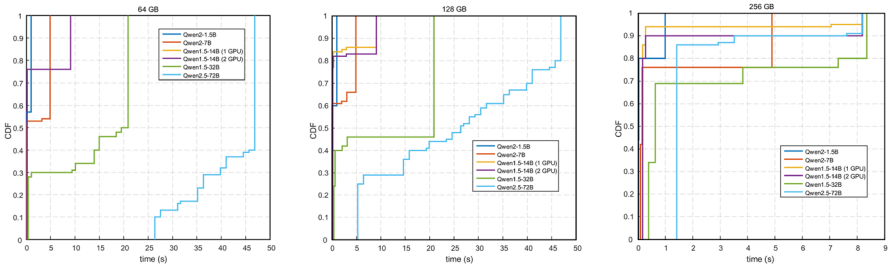


(a) 64 GB (b) 128 GB (c) 256 GB

Fig. 16 Cache hit rate under different CPU cache capacities

decreases: At 64 GB, its high-speed cache hit rate was approximately 15.9% lower than the tiered system with static caching enabled and 8.2% lower than the dynamic-only configuration; at 128 GB, it was 20.5% 27.2% lower than the tiered caching system.

The loading time CDFs of different models under these three CPU cache capacities are presented in Table 1 and Fig.17. As shown in Fig.17, increasing the CPU cache capacity significantly reduces model loading time. When the CPU cache capacity is only 64 GB, it is insufficient to hold an entire 72B model. Consequently, the loading speed bottleneck under this condition is primarily constrained by SSD bandwidth. However, due to the acceleration effect of the static caching layer, models of sizes other than 72B can still benefit from the high-speed GPU cache even with a 64 GB CPU cache. When the cache size increases to 256 GB, nearly all model layers can be accommodated within the CPU cache. This configuration yields a substantial acceleration effect, reducing the worst-case loading time by approximately a factor of 5.6 compared to conditions with 64 GB or 128 GB CPU caches. As evident from Fig.17c, although most model layers can reside in the CPU cache at this size, the significant bottleneck effect of the SSD caching layer means that the final loading time may still depend heavily on the data read latency from the SSD.



(a) 64 GB (b) 128 GB (c) 256 GB

Fig. 17 CDF of Model Loading Time under Different CPU Cache Capacities

6 Conclusion

This paper introduces LLM FlowLoader, an efficient model loader designed to address the cold-start problem in large language models. LLM FlowLoader provides novel optimization insights for large model systems, particularly in terms of loading speed and resource utilization efficiency. By introducing the concept of affinity domains and implementing an asynchronous pipeline transporter that supports topological networks, we successfully simplify complex network transmission into more efficient chain-like transmission. Concurrently, our task-stealing algorithm adaptively leverages parallel channels to significantly enhance I/O bandwidth. Furthermore, we designed an efficient multi-tier caching system that fully utilizes the multi-level memory and storage devices of GPU servers, further accelerating data access and reducing cold-start time. Experiments demonstrate that LLM FlowLoader achieves $2.8 - 10\times$ faster startup speeds compared to state-of-the-art loaders.

Our experimental results show that LLM FlowLoader achieves remarkable optimization in loading performance. We believe this research will play a significant role in practical applications of large models while laying the foundation for future related research. Looking ahead, we aim to extend LLM FlowLoader to support more complex distributed server scenarios, and explore intelligent caching strategies and resource scheduling possibilities.

Acknowledgements The authors sincerely thank Zhejiang Dahua Technology Co., Ltd. for providing product data and supporting the consistency verification experiment of the H800. In addition, we are particularly grateful to Huang Peng for his support and assistance in this project.

Author contributions All authors contributed equally to this work.

Funding This research received no external funding.

Declarations

Conflict of interest All authors disclosed no relevant relationships.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

References

1. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan JD, et al. (2020). Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems

2. Narayanan D, Shoeybi M, Casper J, LeGresley P, et al. (2021) Efficient large-scale language model training on gpu clusters using megatron-lm. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis
3. Touvron H, Martin L, Stone K, Albert P, et al. (2023) Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv: https://arxiv.org/abs/2307.09288
4. Crankshaw D, Sela G, Mo X, Zumar C, et al. (2020) Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In: SoCC '20: Proceedings of the 11th ACM Symposium on Cloud Computing. https://par.nsf.gov/biblio/10245792
5. Fu Y, Xue L, Huang Y, Brabete A-O, Ustiugov D, Patel Y, Mai L (2024) ServerlessLLM: low-Latency serverless inference for large language models. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, pp. 135–153
6. Kojs K (2023) A survey of Serverless machine learning model inference. arXiv: https://arxiv.org/abs/2311.13587
7. Dakkak A, Li C, Gonzalo S, Xiong J, Hwu W-m (2019) Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 372–382
8. Gujarati A, Karimi R, Alzayat S, Hao W, Kaufmann A, Vigfusson Y, Mace J (2020) Serving DNNs like clockwork: performance predictability from the bottom up. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 443–462
9. Zou D, Jin X, Yu X, Zhang H, Demmel J (2023) Computron: serving distributed deep learning models with model parallel swapping. arXiv: https://arxiv.org/abs/2306.13835
10. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778
11. Li Z, Zheng L, Zhong Y, Liu V, 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), pp. 663–679
12. Miao X, Shi C, Duan J, Xi X, Lin D, Cui B, Jia Z (2024) Spotserve: serving generative large language models on preemptible instances. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 1112–1127
13. Amazon Web Services: AWS Lambda Developer Guide. Accessed: 2025-03-03. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
14. Baldini I, Castro P, Chang K, Cheng P, et al. (2017) Serverless computing: current trends and open problems, pp. 1–20
15. PyTorch: Torch.load. Accessed: 2025-03-03. https://pytorch.org/docs/stable/generated/torch.load.html
16. HuggingFace: Safetensors: ML Safer for All. Accessed: 2025-03-03. https://github.com/huggingface/safetensors
17. Li A, Song SL, Chen J, Li J, Liu X, Tallent NR, Barker KJ (2020) Evaluating modern GPU interconnect: PCIe, NVLink, nv-sli, NVSwitch and gpudirect. IEEE Trans Parallel Distrib Syst 31(1):94–110
18. Lutz C, Breß S, Zeuch S, Rabl T, Markl V (2020) Pump up the volume: processing large data on gpus with fast interconnects, pp. 1633–1649
19. AMD: AMD EPYC 7002 Series Processors. Accessed: 2025-03-03. https://www.amd.com/en/products/processors/server/epyc/7002-series.html
20. Xu Q, Siyamwala H, Ghosh M, Suri T, Awasthi M, Guz Z, Shayesteh A, Balakrishnan V (2015) Performance analysis of NVMe SSDs and their implication on real world databases. In: Proceedings of the 8th ACM International Systems and Storage Conference
21. Li S, Reddy D, Jacob B (2018) A performance & power comparison of modern high-speed dram architectures, pp. 341–353
22. Qwen Team: Qwen2.5 Technical Report. Accessed: 2025-03-03. https://huggingface.co/Qwen
23. Chan E, Heimlich M, Purkayastha A, Geijn R (2007) Collective communication: theory, practice, and experience. Concurr Comput Pract Exp 19(13):1749–1783
24. Cho JY, Seo PR, Jin HW (2023) Exploiting copy engines for intra-node MPI collective communication. J Supercomput 79:17962–17982
25. Agarwal N, Nellans D, O'Connor M, Keckler SW, Wenisch TF (2015) Unlocking bandwidth for gpus in cc-numa systems. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 354–365

26. Hewitt C (2015) Actor Model of Computation: Scalable Robust Information Systems. arXiv: https://arxiv.org/abs/1008.1459
27. Juan JS, Wong B (2023) Reducing the cost of GPU cold starts in serverless deep learning inference serving. In: 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops), pp. 225–230
28. Ahuja RK, Kodialam M, Mishra AK, Orlin JB (1997) Computational investigations of maximum flow algorithms. Eur J Oper Res 97(3):509–542
29. Jensen PM, Jeppesen N, Dahl AB, Dahl VA (2023) Review of serial and parallel min-cut/max-flow algorithms for computer vision. IEEE Trans Pattern Anal Mach Intell 45(2):2310–2329
30. Davies S, Moseley B, Vassilvitskii S, Wang Y (2023) Predictive flows for faster ford-fulkerson. In: Proceedings of the 40th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 202, pp. 7231–7248
31. Rajalakshmi V, Ganesh Vaidyanathan S (2019) Efficient traffic management on road network using Edmonds–Karp algorithm. In: Progress in Advanced Computing and Intelligent Engineering, pp. 577–583
32. Chen L, Asai K, Nonomura T, Xi G, Liu T (2018) A review of backward-facing step (BFS) flow mechanisms, heat transfer and control. Ther Sci Eng Prog 6:194–216
33. Vuppalapati M, Agarwal R (2024) Tiered memory management: access latency is the key! In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, pp. 79–94
34. NVIDIA: Introduction to NVIDIA DGX A100 - User Guide. Accessed: 2025-03-03. https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html
35. Wei J, Zhang X, Wang L, Wei Z (2023) Fastensor: optimise the tensor i/o path from SSD to GPU for deep learning training. ACM Trans Archit Code Optim. https://doi.org/10.1145/3630108
36. Jeong J, Baek S, Ahn J (2023) Fast and efficient model serving using multi-GPUs with direct-host-access. In: Proceedings of the Eighteenth European Conference on Computer Systems, pp. 249–265
37. Wang L, Ding G, Zhao Y, Wu D, He C (2017) Optimization of leveldb by separating key and value. In: 2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 421–428
38. Cantalupo C, Venkatesan V, Hammond J, Czurlyo K, Hammond SD (2015) Memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies. https://www.osti.gov/biblio/1245908

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Jun Yin[1] · Fei Wu[1] · Shihao Li[2] · Peng Huang[2]

✉ Shihao Li
aixxzsthl@gmail.com

Jun Yin
12221242@zju.edu.cn

Fei Wu
wufei@zju.edu.cn

Peng Huang
huang_peng@dahuatech.com

[1] Zhejiang University, Hangzhou 310058, China

[2] Zhejiang Dahua Technology Co., Ltd., Hangzhou 310053, China