Efficient Data Passing for Serverless Inference Workflows: A GPU-Centric Approach

Hao Wu[1,2*], Yaochen Liu[1], Minchen Yu[3], Qizhen Weng[4], Junxiao Deng[1], Yue Yu[1], Hao Fan[1], Song Wu[1], Wei Wang[2], and Hai Jin[1]

[1] National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China

[2] Hong Kong University of Science and Technology, Hong Kong, China

[3] The Chinese University of Hong Kong (Shenzhen), Shenzhen, China

[4] Institute of Artificial Intelligence (TeleAI), China Telecom, China

{wuhao5,u202115348}@hust.edu.cn,yuminchen@cuhk.edu.cn,wengqzh@chinatelecom.cn
{dengjunxiao,yuyue18,haofan,wusong}@hust.edu.cn,weiwa@cse.ust.hk,hjin@hust.edu.cn

Abstract

Serverless computing offers a compelling paradigm for deploying machine learning inference workflows composed of heterogeneous CPU and GPU functions. However, existing data-passing solutions in serverless systems primarily rely on host memory for data exchange (host-centric), leading to substantial data movement and salient I/O overhead. Moreover, modern GPU communication libraries (e.g., NCCL, NVSHMEM, UCX) are ill-suited to serverless environments, suffering from redundant data copies, underutilized transfer bandwidth, and inefficient temporary GPU storage.

In this paper, we present GRouter, a GPU-centric data plane system designed for serverless inference workflows. GRouter first introduces a unified data passing framework that abstracts host-to-GPU and GPU-to-GPU communication while leveraging function placement to reduce redundant copies. It then aggregates available bandwidth across PCIe links, NVLinks, and NICs to enable parallel transfers with performance isolation between functions. GRouter also implements elastic GPU storage that adapts to idle memory availability and varying data transfer demands. Evaluations on real-world inference services show that GRouter reduces data passing latency by up to 87% and improves throughput by up to 1.74× compared to state-of-the-art GPU communication libraries.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

ACM Reference Format:
Hao Wu, Yaochen Liu, Minchen Yu, Qizhen Weng, Junxiao Deng, Yue Yu, Hao Fan, Song Wu, Wei Wang, and Hai Jin. 2026. Efficient Data Passing for Serverless Inference Workflows: A GPU-Centric Approach. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3769336

1 Introduction

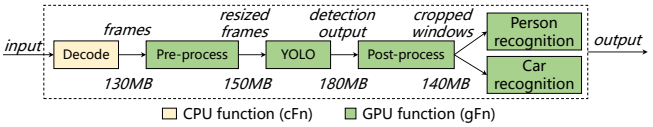

Figure 1. A serverless workflow for traffic monitoring

The rapid advances of *Machine Learning* (ML) and its widespread adoption have driven a growing demand for scalable, cost-effective ML inference services in the cloud [7, 13, 15, 53]. Serverless computing has emerged as a promising paradigm for inference serving. It enables users to deploy ML models as stateless functions while offloading resource provisioning and scaling to the cloud platform [4, 19, 37, 47, 48, 50, 52]. It is also economically attractive as users are only billed for the resources consumed during actual function execution. This pay-per-use billing makes serverless inference particularly suitable for workloads with intermittent or unpredictable traffic patterns [10, 50, 51].

Cloud-based inference services typically comprise complex workflows that orchestrate GPU-accelerated ML model executions alongside CPU-based data processing operations [3, 8, 11, 14]. Fig. 1 illustrates a real-world traffic monitoring application [40], where video frames are first decoded and preprocessed, followed by object detection using a YOLO model; cropped images of pedestrians and vehicles are then routed to specialized recognition models for behavior and type analysis. These components—running in loosely coupled GPU and CPU functions—are stitched together into a unified serverless inference workflow.

Unlike traditional CPU-based function workflows [21, 26, 49], serverless inference involves a mix of *GPU functions* (gFns) and *CPU functions* (cFns), where data exchanges can occur between CPU functions (cFn-cFn), GPU functions (gFn-gFn), or between GPU functions and the host system (gFn-host)—in the latter case, GPU functions interact with CPU functions running in the host or *Input and Output* (I/O) via the host-side in-memory store. Developing an efficient

[1]Work done during an internship at HKUST and TeleAI.

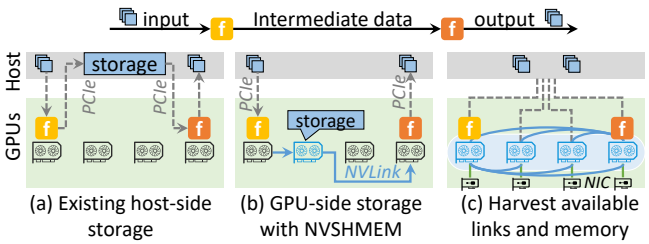

Figure 2. Three data passing approaches for serverless inference system: Host-centric (latest), GPU-enabled (integrated with NVSHMEM), and GPU-centric (our method)

serverless data plane to streamline these exchanges is therefore crucial for accelerating end-to-end inference workflows.

Existing serverless systems employ a host-centric approach for data exchange between functions [20, 21, 23, 26, 49, 50], where intermediate data is stored in an external storage—deployed on the local or a remote host—before being consumed by downstream functions. However, as illustrated in Fig. 2(a), this approach creates an elongated data path with frequent data copies between GPU devices and the host, introducing significant delays in end-to-end workflow execution (up to 92% in our experiment).

To avoid moving data through the slow host-side storage, modern GPU communication libraries, such as NCCL [27], UCX [43], and NVSHMEM [32], provide support for direct communications across GPUs via high-speed interconnects such as NVLink or *GPU Direct RDMA* (GDR) [31]. These libraries enable a GPU-side storage solution to accelerate data exchange. For instance, with NVSHMEM, GPU functions can directly store and retrieve intermediate data within a shared GPU memory space, bypassing host memory (Fig. 2(b)).

However, this approach fails to achieve optimal performance because existing GPU communication libraries are not designed for serverless environments, resulting in three major limitations. (1) *Redundant data copies.* In serverless inference, GPU storage is typically deployed as a decoupled service from function execution, rendering it agnostic to function placement. Without knowledge of where functions are instantiated, the storage cannot prioritize data locality. This forces intermediate data to traverse non-local paths, incurring unnecessary duplication. As shown in Fig. 2(b), the output data of the upstream function is first copied to a GPU store on a remote device and then transferred again to the GPU where the downstream function is located—doubling data movement overhead. (2) *Inefficient bandwidth utilization.* GPU clusters employ heterogeneous interconnects: high-bandwidth NVLinks and lower-bandwidth PCIe links within servers, and *Network Interface Cards* (NICs) across servers. An efficient serverless data plane should leverage these asymmetric links for concurrent data transfers, aggregating available bandwidth between GPU functions. However, existing GPU libraries restrict point-to-point communication to a single path (e.g., NVLink-only), leaving multi-link bandwidth harvesting untapped. (3) *Lack of elastic memory management.* During inference workflow execution, intermediate data must be temporarily stored in GPU memory. While serverless systems inherently exhibit dynamic workloads and on-demand function provisioning—which often leave idle GPU memory available—this availability changes unpredictably. Elastic GPU memory management, capable of dynamically scaling allocations in response to runtime demands, is thus critical. Yet, existing GPU libraries lack this capability, resulting in memory contention and performance degradation during traffic spikes.

To address these challenges, we propose GRouter, a *GPU-centric* data plane system designed for efficient data exchange in serverless inference workflows. Unlike conventional host-centric approaches, GRouter explicitly leverages knowledge of GPU topology and function placement to orchestrate concurrent data transfers across multiple links (e.g., NVLink, PCIe links, and NICs), aggregating available bandwidth and memory resources across the GPU cluster. The design of GRouter comprises four key components. (1) *Unified data passing framework.* GRouter introduces a programming interface that abstracts heterogeneous data-passing patterns (e.g., gFn-gFn, gFn-host). Internally, it dynamically detects function placement and underlying GPU server topology to enable transparent, locality-aware data transfers and storage management, eliminating redundant copies. (2) *Fine-grained bandwidth harvesting.* To fully utilize cluster bandwidth, GRouter enables multi-path data transfers by partitioning and allocating idle GPU links (including NVLinks, PCIe links, and NICs), aggregating available bandwidth while preventing resource contention among concurrent functions. (3) *Topology-aware transfer scheduling.* For asymmetric GPU topologies, GRouter strategically selects *assist GPUs* with optimal NVLink connectivity to target GPUs running inference functions. It further exploits idle parallel NVLink paths for point-to-point data transfers, achieving near-peak throughput. (4) *Elastic data storage.* GRouter dynamically scales GPU memory allocations by monitoring real-time storage demands and memory pressure. When a GPU device has no enough memory to hold all storage data, it migrates low-priority data to other idle GPUs or host memory while retaining critical data (e.g., for upcoming high-priority functions) in GPU memory, minimizing performance penalties from host memory evictions.

We implement GRouter as an extension to INFless [48], a state-of-the-art serverless inference system, utilizing low-level GPU *Inter-Process Communication* (IPC) mechanism for direct data transfers between functions and GPU storage. Our evaluation benchmarks GRouter against two baselines: conventional host-centric serverless systems and NVSHMEM-enhanced systems optimized for GPU communication. Using real-world inference workflows and production request

traces from Azure cloud [39], we show that GRouter reduces data transfer overhead by up to 65% and achieves 11× higher throughput than the best-performing baseline. We also demonstrate the scalability and effectiveness of GRouter in LLM inference applications and large clusters.

2 Background

2.1 Serverless Inference Workflow

Cloud-based ML inference services have increasingly turned to serverless technology to streamline the deployment of the serving pipeline [10, 37, 47, 48, 50, 52, 54]. Serverless inference allows users to deploy ML models and data processing operations as stateless functions and lets the platform to handle resource provisioning, autoscaling, logging, fault-tolerance, and other infrastructure management tasks. Users are only billed when functions are running, eliminating the cost of idle resources.

Modern inference services typically orchestrate multi-stage workflows that integrate ML models with data processing operations. Fig. 1 illustrates this with a real-world traffic monitoring application [40], which comprises one CPU function for video decoding and five GPU functions for pre-processing, object detection, post-processing, and person and car recognition. These heterogeneous functions are loosely coupled, composing a serverless inference workflow. The diversity of these workflows is further demonstrated in Fig. 12, which collects a suite of real-world inference services from recent studies [3, 8, 11, 40, 54, 55]. The suite spans multiple workflow patterns, including linear sequential pipelines, conditional branching for dynamic decision-making, and fan-in/fan-out parallelism for high-throughput data distribution.

2.2 Data Passing in Serverless Inference Workflow

Compared to traditional CPU-centric function orchestration [20, 21, 26], serverless inference workflows introduce a new challenge—managing data exchanges across heterogeneous functions. These workflows involve *GPU functions* (gFns) executing ML models and *CPU functions* (cFns) handling data processing, with three distinct function interaction patterns: cFn-cFn, gFn-gFn, and gFn-host, which includes gFn-cFn or gFn-to-host-storage. Optimizing these interactions demands a purpose-built serverless data plane.

Existing serverless systems rely on external storage (e.g., remote services like AWS S3 [1] or intra-node solutions like Redis [20, 21, 26] and shared memory [16, 25, 49]) to facilitate data passing between stateless functions—an approach we call host-centric data passing. For GPU functions, however, this forces all intermediate data through host memory via PCIe links—a design ill-suited to GPU workflows. As shown in Fig. 2(a), each gFn-gFn transfer requires two PCIe copies (GPU to host to GPU).

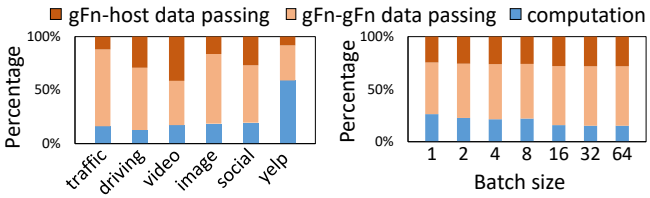

Figure 3. Performance analysis of host-centric data passing approach: (a) Breaking down of overall latency. (b) Breaking down of latency for *Traffic* workflow with various batch sizes. Each bar is broken into three parts: the latencies of gFn-host data passing (top), gFn-gFn data passing (middle), and computation (bottom)

Table 1. Limitations of GPU-side storage using existing GPU communication libraries

	Data locality	Bandwidth harvesting	Efficient temporary storage
NCCL/UCX	×	×	×
NVSHMEM	×	×	×
DeepPlan [15]	×	×	×
GRouter	✓	✓	✓

To quantify this overhead, we deploy real-world inference workflows on INFless [48], a state-of-the-art serverless inference system, using a DGX-V100 cluster with host-memory sharing [49]. As Fig. 3 illustrates, data passing accounts for 92% of end-to-end latency: 63% from gFn-gFn transfers and 29% from gFn-host interactions (see §6 for details). The PCIe-bound copies between GPUs and host memory dominate this cost, while cFn-cFn transfers via shared host memory incur negligible overhead. These results underscore the urgent need for a GPU-native data plane to eliminate host-induced bottlenecks.

3 Challenges

A simple fix to host-centric data passing is to replace the host-side storage with a GPU-side storage using modern GPU communication libraries, such as NCCL [27], NVSHMEM [32], and UCX [41]. While these libraries enable fast, direct GPU-to-GPU communication via high-speed interconnects like NVLink and *GPUDirect RDMA* (GDR) [31], their design assumptions (i.e., collective or point-to-point communication across long-running serverful processes) are not aligned with the serverless environments, leading to many limitations as summarized in Table 1.

To demonstrate these problems, we augment INFless [48], a SOTA serverless inference system, with NVSHMEM to create NVSHMEM+, a prototype implementing GPU-side storage. Using this setup, we expose fundamental challenges in adapting GPU communication libraries to serverless workflows: redundant data copies (§3.1), bandwidth underutilization (§3.2), and inefficient memory management (§3.3).

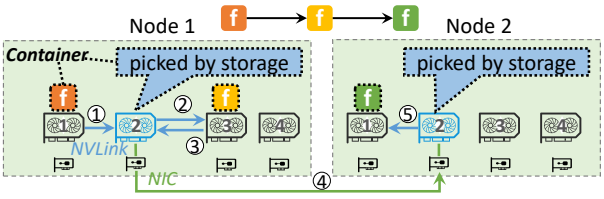

Figure 4. GPU data passing in serverless inference with NVSHMEM

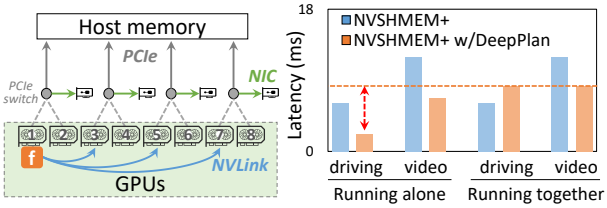

Figure 5. (a) Parallel PCIe and NIC transfers for function on GPU1. (b) Comparison of gFn-host data transfer overhead when running inference workflows alone and together.

3.1 Challenge #1: Redundant Data Copies

In serverless inference, GPU functions and data storage are deployed as decoupled services running in isolated containers, making them unable to identify their own physical location (e.g., GPU device ID). This opacity stems from two factors. First, GPU virtualization, where functions perceive virtualized device IDs (e.g., a function on physical GPU3 sees it as GPU0). Second, address mapping limitations in GPU *Inter-Process Communication* (IPC), which underpins GPU communication libraries like NVSHMEM. When a storage container retrieves GPU memory addresses via `cudaPointerGetAttributes()`, it resolves them to its own local GPU device rather than the physical GPU where the source function is located.

Without knowledge of function placement, the storage cannot provide data locality but blindly selects GPUs to store intermediate data. This results in unnecessary relay copies instead of direct transfers. Fig. 4 illustrates a chain workflow where three functions exchange data across GPUs and nodes: the first two functions (GPU1 and GPU3 on Node 1) relay data through GPU2, requiring two copies (GPU1 to GPU2 and to GPU3) instead of a direct NVLink transfer; the last two functions (GPU3 on Node 1 and GPU5 on Node 2) force data through two remote GPUs—because GPU functions can only interact with local storage on the same node—tripling copies versus a single GDR transfer. In total, NVSHMEM+ incurs 3 more data copies than the optimum scheme. This inefficiency grows rapidly with workflow complexity, as each hop introduces PCIe or NIC bandwidth contention.

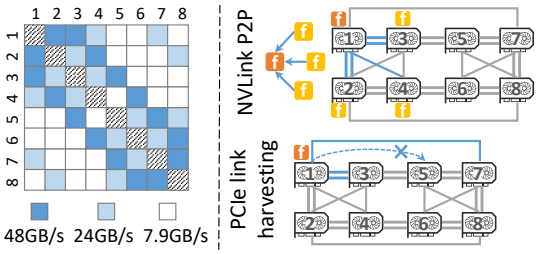

Figure 6. The asymmetric GPU topology. (a) Point-to-point bandwidth of different GPU pairs in a DGX-V100 GPU server. (b) Bandwidth constraints in asymmetric GPU topology.

3.2 Challenge #2: Underutilized Link Bandwidth

In serverless inference, functions are encapsulated in containers [28], typically limiting their access to a single GPU. Also, existing GPU communication libraries only use the transfer link dedicated to the local GPU (e.g., a single PCIe link, NIC, or NVLink connection), failing to exploit node- and cluster-wide bandwidth. By contrast, modern GPU interconnects enable bandwidth harvesting—borrowing idle links from peer GPUs for parallel transfers by three means:

Parallel PCIe transfers. Existing libraries transfer data to host memory exclusively via local GPU PCIe link, which is usually a bottleneck. As shown in Fig. 3, gFn-host transfers contribute 29% of end-to-end latency. By contrast, routing data via NVLink to peer GPUs and leveraging their PCIe links in parallel (Fig. 5) can achieve 2–4× higher aggregate bandwidth.

Parallel NIC transfers. Instead of confining cross-node transfers to the nearest NIC of the local GPU—the current practice—forwarding data via NVLink to other GPUs and utilizing their NICs in parallel (Fig. 5) enables multi-path transmission, effectively enhancing inter-node throughput.

Parallel NVLink transfers. While existing libraries use only direct NVLink paths for point-to-point transfers, the mesh topology of NVLink allows routing through intermediate GPUs to exploit parallel links. For example, a two-hop transfer across three GPUs can utilize twice the NVLink bandwidth of a single direct path.

However, realizing these optimizations in serverless systems requires two key innovations. First, bandwidth partitioning to prevent contention among concurrent functions sharing links. Second, topology-aware path selection to identify optimal parallel routes across functions.

3.2.1 Bandwidth partitioning. Harvesting transfer links from peer GPUs in multi-tenant environments can induce bandwidth contention. To demonstrate this, we evaluate two workflows from the benchmarking suite we collect (Fig. 12): the *driving* and *video* workflows. To enable concurrent PCIe transfers, we augment NVSHMEM+ with parallel data loading techniques from DeepPlan [15] (termed NVSHMEM+ w/

DeepPlan). We first run the two workflows alone. As illustrated in Fig. 5(b), transferring data over parallel PCIe links significantly reduces the gFn-host latency for both workflows. We next run the two workflows together in the same node: we observe significant interference that increases the gFn-host latency of the driving workflow by 3.65× compared to running alone (orange bars). This degradation occurs because the collocated video workflow is I/O-intensive, grabbing most PCIe bandwidth as its multiple functions load video chunks simultaneously. Therefore, effective bandwidth harvesting requires judicious partitioning of global GPU links (e.g., PCIe links, NICs) to ensure high throughput without contention-induced latency spikes.

3.2.2 Topology-aware path selection. Effective parallel transfer paths require careful planning to align with the underlying GPU topology. Notably, GPUs sharing a PCIe switch (Fig. 5 (a)) connect to host memory via a single PCIe link. Selecting multiple such GPUs for parallel transfers likely induces link contention and should be avoided. In addition, NVLink topologies can be asymmetric. Cost-effective servers like DGX-V100 (Fig. 6(a)) exhibit uneven NVLink bandwidth: 28% of GPU pairs (e.g., GPU1–GPU4) achieve only half the expected bandwidth, while 42% lack direct NVLink (e.g., GPU1–GPU5) and must rely on slower PCIe links. Such configurations are prevalent in production environments [46].

While existing libraries [5, 36, 44] optimize collective communication in asymmetric topologies, no optimization is made for point-to-point transfers. This limitation creates bottlenecks when upstream/downstream functions are placed on weakly connected GPUs (Fig. 6(b))—a common scenario in workflows with fan-in/fan-out patterns. Weak connectivity also undermines PCIe harvesting: if GPU1 borrows PCIe link from GPU5 without a direct NVLink, data must traverse the PCIe bus of GPU1 twice (GPU1 to host and to GPU5), congesting its local PCIe bandwidth and degrading gFn-host transfer performance.

3.3 Challenge #3: Inefficient Memory Management

Serverless functions rely on external storage for indirect data exchange, where intermediate data is temporarily held until consumed by downstream functions. Fig. 7(a) shows the GPU memory usage of the *driving* workflow in our benchmarking suite with simulated requests sampled from the Azure trace [39] on a DGX-V100 server (16 GB per GPU). While GPU memory is often underutilized in serverless inference—due to on-demand function provisioning and small batch size (≤128) [53]—efficient memory management remains critical.

However, existing GPU memory management for serverless inference results in two inefficiencies. (1) *Excessive memory reservation.* To minimize allocation overhead, existing systems pre-reserve GPU memory for storage. However, methods like those in [12, 34] impose no usage constraints but rely on manual reclamation, leading to memory bloat.

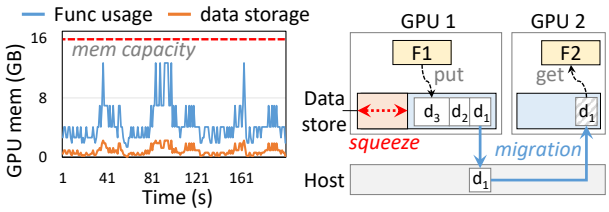

Figure 7. (a) Available idle GPU memory in a serverless inference system under Azure Function trace. (b) Forced data eviction when available GPU memory diminishes.

Our experiments reveal GPU storage consumes 4× more memory than actual demand, a significant waste. (2) *Suboptimal data eviction.* During traffic spikes or data accumulation, GPU memory quickly exhausts, forcing eviction of intermediate data to host memory (Fig. 7(b)). As a result, downstream functions have to retrieve data from host memory, incurring significant gFn-host transfer overhead. Furthermore, traditional eviction polices (e.g., LRU) are designed for intra-program access patterns (e.g., DNN training) without considering function scheduling. Under these policies, data scheduled for imminent use can be mistakenly evicted.

4 GRouter System Design

In this section, we present GRouter, a *GPU-centric* data plane system designed for efficient data exchange in serverless inference workflows. We start with a system overview followed by the detailed descriptions of its key components.

4.1 Design Overview

Fig. 8 illustrates an architecture overview of GRouter, which comprises four key components: (1) *Unified data passing framework.* GRouter provides a unified `put/get` API that abstracts heterogeneous data-passing patterns (e.g., gFn-gFn, gFn-host). Under the hood, it dynamically tracks function placement (physical GPU/CPU locations) and server topology (i.e., the connectivity of NVLinks, PCIe links, and NICs) to orchestrate transfers. (2) *Efficient parallel data transfers.* To fully utilize cluster-wide transfer bandwidth, GRouter enables multi-path data transfers by partitioning and allocating idle GPU links (including NVLinks, PCIe links, and NICs), aggregating available bandwidth while preventing contention among concurrent functions. (3) *Topology-aware transfer scheduling.* For asymmetric GPU topologies, GRouter judiciously selects route GPUs with optimal NVLink connectivity to target GPUs running inference functions. It further exploits idle parallel NVLink paths to accelerate point-to-point data transfers. (4) *Elastic data storage.* GRouter dynamically scales GPU memory allocation in storage by monitoring real-time storage demands and memory pressure. When storage space becomes limited, it evicts low-priority data to host memory or remote idle GPUs while keeping critical data (e.g., for upcoming high-priority functions) on local GPUs to reduce performance penalties from host memory evictions.

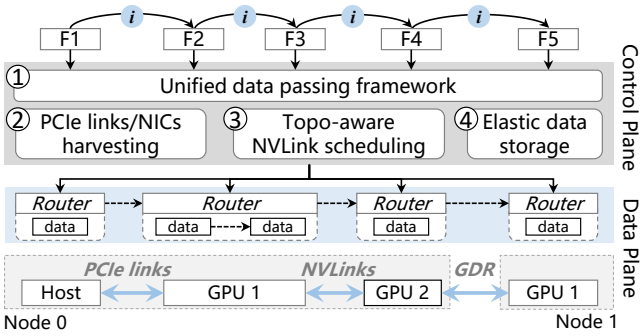

Figure 8. GRouter system overview

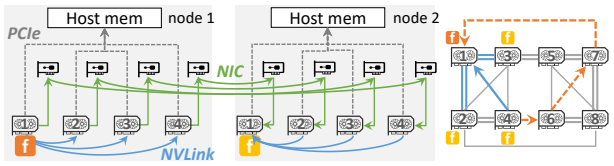

Figure 9. (a) Parallel data transfer for cross-node gFn–gFn transfer. (b) Parallel gFn–gFn data transfer on asymmetric GPU topology.

4.2 Unified Data Passing Framework

4.2.1 Locality-aware transfers and library interface. To avoid unnecessary data transfers across remote GPUs, GRouter detects function placement and caches data locally on the same GPU. It offers two simple data-passing APIs—`Put()` for storing data and `Get()` for retrieving it—similar to cloud storage services like AWS S3 [1]. When `Put()` is called by a function, GRouter identifies its resident GPU, allocates local GPU memory to store the data, and returns a globally unique identifier. This identifier can then be passed to downstream functions. When a function calls `Get()`, GRouter locates the data using the identifier and selects an appropriate GPU transfer method based on the placement of the downstream function. As a result, each piece of data is transferred only once across GPUs during data exchange between GPU functions.

4.2.2 Heterogeneous GPU data-passing patterns. Since the required data of a GPU function may reside in different locations (e.g., host memory, other GPUs, or remote nodes), GRouter supports three data-passing patterns. (1) *Intra-node gFn-gFn transfer.* When the function and data reside on *different* GPUs within a node, GRouter allocates memory on the GPU of functions, maps the address into the address space of function via CUDA IPC [29], and then leverages NVLink to transfer data into the mapped address. In case that the function and data reside on the *same* GPU, GRouter shares the address of data with function directly, enabling zero-copy data access. For asymmetric topologies, it exploits parallel NVLink paths to maximize throughput (§4.3.3). (2) *Cross-node gFn-gFn transfer.* For functions and data on separate nodes, GRouter first allocates memory on the GPU of the function and maps the address into function, then employs *GPUDirect RDMA* (GDR) to directly write data into that address over the network. It further exploits idle NICs (if any) to accelerate transfer (§4.3.3). (3) *gFn-host transfer.* When data resides in host memory, GRouter stages it to the target GPU via parallel PCIe links (§4.3.3), then maps it into the address space of the function.

To transparently manage GPU/host memory and cross-node storage, GRouter uses globally unified data identifiers. It maintains mappings between data identifiers, memory addresses, and data locations (node ID and GPU device ID). For scalability, each node maintains a local mapping table, while a centralized scheduler holds a global table. Lookups and updates are first served by the local table, falling back to the global table only on misses.

4.3 Efficient Parallel Data Transfers

4.3.1 Parallel transfer strategies. GRouter maximizes bandwidth utilization by orchestrating multi-path transfers with strategies tailored to each data-passing pattern, leveraging idle PCIe, NIC, and NVLink resources across the cluster.

- *gFn-host.* For host-bound data, GRouter distributes transfers across idle PCIe links from *route GPUs*. As shown in Fig. 5(a), data from GPU1 is first routed via NVLink to peer GPUs (GPU3, GPU5, and GPU7), which concurrently stage it to host memory through their PCIe links. To avoid contention, GPUs sharing a PCIe switch (e.g., GPU2) are excluded as route GPUs, as they share a single PCIe link to host memory.
- *Cross-node gFn-gFn.* For cross-node transfers, GRouter harnesses idle NICs from multiple GPUs. As illustrated in Fig. 9(a), data from GPU1 (node 1) is split and routed via NVLink to local route GPUs (GPU2–GPU4). These GPUs then transmit chunks in parallel using their dedicated NICs, targeting corresponding GPUs on the remote node (e.g., GPU2→GPU2 on node 2) to minimize NUMA latency. The data is finally aggregated on the destination GPU (GPU1, node 2) via NVLink.
- *Intra-node gFn-gFn.* GRouter exploits indirect NVLink paths for intra-node transfers. In Fig. 9(b), data from GPU4 is split and routed through two parallel paths (GPU4→GPU1 and GPU4→GPU6→GPU7→GPU1), utilizing idle NVLinks to bypass congested direct connections.

To coordinate these strategies, GRouter splits data into small chunks (2 MB by default) and precomputes a parallel transfer plan. Chunks are pipelined across GPU streams, with synchronization primitives ensuring in-order delivery.

To fully utilize cluster bandwidth and accommodate the underlying GPU topology, GRouter incorporates two key mechanisms. First, fine-grained bandwidth harvesting (§4.3.2)

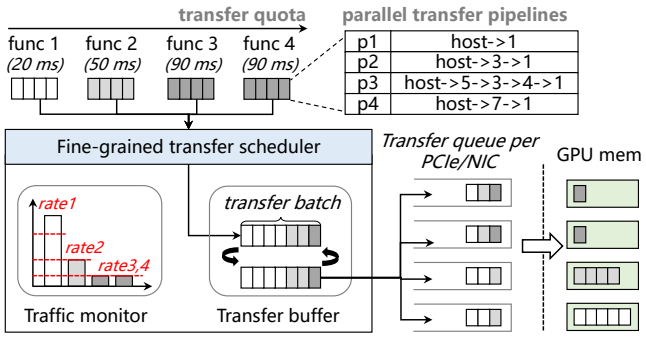

Figure 10. SLO-aware PCIe data transfer scheduling

to avoid contention among concurrent functions sharing the same link—primarily for parallel PCIe and NIC transfers. Second, topology-aware transfer scheduling (§4.3.3) to identify optimal parallel paths based on GPU topology—primarily for parallel NVLink transfers.

4.3.2 Fined-grained bandwidth harvesting. For PCIe and NIC transfers, where bandwidth is the main bottleneck, GRouter aggregates available bandwidth and applies fine-grained partitioning to efficiently allocate it among concurrent functions. Fig. 10 shows the process of transfer scheduling in GRouter. First, data from each function is divided into smaller chunks to enable fine-grained transfer control. GRouter allocates bandwidth to meet the *Service Level Objective* (SLO) of each function and proportionally schedules data chunk transfers. Consistent with prior inference systems [7, 53], the SLO is defined as 1.5–2× the average execution time of each inference service, based on measurements from 10 runs.

Transfer rate control. GRouter first calculates the minimum required transfer rate $Rate_{least}$ for each function based on its SLO and data size, representing the minimum bandwidth necessary to meet the SLO of each function. Let L_{slo} denote the SLO, and L_{infer} denote its inference computation latency. The $Rate_{least}$ is defined as $data\_size/(L_{slo} - L_{infer})$. Given that DNN inference execution exhibits a highly predictable pattern[1] [4, 7, 47, 54], offline profiling can effectively guide transfer control to meet the latency SLOs for each functions.

GRouter monitors the transfer rate of the data block from each function in real time to ensure it remains above $Rate_{least}$. GRouter then calculates the idle transfer rate (i.e., bandwidth) $Rate_{idle}$, which reflects the remaining bandwidth after meeting the minimum bandwidth requirements of all functions. Let BW_{all} denote the total bandwidth in the GPU server, we have $Rate_{idle} = BW_{all} - \sum_{i=0}^{all\_funcs} Rate^{i}_{least}$. GRouter allocates this idle bandwidth to the function with the tightest SLO, enabling latency-sensitive functions to complete their data transfers first without impacting other functions.

[1] In serverless inference, functions running DNN models share GPU devices in a time-multiplexed manner [47, 50], leading to minimum interference with one another.

Algorithm 1: Contention-aware paths selection

Input: Func_id $func$; Source GPU g_s; Destination GPU g_d; The real-time global bandwidth usage matrix BW_{nxn}, The topology matrix $Topo_{nxn}$
Output: The available parallel transfer paths $Paths$

```
while path == null do
    path ← next_shortest_path(BW_nxn, g_s, g_d);
    if all edges in path is idle then
        Paths ← path;
        Update(BW_nxn, path, func);
    if BW_out(g_s) == 0 ∪ BW_in(g_d) == 0 then
        break;
if BW_out(g_s) ≠ 0 ∩ BW_in(g_d) ≠ 0 then
    while path == null do
        path ← next_busy_path(BW_nxn, g_s, g_d);
        bandwidth_balancing(path, func, BW_nxn);
        Paths ← path;
        if BW_out(g_s) == 0 ∪ BW_in(g_d) == 0 then
            break;
return Paths;
```

Batched data transfer. Since initiated data chunk transfers cannot be interrupted, launching all transfers simultaneously would block newly arrived functions from acquiring bandwidth. Conversely, transferring individual chunks incurs excessive connection setup overhead. GRouter balances these tradeoffs with *batched transfers*, grouping chunks into batches (default: 5 chunks per batch). This allows new functions to inject their chunks into subsequent batches, ensuring fair bandwidth preemption while amortizing per-transfer costs. To further optimize PCIe transfers, GRouter maintains a circular pinned memory buffer shared across functions. By reusing this fixed buffer for multiple batches, the system minimizes pinned memory allocation overhead and reduces cache bloat.

4.3.3 Topology-aware transfer scheduling. To optimize parallel NVLink transfers in asymmetric topologies, GRouter employs a *topology-aware* path selection algorithm that maximizes point-to-point bandwidth for weakly connected GPU pairs by exploiting multiple NVLink paths, while avoiding path overlap to prevent contention.

Once the function placement of a workflow is finalized (function scheduler is described in §??), GRouter prioritizes *direct NVLink paths* between GPUs. If these paths are already occupied by other functions (as part of indirect routes), GRouter reassigns those functions to alternative routes (i.e., prioritizing direct path over an indirect route). Then, GRouter searches for available free NVLink paths for each inter-GPU data transfer in the serverless inference workflow, starting with the GPU pair having the least residual

bandwidth. GRouter maintains a bandwidth usage matrix $BW(g, b)$, where g represents GPUs and b is the available bandwidth between them. GRouter continuously monitors and updates global bandwidth usage in real-time on this matrix, which is used to guide path selection.

As shown in Algorithm 1, the selection process involves: GRouter first searches for free paths to avoid contention with other functions (lines 1-7). When a free $path$ is found, the bandwidth usage matrix $BW(g, b)$ is updated. The bandwidth occupied by the $path$ determined by the NVLink with the smallest bandwidth along the path, denoted as $b_{min}(path)$. Thus, the update to $BW(g, b)$ subtracts $b_{min}(path)$ from the free bandwidth of each GPU pair on the path. If all free paths are exhausted and the outgoing bandwidth of g_s and incoming bandwidth of g_d are not saturated, GRouter searches busy paths to see if bandwidth can be balanced between the current function and the one occupying the path (lines 8-14). GRouter compares the total bandwidth used by the running function and the current function, and checks whether the running function can switch to another path. If it is available, the busy path is assigned to the current function. Because a GPU server usually has 4-8 GPUs, after using path pruning and other loop-free path search acceleration, the overhead of path selection is less than 10us in our experiments.

Parallel NVLink transfers use the same pipelined method as in PCIe/NICs transfers. However, to accommodate heterogeneous NVLink bandwidth (24 GB/s or 48 GB/s per link), GRouter dynamically sizes data chunks proportionally to the capacity of each path. For example, a 48 GB/s link receives twice the chunk size of a 24 GB/s link, ensuring balanced utilization and minimizing transfer tail latency.

4.4 Elastic GPU Data Storage

We design elastic GPU data storage to reduce GPU memory usage and adapt to changes of available GPU memory. GRouter dynamically scales storage size based on actual demand and migrates data when memory pressure arises.

4.4.1 GPU storage scaling. Temporary GPU memory allocation incurs significant overhead, as native GPU allocations (e.g., `cudaMalloc()` and `cudaFree()`) incur millisecond-level delays. To address this, existing memory management systems [2, 12, 34] maintain pre-allocated memory blocks as a reusable pool. However, these pooling mechanisms are typically static. For example, in PyTorch [34], users must manually reclaim memory pools, which releases all reserved blocks at once. Therefore, applying static memory pooling to GPU storage results in excessive memory usage from idle reserved memory.

Our key idea is to enable GPU storage to scale the memory pool dynamically based on actual demand. However, estimating the required size is difficult because intermediate data sizes vary with function inputs, batch sizes, and request loads. GRouter adopts a memory pre-warming strategy

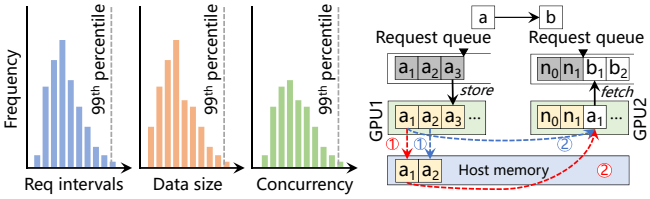

Figure 11. (a) Histogram policy characterizing both request arrivals (blue), intermediate data size (orange), and data accumulation (green) of each function. (b) Illustration of the inefficiency of LRU-based data migration (red line) vs. queue-aware data migration (blue line).

inspired by function pre-warming [39, 48] in serverless systems, which tracks request intervals ($R_{window} = Interval^{99th}$) to estimate how long functions stay active in memory. Beyond this, GRouter also monitors intermediate data sizes ($R_{size} = Data_size^{99th}$) and the degree of data accumulation ($R_{con} = Concurrency^{99th}$) in GPU storage, as shown in Fig. 11(a). After each function execution, memory reservation is calculated as $Data\_size = R_{size} \cdot R_{con}$. If no new requests arrive within the reservation window, the reserved memory is reclaimed. The total memory pool size is given by $MemPool\_size = \sum_{func} Data\_size \cdot 1_{\{R_{window} \cap t \neq \emptyset\}}$, where 1_A is an indicator function of events that returns 1 if event A is true and 0 otherwise. To handle bursty requests, GRouter maintains a minimum memory pool (e.g., 300 MB) in idle periods, when GPU memory is sufficient.

4.4.2 Proactive data migration. When GPU memory pressure increases, available memory for storage becomes limited, requiring intermediate data to be evicted to reduce GPU storage usage. However, migrating data to host memory forces downstream functions to fetch it with additional latency. An effective migration strategy is thus critical. Existing approaches [6, 17, 33] typically adopt an LRU strategy, which evicts the least recently accessed data. However, LRU ignores function scheduling and often migrates data that will soon be accessed. For instance, as shown by the red line in Fig. 11 (b), the LRU strategy tends to evict the output data of function a_1 first, ignoring that b_1 (the downstream function of a_1) is enqueued earlier, forcing b_1 to reload data from host memory and introducing additional delays. To address this, GRouter uses a request queue-aware migration strategy that prioritizes evicting data needed by functions at the tail of the queue, ensuring that data required by imminent function invocations remains in GPU storage. As shown by the blue line in Fig. 11(b), the output data of function a_2 is migrated before the output of a_1.

Furthermore, GRouter promptly removes intermediate data that is no longer needed and proactively restores previously migrated data when sufficient GPU memory becomes available. For instance, after the output of a_1 is processed,

the output of a_2 is reloaded into GPU memory. This proactive migration approach ensures upcoming functions can access data locally, minimizing performance degradation under fluctuating available GPU memory. GRouter triggers data migration and restoration automatically based on available GPU memory, maintaining storage usage within a fixed threshold (50% of free memory in our experiments) to avoid contention with function execution while maximizing GPU memory utilization.

5 Implementation

GRouter is built on INFless [48], a state-of-the-art serverless inference system. It comprises 5K lines of C++ code. Each function runs in a container with on-demand CPU and GPU allocation [28].

Data storage. GRouter mounts a shared memory region to each function for efficient data and message exchange. On the host side, it attaches a host volume to each function. On the GPU side, it maintains an elastic memory pool on each GPU for data storage. When a function stores or retrieves data, GRouter allocates memory from the local pool and maps it into the address space of functions using CUDA IPC [29]. Each GPU runs an I/O thread to reclaim unused memory and migrate data between GPU and host memory based on available storage space.

Data transfer management. GRouter launches a daemon thread on each GPU to manage data transfers from functions. Each thread uses multiple GPU streams to enable parallel transfers in different directions and coordinates with other threads based on pre-planned pipeline paths. Most parallel transfer paths, such as PCIe links and NIC routes, are fixed and can be pre-generated during GRouter initialization, allowing real-time requests to use them directly.

Function scheduling. For function scheduling in GPU clusters, GRouter adopts a hierarchical control plane. Most data transfers and scheduling decisions are handled by local control plane within a node, while the global plane is invoked only for infrequent cross-node coordination, thereby minimizing inter-node transfers and scheduling overhead. Within a GPU node, GRouter employs the MAPA strategy [36] maximize the utilization of GPU interconnects across functions. To further mitigate the performance impact of cold starts, GRouter pre-warms necessary functions and models, similar to the approach used in SHEPHERD [53].

6 Evaluation

Setup. We evaluate GRouter using two AWS GPU testbeds. *Testbed 1* (DGX-V100) uses `p3.16xlarge` instances, each containing 8 NVIDIA V100 GPUs connected via NVLinks, a Xeon E5-2686 v4 CPU (32 vCPUs), 244 GB of memory, and 4×100 Gbps NICs. *Testbed 2* (DGX-A100) uses `p4d.24xlarge` instances, each having 8 NVIDIA A100 GPUs connected via NVSwitch, a Xeon Plati. 8275CL CPU, 1152 GB of memory, and 8×200 Gbps NICs.

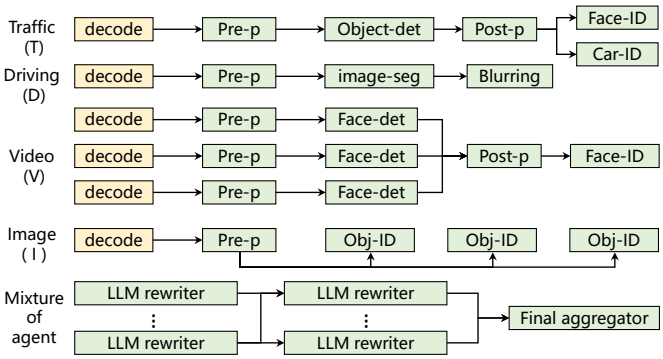

Figure 12. Real-world inference workflows composed of GPU functions (green) and CPU functions (yellow). They are organized into four typical patterns: condition, sequence, fan-in, and fan-out.

Real-world inference workflows. We conduct experiments using six inference workflows collected from the latest studies, as detailed below and in Fig. 12. All pre-processing and post-processing are performed on the GPU using NVIDIA CV-CUDA [30]. The input datasets are from Adainf [40].

- *Traffic* (T). Following Boggard [3], we implement a traffic monitoring workflow which first detects objects using the Yolo-det model, and then performs feature recognition on pedestrian and vehicle sub-images using ResNet models.
- *Driving* (D). Following Adainf [40], we implement a road segmentation workflow for auto-driving. The process involves denoising the image, applying a semantic segmentation model, and outputting a colored image.
- *Video* (V). Following Aquatope [55], we implement a video processing workflow that runs a face detection model on video chunks in parallel, followed by a recognition model to identify a specified actor.
- *Image* (I). Following Cocktail [11], we implement an image classification workflow that first denoises the image, then applies multiple classification models simultaneously, and aggregates the results to improve accuracy.
- *Mixture of Agent* (MoA). Following MoA [45], we implement a layered agent workflow wherein each layer comprises multiple LLM agents. Each agent takes all the outputs from agents in the previous layer as auxiliary information in generating its response.

Baselines. We compare GRouter to the following baselines:

- *INFless+*. This baseline represents a *host-centric* design that extends INFless [48]—a state-of-the-art serverless inference system—by incorporating a host-side shared-memory storage layer for efficient inter-function communication. We denote this approach as INFless+.
- *NVSHMEM+*. This baseline adopts NVSHMEM [32] to enable GPU-side storage layer (randomly assigned to one

GPU per data object). With NVSHMEM, GPU functions can directly store and retrieve intermediate data through a shared GPU memory space, bypassing host memory. We refer to this approach as NVSHMEM+.

- *DeepPlan+*. This baseline further enhances NVSHMEM+ by integrating PCIe optimizations from DeepPlan, which enables parallel data transfers across all available PCIe links in a GPU node. We refer to this approach as DeepPlan+. Note that parallel PCIe transfers are handled by the storage service, as other GPUs' PCIe are invisible to functions.

Workloads. We simulate the invocation of inference workflows using production traces from Azure Function [39], following the methodology of prior serverless inference systems [23, 48, 53]. The traces exhibit three characteristic request arrival patterns: sporadic, periodic, and bursty.

6.1 Data Passing Performance

We first evaluate the data passing latency between two functions under various scenarios. Fig. 13 illustrates the data passing latency between functions under varying data volumes. The latency measures the time elapsed between the upstream function sending the data and the downstream function receiving it.

Intra-node gFn-gFn. When GPU functions are colocated within the same node (Fig. 13(a)), GRouter achieves the lowest data passing overhead, reducing latency by 95%, 75%, and 75% compared to INFless+, NVSHMEM+, and DeepPlan+, respectively. INFless+ uses host memory for data exchange, leading to large overhead. NVSHMEM+ lacks awareness of function locations, leading to extra data copies with a remote GPU. DeepPlan+ optimizes gFn-host transfers but neglects gFn-gFn transfers. GRouter detects function placement and stores data on the local GPU to eliminate redundant data copies. It further accelerates data transfer on DGX-V100 server by leveraging parallel NVLinks.

Host-gFn. For data passing between GPU functions and host memory (Fig. 13(b)), GRouter uses the global PCIe links, reducing latency by 63%, 63%, and 75% compared to NFless+, NVSHMEM+, and DeepPlan+, respectively. INFless+ and NVSHMEM+ only use the PCIe link of the local GPU, leading to long delays. DeepPlan+ also uses parallel PCIe links, but it lacks topology awareness, leading to worse performance than NVSHMEM+ on asymmetric topologies (DGX-V100), as it selects route GPUs with limited NVLink connectivity to the current GPU, causing PCIe bandwidth congestion. Moreover, since functions have limited access to GPU resources, only the external storage can see the all PCIe links and underlying topology. The storage service of DeepPlan+, however, cannot detect function placement, resulting in redundant data copies—for instance, data is first pulled to a remote GPU, then copied to the GPU device of the target function.

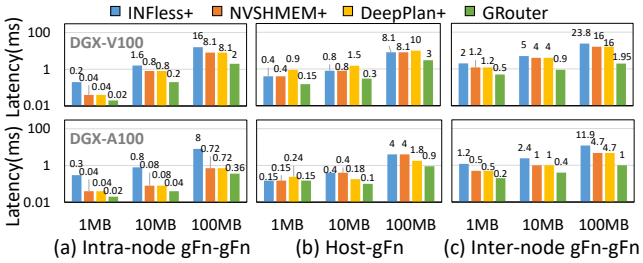

Figure 13. Comparison of the data passing latency

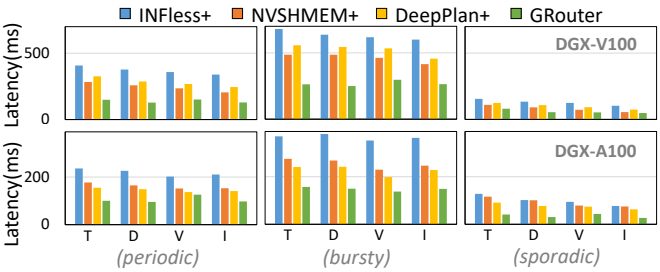

Figure 14. Comparison of the end-to-end latency

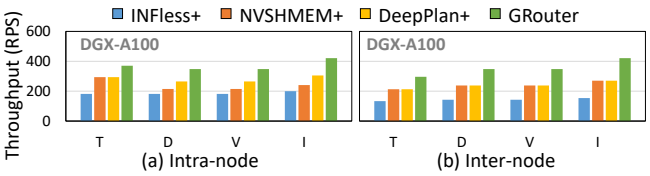

Figure 15. Comparison of the maximum throughput

Inter-node gFn-gFn. For GPU functions distributed across different nodes (Fig. 13(c)), GRouter reduces data passing latency by 91%, 87%, and 87% compared to INFless+, NVSHMEM+, and DeepPlan+, respectively. INFless+ incurs high overhead by routing data through host memory. Both NVSHMEM+ and DeepPlan+ use only a single NIC for cross-node data transfers. In contrast, GRouter enables locality-aware data transfer between GPUs across nodes without redundant data copies and leverages multiple NICs for parallel transfers.

6.2 Performance under Real-world Workloads

We next evaluate GRouter using real-world inference workflows and production traces from Azure Function [39]. We scale the traces to ensure effective resource utilization, aligning with prior studies [55].

End-to-end latency. Fig. 14 shows the P99 latency across various applications under different production workloads. On DGX-V100 servers, GRouter reduces latency by 61%, 48%, and 54% compared to INFless+, NVSHMEM+, and DeepPlan+, respectively. DeepPlan+ performs worse than NVSHMEM+ due to its lack of NVLink connectivity awareness. On DGX-A100 servers, GRouter reduces latency by 53%, 36%, and 30% compared to INFless+, NVSHMEM+, and DeepPlan+.

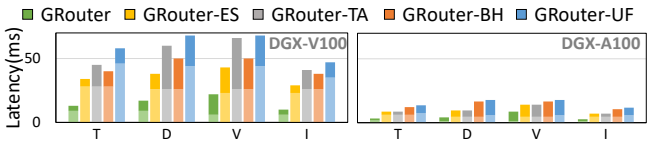

Figure 16. The average data passing latency when disenabling each optimization in GRouter one by one

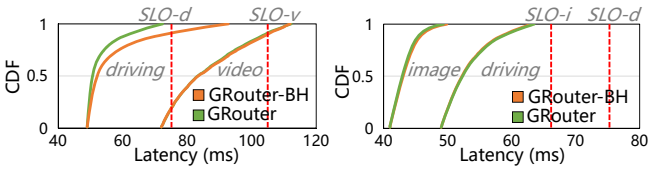

Figure 17. The effectiveness of fine-grained bandwidth harvesting in GRouter

Compared to NVSHMEM+, GRouter aggregates available bandwidth and eliminates redundant data transfers. It also optimizes GPU storage by keeping high-priority data (for upcoming functions) in GPU memory, avoiding costly host-memory fetches.

Throughput. Fig. 15 shows the maximum throughput of these inference workflows within the same node and across different nodes. When functions are colocated within the same node, GRouter surpasses INFless+, NVSHMEM+, and DeepPlan+ by 2.1×, 1.74×, and 1.37×, respectively, by locality-aware GPU data transfer and efficiently leveraging parallel NVLink and PCIe links. For functions distributed across nodes, GRouter outperforms INFless+, NVSHMEM+, and DeepPlan+ by 2.73×, 1.55×, and 1.39×, respectively, through direct inter-node GPU data transfers and utilization of multiple NICs.

6.3 Performance of Components in GRouter

We next evaluate the effectiveness of each design in GRouter.

Ablation study. We incrementally disable optimizations in GRouter to assess their impact on data passing latency, including *elastic storage* (ES), *topology-aware scheduling* (TA), *GPU bandwidth harvesting* (BH), and *the unified data passing framework* (UF). Fig. 16 presents the average data passing latency under a bursty workload. On DGX-V100 servers, disabling all optimizations (rightmost bar) increases latency by 1.57×–1.82× compared to GRouter, with ES, TA, and UF having the greatest effects. On DGX-A100 servers, latency increases 1.30×–1.61× when all optimizations are removed, with ES and BH having the greatest impact.

Bandwidth partitioning. To demonstrate the effectiveness of the fine-grained *bandwidth harvesting* (BH) in achieving performance isolation between concurrent functions, we conduct mixed workload experiments using two workflow pairs on DGX-V100 servers. Following GPUlet [7], the

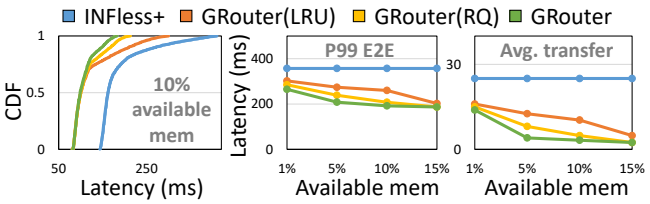

Figure 18. (a) Latency under 10% available GPU memory. (b) End-to-end latency under different available memory ratios. (c) Average gFn-gFn data passing latency.

SLO for each workflow is set to 1.5× its independent execution time. We compare GRouter with GRouter-BH, which employs PCIe bandwidth sharing as in DeepPlan+. Both workflows run under bursty workload, consistent with previous experiments. Fig. 17(a) presents the results for a high-contention case where the latency-critical driving workflow is paired with a transfer-intensive video workflow, which involves multiple functions loading video chunks simultaneously. Without bandwidth partitioning, the latency of driving workflow is increased due to interference from the video workflow. In contrast, GRouter controls PCIe bandwidth usage by the video workflow, allowing more bandwidth for the driving workflow. This reduces driving workflow latency by 32% and improves *Service Level Objective* (SLO) compliance. Fig. 17(b) shows results for a low-contention scenario, where driving workflow is paired with image workflow. In this case, GRouter and GRouter-BH performes identically, indicating that GRouter introduces minimal overhead in transfer scheduling.

Elasticity of GPU storage. To evaluate the efficiency of the GPU storage of GRouter, we measure latency under limited available memory and compare it with INFless+, LRU (used by NVSHMEM+), and a *request queue-aware approach* (RQ) without proactive data migration. Fig. 18(a) shows the end-to-end latency distribution under a bursty workload with GPU storage limited to 10% of the GPU memory. Compared to INFless+, LRU, and RQ, GRouter reduces tail latency by 46%, 27%, and 7%, respectively. RQ prioritizes keeping data accessed earlier by downstream functions in GPU memory, while GRouter further reduces latency through proactive data migration compared to RQ. As shown in Fig. 18(b), further tests under different memory availability ratios show that even with only 1% available memory, GRouter reduces end-to-end latency by 24%, 14%, and 9%, respectively. Fig. 18(c) shows the average data passing latency. Compared to INFless, LRU, and RQ, GRouter reduces delays by 83%, 72%, and 49%, respectively. These results demonstrate that GPU storage management in GRouter and proactive data migration efficiently utilize available GPU memory and maintain performance under memory constraints. Despite severe memory constraints (1%), parallel PCIe transfers in GRouter mitigate the overhead of fetching data from host memory.

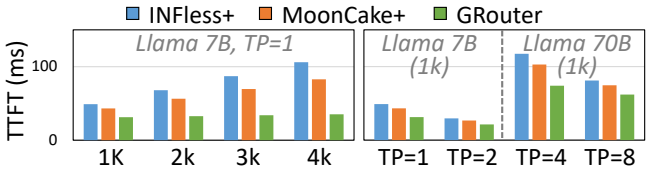

Figure 19. (a) TTFT under different input lengths. (b) TTFT under different models and *tensor parallelism* (TP).

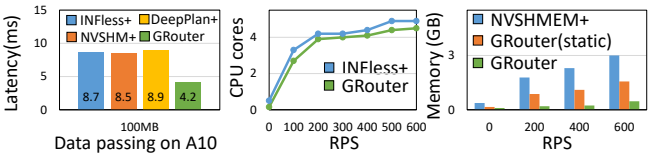

Figure 20. (a) Data passing latency in 4xA10 GPU server. (b) Comparison of CPU overhead. (c) Comparison of GPU memory overhead.

6.4 Performance under Emerging LLM Applications

We evaluate the performance of GRouter in *Large Language Model* (LLM) workflows, using the *Mixture-of-Agent* [45] (MoA) as an example. In this multi-stage workflow, multiple LLMs optimize answers from the previous stage to improve quality, passing the Key-Value Cache (KV cache) of the prompt and response among stages to avoid recomputation. Different stages are deployed on separate 8×H800 GPU nodes, with GPUs connected via 200 GB/s NVLink and nodes connected by 200 Gbps networks. Due to the specialized management of the KV cache, we select Mooncake [35]—a state-of-the-art KV cache system—as the baseline and implement it on the serverless system, referred to as Mooncake+. Following DroidSpeak [24], we report *The First Token Time* (TFTT) of the receiver LLM.

Fig. 19(a) shows the TTFT for different input lengths. For a 4K input length, GRouter reduces TFTT by 66% and 57% compared to INFless+ and Mooncake+, respectively. Fig. 19(b) further shows that GRouter reduces TFTT by 36% and 28% under various models and *Tensor Parallelism* (TP) settings, respectively. INFless+ transfers the KV cache to host memory, incurring high overhead. Mooncake incurs extra copies due to lack of function placement awareness and utilization of single NIC. In contrast, GRouter avoids redundant copies and uses multiple NICs. As TP increases, Mooncake begins using multiple NICs, narrowing the advantage of GRouter. At TP=8, the advantage of GRouter mainly comes from locality-aware data transfers without extra copies.

6.5 Applicability and System Overhead

Testbed without NVLink. Fig. 20(a) shows the data passing latency between GPU functions on 4×10 GPU servers (without NVLink). GRouter reduces latency by 51% compared to INFless+, NVSHMEM+, and DeepPlan+. NVSHMEM+ performs similarly to INFless+ due to lack of function placement awareness, leading to two peer-to-peer GPU data copies via PCIe. In contrast, GRouter only requires one copy as it can detect the location of functions. Therefore, GRouter proves to be highly effective in testbeds even without NVLink.

CPU overhead. We evaluate the system overhead in GRouter. Fig. 20(b) shows that the CPU resources used by GRouter are similar to those of the state-of-the-art serverless inference system, INFless+. While the control plane of GRouter introduces additional tasks, such as monitoring GPU link usage and memory pressure, these operations are performed periodically or triggered only by new requests or data, resulting in negligible CPU overhead.

GPU memory overhead. Fig. 20(c) shows that GRouter uses the least GPU memory. In NVSHMEM, symmetric memory allocation [32] leads to significant waste, as all processes allocate and release GPU memory simultaneously. The static memory pooling method also lacks awareness of storage needs, causing over-pooling. In contrast, GRouter dynamically scales storage space based on actual requirements.

7 Discussion and Related Work

Threat Model of GRouter. GRouter provides a unified data storage service for functions while placing a strong emphasis on data security, even in the presence of shared resources such as transfer buffers and data storage. To achieve this, GRouter enforces two key forms of isolation: (1) *Address isolation.* In GRouter, both data storage and transmission buffers are allocated in containers that are isolated from the function itself, each with its own separate address space (e.g., a dedicated CUDA context). Functions can only access GPU storage through pre-mapped addresses (e.g., via CUDA IPC with enforced alignment). Moreover, transmission buffers are never mapped into a function's address space, preventing any direct access by the function. These isolation mechanisms ensure that functions cannot reach data outside their designated boundaries, thereby mitigating the risk of leakage through out-of-bounds accesses. (2) *Access control.* Data items are exchanged across functions using data IDs, which introduces the potential risk of ID leakage or attacks. To address this, GRouter authenticates the requesting function using both *function_ID* and *workflow_ID* on every access, ensuring that only authorized functions can read or manipulate specific data items. To minimize overhead, GRouter employs a hierarchical control plane: IDs and metadata are synchronized to the local node at invocation time, avoiding frequent cross-node lookups during execution.

In addition to these mechanisms, GRouter provides a security level comparable to the latest serverless platforms [19, 20, 48] across functions. Each function operates within its own independent container, with isolated host memory, NIC buffers, and GPU runtime contexts (separate CUDA contexts

with private GPU address spaces). For workloads requiring even stronger guarantees, GRouter can also support microVMs [49].

GPU sharing supports in GRouter. Existing GPU-enabled serverless systems typically employ GPU sharing to maximize resource utilization, including temporal sharing (e.g., DGSF [9] and FaaSwap [50]) and spatial-sharing (e.g., StreamBox [47] and Llama [37]). While GRouter adopts a temporal-sharing model, its optimizations are orthogonal to GPU sharing strategies. In fact, spatial GPU sharing inevitably incurs more serious bandwidth and memory contention, which makes optimizations in GRouter—transfer bandwidth partitioning and GPU storage management—even more critical.

Multi-GPU communication. Existing GPU communication libraries [5, 18, 27, 36, 38] leverage high-speed GPU interconnects for collective communication such as allReduces. Some multi-GPU inference systems also utilize these interconnects to transfer embeddings (e.g., UGache [42]) or KV caches (e.g., MoonCake [35]) for recommendation and LLM workloads. However, these systems are not designed for serverless environments, resulting in redundant data copies and limiting each GPU to utilize only its own bandwidth resources (e.g., a single PCIe, NVLink, or NIC). In contrast, GRouter aggregates available bandwidth across GPUs via multi-path transfers. Unlike collective communication methods that coordinate bandwidth globally, GRouter can dynamically aggregate global bandwidth resources for GPU functions running on a single GPU.

GPU memory management. Existing methods focus on pooling memory and unifying multi-level memory. GMlake [12] uses CUDA virtual memory to reduce fragmentation in memory pooling, while CUDA UVM [33], HUVM [6], and DeepUM [17] address GPU memory limits by swapping data between GPU and host memory. However, these methods lack elastic memory management and awareness of request scheduling, which can lead to large memory occupation and suboptimal data eviction. In contrast, GRouter dynamically scales GPU storage on demand and migrates data when memory pressure arises.

Serverless workflow optimizations. Current research primarily focuses on traditional CPU-based workflows. Systems such as Pheromone [49] and Unum [22] optimize function composition, while Dataflower [21] and Fuyao [23] improve data transfer in host memory, Nightcore [16] minimizes runtime redundancy, and FaasFlow [20] enhances function scheduling. Although these methods are orthogonal to GRouter, none addresses the need for efficient GPU data transfer in serverless inference workflows. In contrast, GRouter fully utilizes available GPU transfer links and memory across GPU cluster.

8 Conclusions

In this paper, we present GRouter, a *GPU-centric* serverless data plane that efficiently transfers data between heterogeneous CPU and GPU functions for ML inference through three key innovations. First, a unified GPU memory storage enabling direct GPU-to-GPU data exchange via topology-aware transfers. Second, multi-link bandwidth harvesting that aggregates PCIe and NVLink interconnects for parallel data movement. Third, elastic memory management adapting to dynamic workload demands. Evaluations show that GRouter reduces data passing latency by up to 87% and improves throughput by up to 1.74× compared to state-of-the-art GPU communication libraries.

9 Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under grant 2022YFB4500704, the National Science Foundation of China under grants 62032008 and 62572204, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grants (Ref. #16217124 and #16210822), NSFC/RGC CRS Grants (#CRS_HKUST601/24), HUST Kunpeng&Ascend Center of Cultivation, CCF-Huawei Populus Grove Fund. Hao Fan, Minchen Yu, Song Wu, and Wei Wang are the corresponding authors.

References

[1] 2007. ZeroMQ. https://aws.amazon.com/s3/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, 2016.

[3] Neil Agarwal and Ravi Netravali. Boggart: towards general-purpose acceleration of retrospective video analytics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 933-951, 2023.

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing*, pages 123–135, 2020.

[5] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 62–75, 2021.

[6] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. Memory harvesting in multi-gpu systems with hierarchical unified virtual memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 625-638, 2022.

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing. In *Proceedings of the USENIX Annual Technical Conference*, pages 199-216, 2022.

[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[9] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. Dgsf: disaggregated gpus for serverless functions. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 739-750, 2022.

[10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: low-latency serverless inference for large language models. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 135-153, 2024.

[11] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: a multidimensional optimization for model serving in cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 1041-1057, 2022.

[12] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. Gmlake: efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 450–466, 2024.

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 539-558, 2022.

[14] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: a cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 624–638, 2021.

[15] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the ACM European Conference on Computer Systems*, pages 249–265, 2023.

[16] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[17] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. Deepum: tensor migration and prefetching in unified memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 207-221, 2023.

[18] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. Tccl: discovering better communication paths for pcie gpu clusters. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 999–1015, 2024.

[19] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: memory-efficient serverless inference through tensor sharing. In *Proceedings of the USENIX Annual Technical Conference*, pages 125-142, 2022.

[20] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 782-796, 2022.

[21] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. Dataflower: exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 57–72, 2024.

[22] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing more with less: orchestrating serverless applications without an orchestrator. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 1505-1519, 2023.

[23] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. Fuyao: dpu-enabled direct data transfer for serverless computing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 431–447, 2024.

[24] Yuhan Liu, Yuyang Huang, Jiayi Yao, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. Droidspeak: kv cache sharing for cross-llm communication and multi-llm serving. *arXiv preprint arXiv:2411.02820* , 2024.

[25] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/Deserialization-free state transfer in serverless workflows. In *Proceedings of the European Conference on Computer Systems*, pages 132–147, 2024.

[26] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. Sonic: application-aware data passing for chained serverless applications. In *Proceedings of the USENIX Annual Technical Conference*, pages 285-301, 2021.

[27] Nvidia Collective Communications Library NCCL. https://developer.nvidia.com/nccl.

[28] Nvidia Container Toolkit. https://github.com/NVIDIA/nvidia-container-toolkit?tab=readme-ov-file.

[29] Nvidia CUDA IPC. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] Nvidia CV-CUDA. https://github.com/CVCUDA/CV-CUDA.

[31] Nvidia GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[32] Nvidia NVSHMEM. https://docs.nvidia.com/nvshmem/api/index.html.

[33] Nvidia Unified Memory. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In *Proceedings of the Advances in Neural Information Processing Systems*, 8024-8035, 2019.

[35] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: trading more storage for less computation — a kvcache-centric architecture for serving llm chatbot. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 155-170, 2025.

[36] Kiran Ranganath, Joshua D. Suetterlein, Joseph B. Manzano, Shuaiwen Leon Song, and Daniel Wong. Mapa: multi-accelerator pattern allocation policy for multi-tenant gpu servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1-14, 2021.

[37] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: a heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–17, 2021.

[38] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: guiding collective algorithm synthesis using communication sketches. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 593-612, 2023.

[39] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark

Russinovich, and Ricardo Bianchini. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the USENIX Annual Technical Conference*, pages 205-218, 2020.

[40] Sudipta Saha Shubha and Haiying Shen. Adainf: data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proceedings of the ACM SIGCOMM Conference*, pages 473–485, 2023.

[41] Amirhossein Sojoodi, Yiltan Hassan Temucin, and Ahmad Afsahi. Enhancing intra-node gpu-to-gpu performance in MPI+UCX through multi-path communication. In *Proceedings of the International Workshop on Extreme Heterogeneity Solutions*, pages 9-14, 2024.

[42] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. Ugache: a unified gpu cache for embedding-based deep learning. In *Proceedings of the Symposium on Operating Systems Principles*, pages 627–641, 2023.

[43] Unified Communication X. https://openucx.readthedocs.io/en/master/.

[44] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: fast and generic collectives for distributed ML. In *Proceedings of Machine Learning and Systems*, pages 172-186, 2020.

[45] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. *arXiv preprint arXiv:2406.04692* , 2024.

[46] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 945-960, 2022.

[47] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. Streambox: a lightweight gpu sandbox for serverless inference workflow. In *Proceedings of the USENIX Annual Technical Conference*, pages 59-73, 2024.

[48] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[49] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: rethinking function orchestration in serverless computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 1489-1504, 2023.

[50] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Torpor: GPU-enabled serverless computing for low-latency, resource-efficient inference. In *Proceedings of the USENIX Annual Technical Conference*, pages 125-142, 2025.

[51] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. λScale: enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922* , 2025.

[52] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: exploiting cloud services for cost-effective, slo-sware machine learning inference serving. In *Proceedings of the USENIX Annual Technical Conference*, pages 1049-1062, 2019.

[53] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: serving dnns in the wild. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 787-808, 2023.

[54] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. Astraea: towards qos-aware and resource-efficient multi-stage gpu services. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 570-582, 2022.

[55] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1-14, 2022.