FaaSTube: Optimizing GPU-oriented Data Transfer for Serverless Computing

Hao Wu[†], Junxiao Deng[†], Minchen Yu[§], Yue Yu[†], Yaochen Liu[†], Hao Fan[†], Song Wu[†], Wei Wang[‡]

[†]*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
[‡]*Hong Kong University of Science and Technology*
[§]*The Chinese University of Hong Kong, Shenzhen*

Abstract

Serverless computing has gained significant traction for machine learning inference applications, which are often deployed as serverless workflows consisting of multiple CPU and GPU functions with data dependency. However, existing data-passing solutions for serverless computing primarily reply on host memory for fast data transfer, mandating substantial data movement and resulting in salient I/O overhead. In this paper, we present FaaSTube, a GPU-efficient data passing system for serverless inference. FaaSTube manages intermediate data within a GPU memory pool to facilitate direct data exchange between GPU functions. It enables fine-grained bandwidth sharing over PCIe and NVLink, minimizing data-passing latency for both host-to-GPU and GPU-to-GPU while providing performance isolation between functions. Additionally, FaaSTube implements an elastic GPU memory pool that dynamically scales to accommodate varying data-passing demands. Evaluations on real-world applications show that FaaSTube reduces end-to-end latency by up to 90% and achieves up to 12x higher throughput compared to the state-of-the-art.

1 Introduction

The widespread adoption of Machine Learning (ML) inference applications has heightened the demand for efficient inference service systems [10, 17, 19, 52]. Recent research suggests deploying inference applications on GPU-enabled serverless platforms [5, 23, 38, 48, 49, 51]. With serverless computing, users package ML models as functions and leave resource provisioning and scaling to the serverless platform. This approach enables users to concentrate on the development of application logic while maintaining resource efficiency, eliminating the need for overprovisioning.

In real-world scenarios, inference applications often stitch together multiple models and operations into a workflow. Table 1 presents several representative inference applications from recent research [4, 11, 15, 18, 42]. As a concrete example, in a traffic monitoring application [42] that analyzes pedestrian and vehicle traffic (Fig. 1), video frames are first decoded and preprocessed. A detection model then extracts
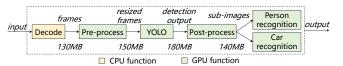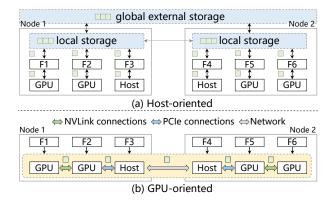

Figure 1. A typical traffic analysis application [4, 42].

Figure 2. Comparison of host-oriented inter-function data passing and our GPU-oriented inter-function data passing.

objects from these frames. The sub-images of detected pedestrians and vehicles are subsequently passed to two separate recognition models for further analysis of behavior and type. Inference applications in serverless computing are structured as *serverless inference workflows*. These workflows are hybrid, consisting of GPU functions (gFunc), CPU functions (cFunc), and their data dependencies.

Serverless inference workflows involve various types of data passing. In addition to typical cFunc-to-cFunc data passing, there are host-to-gFunc (where the host represents either a cFunc or I/O data in host memory) and gFunc-to-gFunc data passing. Unfortunately, current serverless systems rely on a **Host-oriented** data passing approach (Fig. 2(a)), where intermediate data is stored and exchanged via host memory (i.e., host-side external storage [20, 25, 27, 29, 50]). This host-oriented method overlooks the decoupling of GPU and host memory and the potential of various connections in GPU servers, resulting in substantial unnecessary overhead.

For instance, in gFunc-to-gFunc data passing, this method ignores direct NVLinks between GPUs, requiring multiple sequential copies: data is first copied to host memory and then back to the target GPU, leading to large overhead (92% of the total latency in our experiments). Similarly, for host-to-gFunc data passing, this method only uses a single PCIe link, neglecting the parallel PCIe links available within GPU servers. For instance, parts of the data can be routed to neighboring GPUs via NVLink and then transferred in parallel to host memory through the neighboring GPUs' PCIe links.

Our key idea is to introduce a **GPU-oriented** data-passing approach for serverless inference workflows. This approach allows data to be stored and exchanged directly in GPU memory, avoiding redundant transfers between GPU and host memory. It also leverages various connections within GPU servers (e.g., parallel PCIe links and NVLinks) to accelerate inter-function data passing, specifically focusing on host-to-gFunc and gFunc-to-gFunc data passing in this paper. To design an efficient GPU-oriented data-passing framework for serverless functions, the following requirements must be met: (1) *Bandwidth efficiency*: The available bandwidth of the various connections must be fully utilized while avoiding contention among concurrent functions. (2) *Topology awareness*: There are different connection topologies in modern GPU servers, requiring careful management of transfer scheduling based on the specific GPU topology during inter-function data passing. (3) *GPU memory efficiency*: As highlighted by existing research [9, 21], GPU memory is a limited resource; therefore, its usage should be minimized when providing GPU data storage for functions. (4) *Transparent deployment*: All of the above should be achieved transparently to users, simplifying the development of serverless inference workflow and hiding the complexities of various connections and GPU topologies involved.

Based on these requirements, we propose FaaSTube, a GPU-efficient data transfer framework designed for serverless inference workflows. FaaSTube acts as a transparent "tube" across GPUs, facilitating efficient data storage and transfer for functions. It comprises three key components:

First, to simplify the management of various data passing in serverless inference workflows, we provide a unified data-passing interface and data index. This allows users to focus on application logic while offloading the complexities of data transfer implementation to FaaSTube. It automatically locates intermediate data within GPU servers and selects the appropriate connections (PCIe, NVLink, or network) and transfer methods (e.g., parallel or pipelined) to deliver data to the requesting function.

Second, in order to fully utilize connections such as PCIe and NVLink for serverless functions, we design effective transfer scheduling mechanisms. (1) To achieve interference-free sharing of PCIe connections among concurrent functions on a shared GPU server, FaaSTube proposes fine-grained PCIe bandwidth isolation. Native GPU PCIe scheduling is not optimized for parallel transfers, and related works [13, 19] focus on exclusive inference systems while overlooking scenarios where multiple functions share a GPU server. FaaSTube manages data transfers across all PCIe connections, flexibly partitioning bandwidth based on each function's SLO and controlling bandwidth usage during transfers. In addition, FaaSTube employs circular pinned memory buffers to enhance PCIe transfer efficiency while minimizing allocation overhead. (2) To ensure robust inter-function data transfer performance across different GPU topologies, FaaSTube implements topology-aware parallel NVLink transfer scheduling. In non-uniform topologies, many GPUs with limited direct NVLink bandwidth can severely restrict the performance of point-to-point (i.e., gFunc-to-gFunc) transfers in serverless inference workflows. Existing works [7, 37, 45] typically optimize task placement but still rely on a single direct NVLink path, failing to address this issue comprehensively. Other multi-GPU communication methods [22, 30, 39] focus on collective communication and often ignore point-to-point transfers, also utilizing only one NVLink path. FaaSTube identifies multiple parallel NVLink paths between bandwidth-constrained GPUs, employing contention-aware path selection to accelerate inter-function data passing.

Third, to efficiently manage GPU memory and intermediate data in GPU data store for serverless functions, FaaSTube proposes an auto-scaling GPU memory pool, as function workloads and the size of intermediate data (e.g., the number of objects in a video frame) can vary dynamically. Existing GPU memory management systems [16, 21, 36] are designed for long-running, exclusive GPU tasks like ML training. Therefore, they often retain a lot of idle memory blocks and lack flexible reclamation mechanisms, leading to excessive memory usage in serverless environments. FaaSTube tracks data storage requirements in real-time and caches only the necessary memory blocks in GPU data store, dynamically resizing the memory pool. Furthermore, when intermediate data accumulates in GPU memory, FaaSTube implements a smart data migration based on function request queue, migrating data to host memory and adaptively prefetching data back to the GPU, thereby effectively alleviating memory pressure without compromising performance.

We implement FaaSTube on top of INFless [49], a recent serverless inference platform built on OpenFaaS. We evaluate FaaSTube on various real-world inference workflows using Azure cloud traces [40]. The results show that FaaSTube reduces end-to-end latency by up to 90% and achieves up to 12X higher throughput compared to state-of-the-art serverless systems. In addition, we also present the scalability and effectiveness of FaaSTube on a 4-node cluster and GPU servers with various GPU topologies.

Table 1. Real-world inference workflows. GPU functions are underlined, and there are four typical workflow types: *condition*, *sequence*, *fan-in*, and *fan-out*.

GPU workflows	Functions	Type
Traffic monitor [4]	decode, pre/postproc., Yolo-det, ResNet	condition
Auto-driving [42]	decode, denoising, Yolo-seg, bluring	sequence
Video editing [54]	decode, preproc, Yolo-face, ResNet	fan-in
Image editing [15]	decode, denoising, ResNet, AlexNet	fan-out
Social media [11]	decode, preprocess, OCR, Bert	condition
Yelp [53]	Bert, Bert	sequence

2 Background

This section introduces serverless inference workflows and analyzes the problem of current inter-function data passing.

2.1 Serverless Inference Workflow

ML inference applications are increasingly prevalent in daily life [4, 15, 18, 41, 42, 52]. To efficiently deploy inference applications, many studies [38, 48, 49, 53] have developed GPU-enabled serverless inference systems, allowing users to publish ML models as functions that scale resources on demand with workload fluctuations. Compared to managing GPU clusters directly (i.e., serverful deployment), serverless inference enables users to focus on application logic while eliminating the need for overprovisioning.

Real-world inference applications often involve multiple ML models and operations working together to perform complex tasks. In this paper, we study several representative inference applications collected from recent research [4, 11, 15, 42, 53, 54], as shown in Table 1 (detailed in Section 9). In serverless systems, these applications combine GPU functions (gFunc) and CPU functions (cFunc) as serverless inference workflows. Serverless inference workflows can be represented as directed acyclic graphs (DAGs), where each node corresponds to ML models on the GPU or operations on the host, with edges representing dataflow between functions. Similar to serverless workflows in data analysis [24], serverless inference workflows exhibit diverse dataflows, including sequence, condition, and fan-in/fan-out patterns.

2.2 Current Inter-Function Data Passing

Since media data (e.g., images and video frames) can reach hundreds of MB, efficient data passing is crucial in serverless inference workflows. However, passing data between different types of functions in these workflows is non-trivial. Beyond typical cFunc-to-cFunc data passing, there are also host-to-gFunc (where the host represents a cFunc or I/O data in host memory) and gFunc-to-gFunc data passing. Unfortunately, current serverless systems rely on a **host-oriented** data passing approach (Fig. 2(a)), where intermediate data is stored and exchanged via an external storage in host memory, such as remote data storage (e.g., AWS S3 [2]) or local

data storage (e.g., intra-node message queues [24, 25, 29] or shared memory [20, 28, 50]). This method ignores the decoupling of GPU and host memory, forcing the intermediate data of GPU function to be exchanged through host memory, resulting in large overhead.
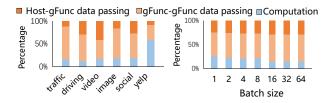
Next, we evaluate the latency of INFless+, which combines the latest serverless inference system, INFless [49], with recent data passing optimizations for serverless workflows in Pheromone [50], i.e., sharing host memory between functions. Experiments are conducted using a set of real-world inference applications (Table 1) on a DGX-V100 GPU server. Additional details can be found in Section 9. Fig. 3 shows that inter-function data passing accounts for up to 92% of the end-to-end latency, with 29% of host-to-gFunc data passing, 63% of gFunc-to-gFunc data passing. This high latency arises from the overhead of copying data between GPU and host memory, as each GPU is connected to host memory via PCIe links with limited bandwidth (12GB/s under PCIe 3.0). The gFunc-to-gFunc transfer overhead is especially high, as it involves two sequential data copies: first from the source GPU to host, then from host to the target GPU. Since functions share host memory, the overhead of cFunc-to-cFunc will be less than 1%. Therefore, it is omitted from the figure.

3 GPU-oriented Data Passing

This section presents the benefits of exploiting various connections (i.e., parallel PCIe links and direct NVLinks) in GPU servers and analyzes the associated challenges.

3.1 Opportunities and Requirements

Modern GPU servers are equipped with high-speed NVLink connections between GPUs and multiple PCIe connections between host and GPUs, facilitating faster data transfer in serverless inference workflows. For instance, in gFunc-to-gFunc data passing, data can be transferred directly via NVLink (25-50 GB/s), avoiding unnecessary copies between the host and GPUs. In host-to-gFunc data passing, part of the data can be routed to a neighboring GPU via NVLink and

Figure 3. Performance analysis of real-world inference workflows on INFless+. (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 host-to-gFunc data passing (top), gFunc-to-gFunc data passing (middle), and computation (bottom).

then transferred to the host in parallel through the neighboring GPU's PCIe link. As shown in Fig 4, DGX-V100 and DGX-A100 servers connect 8 GPUs to the host via 4 PCIe links, allowing parallel transfers with up to 4x bandwidth.

However, existing host-oriented inter-function data passing overlooks these opportunities. This motivates us to propose a GPU-oriented inter-function data passing approach that allows intermediate data to be exchanged or stored directly in GPU memory, thereby avoiding costly data copies between the host and GPUs. Additionally, we leverage various connections available in GPU servers, such as parallel PCIe links and NVLinks, to accelerate inter-function data passing. Nevertheless, designing an efficient GPU-oriented data passing framework for serverless functions needs to meet several key requirements:

R1: *Bandwidth efficiency*: The available bandwidth of the various connections must be fully utilized while avoiding contention among concurrent functions.

R2: *Topology awareness*: There are different connection topologies in modern GPU servers, requiring careful management of transfer scheduling and link usage based on the specific GPU topology during inter-function data passing.
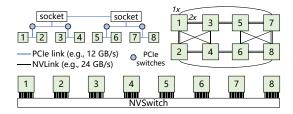
R3: *GPU memory efficiency*: As highlighted by existing research [9, 21], GPU memory is a limited resource; therefore, its usage should be minimized when designing GPU data storage for functions.

R4: *Transparent deployment*: All of the above should be implemented transparently to users, simplifying the development of serverless workflow and hiding the complexities of diverse connections and GPU topologies.

3.2 Technical Challenges

Fulfilling the above requirements still faces several challenges that existing methods fail to address.

Challenge #1: PCIe connections sharing. *Harvesting parallel PCIe links necessitates avoiding bandwidth contention among concurrent functions and implementing effective pinned memory allocation (**R1**).*

When leveraging parallel PCIe links for host-to-gFunc data passing, current PCIe transfer scheduling faces two limitations. First, borrowing multiple PCIe links introduces *interference* when concurrent functions on neighboring GPUs simultaneously transfer data to the host. However, related works [13, 19] that utilize parallel PCIe links focus on exclusive inference systems and overlook scenarios where multiple functions share a GPU server. Additionally, native GPU PCIe scheduling is not optimized for parallel transfers. We evaluated the performance of concurrent workflows in DeepPlan+, which implements parallel PCIe transfers as in DeepPlan [19] on INFless+. Fig. 5(a) shows that running the video and driving workflows concurrently leads to a 5x increase in host-to-gFunc transfer overhead for the driving workflow compared to running them separately. This increase is due to the video workflow's multiple functions loading video blocks simultaneously, saturating the global PCIe bandwidth. Such interference can significantly impact the performance of SLO-critical functions. Second, *pinned memory* is needed to maximize PCIe link utilization, as it increases each link's bandwidth from 3 GB/s to 12 GB/s. However, pinned memory allocation incurs large overhead (e.g., 70 ms for 100 MB), reducing transfer bandwidth to 1 GB/s, as shown in Figure 5(b). This overhead greatly impacts the performance of short-lived functions.

Challenge #2: Various topologies of GPUs and NVLinks. *Function workflows rely on point-to-point transfers, making them more sensitive to topology and NVLink usage (**R1, R2**).*

Modern GPU server have diverse topologies with NVLinks. In high-end GPU servers, GPUs are fully connected via NVSwitch (Fig 4(c)). In contrast, more cost-effective GPU servers commonly exhibit non-uniform topologies with hard-wired NVLinks (Fig 4(b)). Recent production traces [8, 46] indicate that this kind of GPU servers comprise 36% of the cluster, and many cloud providers offer them as rental options (AWS P3 instance [1], Azure NDv2 instance [6], Google cloud N1 instance [14]). In non-uniform topologies, point-to-point transfer bandwidth between GPUs can vary significantly. As shown in Fig. 6(a), 28% of GPU pairs have only half the bandwidth, while 42% lack a direct NVLink and must transfer data over PCIe with a bandwidth of just 7.9 GB/s.
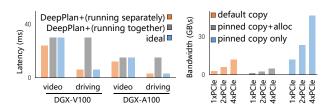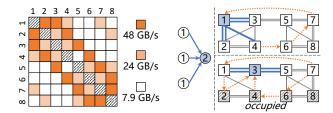

Figure 4. Vairous connection topologies in GPU servers: (a) PCIe connections between host and GPUs, (b) 8 GPUs connected via hard-wired NVLinks like DGX V100 and (c) 8 GPUs connected via switch-based NVLinks like DGX A100.

Figure 5. (a) Comparison of host-to-gFunc passing overhead between separate execution and together execution of video and driving workflows. (b) The Impact of pinned memory in PCIe transfers.

Figure 6. The non-uniform GPU topology. (a) The P2P bandwidth of different GPU pairs in a DGX-V100 server. (b) Different placement of a video workflow. Blue lines show the currently used paths, and orange dotted lines show possible parallel paths.

Figure 7. (a) The object count fluctuates in different videos. (b) The accumulation of intermediate data results in its migration to host memory.

When functions in a serverless workflow are deployed on GPUs with limited point-to-point bandwidth, performance can significantly degrade. Existing methods [7, 37, 45] optimize placement to avoid transfers between these GPUs. However, in complex workflows, e.g., traffic(condition), video (fan-in), and image (fan-out), using bandwidth-limited GPUs is unavoidable due to connection limitations. Consider a video processing workflow that includes three parallel detection functions for frame extraction and an identification function for actors. Even with optimal deployment (top of Fig. 6(b)), this workflow still involves bandwidth-limited GPU pairs, such as G_1 and G_4. In a worst-case (bottom of Fig. 6(b)), optimal deployment may be impossible (e.g., G_2, G_4, G_6 and G_8 are occupied by other workflows), forcing G_7 and G_3 to participate with no NVLink.

This problem cannot be completely solved by placement optimisation alone, as it relies on only the direct NVLink path between GPUs. We propose viewing a GPU server as a network where each GPU pair has parallel NVLink paths. For instance, routing G1-G4 through G4-G6-G7 can double the bandwidth. In addition, G3-G7 can utilize paths like G7-G1-G2-G3 and G7-G6-G4-G3, increasing bandwidth by 6X. Existing multi-GPU communication methods [22, 39], such as NCCL [30], focus on collective communication and also utilize only a single NVLink path for point-to-point transfers.

Challenge #3: Efficient GPU data store. *Fluctuations in function workload and intermediate data size require an elastic GPU data store that can dynamically scale and efficiently migrate data between GPUs and host memory (**R3**).*

To avoid storing data in host memory, a GPU data store is necessary. As many studies highlight [16, 21, 47], GPU memory is limited, so this data store must be space-efficient. However, managing the GPU data store for serverless workflows needs to adapt to two types of fluctuations: 1) *Data size fluctuations*: In media-based workflows, the size of intermediate data varies with the semantic content of the input. For example, in a traffic monitoring workflow, the detection function extracts objects from each frame and passes them to the recognition function. As shown in Fig. 7(a), the number of objects in different video frames fluctuates with

pedestrian and vehicle movements. Moreover, the intermediate data size varies with different input batch sizes (e.g., TensorRT dynamic batching [44]). Therefore, the GPU data store must allocate memory flexibly and dynamically scale based on function's demand. 2) *Memory pressure fluctuations*: Large amounts of cached intermediate data can occupy a lot of GPU memory, particularly when production rates exceed consumption, leading to data accumulation, or when functions face bursty requests. In such cases, the GPU data store must migrate data to host memory as necessary (Fig. 7(b)).

However, existing GPU memory management, such as memory pooling [16, 36] and unified memory [9, 35], are designed for long-running, exclusive tasks (e.g., ML training). Therefore, they often retain significant idle memory blocks and lack the necessary elasticity for serverless functions, particularly in terms of flexible memory recycling mechanisms (Section 7.1), resulting in excessive memory occupation.

Challenge #4: Developer-friendly. *The development of serverless inference workflows is complicated by the diverse distribution of functions and diverse data passing, involving various connections and memory types in GPU servers (**R4**).*

Developing serverless inference workflows presents two challenges. First, *various data storage*: Data may be stored in either host or GPU memory, each requiring distinct management and retrieval mechanisms. Host-side storage typically uses shared memory addresses as Pheromone [50] or message queue keys (e.g., Redis), whereas GPU-side storage relies on CUDA IPC handles. Second, *various transfer methods*: Serverless functions are distributed across nodes and GPUs, necessitating multiple data transfer methods—such as intra-GPU, inter-GPU, host-GPU, and inter-node—utilizing PCIe, NVLink, or network connections. In addition, optimizations such as parallel and pipelined data transfers need to be selectively implemented to enhance transfer efficiency. What is worse, serverless functions typically run in containers [31], which obscures critical information (e.g., GPU topology and function placement), preventing developers from configuring efficient data passing in GPU servers.

4 Overview of FaaSTube

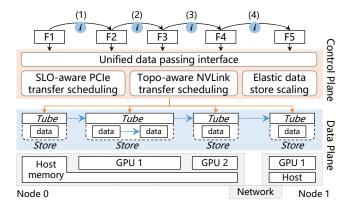We propose *FaaSTube*, a GPU-oriented data transfer framework designed for serverless inference workflows. As shown

Figure 8. The overview of FaaSTube.

in Fig. 8, FaaSTube acts as a transparent "tube" across GPUs, facilitating efficient data storage and transfer for functions.

Functions initiate data passing through a *unified data passing interface (Challenge #4)*, which simplifies the management of various data passing in serverless inference workflows and determines the optimal data transfer methods for different scenarios. FaaSTube's control plane oversees transfer scheduling and storage management through three core components: First, *SLO-aware PCIe transfer scheduling (Challenge #1)*, which partitions the bandwidth of global PCIe connections to ensure performance isolation among concurrent functions; Second, *Topology-aware NVLink transfer scheduling (Challenge #2)*, which exploits parallel NVLink paths to facilitate robust inter-function data passing across different GPU topologies; and Third, *Elastic data store (Challenge #3)*, which elastically manages GPU memory and migrates intermediate data to reduce GPU memory usage. The data plane executes the decisions of the control plane, with a daemon thread running on each GPU and host that acts as both a *tube*, responsible for transferring data within GPU servers, and a *Store*, responsible for managing GPU memory and intermediate data.

```
1  void FaaSTube.unique_id(char** data_index);
2  void FaaSTube.fetch(char** index, void* input);
3  void FaaSTube.store(char** index, void* output,
       int response=0);
```

Listing 1. The APIs of user library

5 Unified Data Passing Interface

First, FaaSTube offers a unified data passing interface (List. 1) to simplify the composition of serverless inference workflows. Second, FaaSTube enhances scalability through a two-tier distributed design.

5.1 Unified Programming Interface

FaaSTube provides a unified data ID to simplify managing host-side storage (e.g., shared host memory, Redis servers) and GPU-side storage (e.g., CUDA IPC handles). FaaSTube

maintains a mapping between data IDs and corresponding address information, allowing functions to transfer data by simply passing the data ID. When a function calls *store()*, FaaSTube saves the data locally by default, such as in the current GPU's memory, and updates the mapping table. When a function calls *fetch()*, FaaSTube locates the data in GPU servers using the data ID and selects an appropriate transfer method based on the locations of data and the requesting function. FaaSTube supports four transfer methods (Fig. 8):

- *Host-GPU*: For data transfer between host and GPU memory, the data is partitioned according to the number of PCIe connections in the GPU server and transferred in parallel using a pipelined approach. Specifically, the data is divided into smaller chunks, and once the first chunk reaches the neighboring GPU, it is immediately transferred to the target GPU. Notably, PCIe bandwidth usage for each function is managed by the PCIe scheduling module.
- *Intra-GPU*: When a function fetches data on the same GPU, it can be directly mapped and copied into the function's address space using CUDA IPC [32].
- *Inter-GPU*: When a function fetches data from a different GPU within the same node, FaaSTube will select appropriate NVLink paths based on the GPU topology. For example, on a non-uniform GPU topology, FaaSTube will use multiple NVLink paths to enable parallel data transfer.
- *Inter-node*: When a function fetches data across different nodes, the data is transferred over the network. This involves copying the data to host memory, transmitting it over the network to the target node's host memory, and subsequently copying it to the target GPU. Note that FaaSTube conducts these data copies in a pipelined manner, whereas the host-oriented approach conducts them sequentially. While high-end GPUs support direct GPU RDMA, many cost-effective GPU servers lack this feature. Further exploration of GPU-direct RDMA is left for future work.

5.2 Scalable Distributed Framework

To ensure system scalability, FaaSTube uses a two-level mapping table: a local table for each node and a global table maintained by a central node. Functions within each node first query and update their local table; if the data ID is not found, they then query the global table. This method minimizes cross-node messaging and reduces latency. To ensure consistency, local mapping tables are periodically synchronized with the global table. Additionally, FaaSTube and functions use a shared communication channel (i.e., Linux pipe) for fast interface invocation, further reducing latency.

6 Efficient Transfer Scheduling

We present how FaaSTube schedules inter-function data passing to fully utilize various connections in GPU servers.
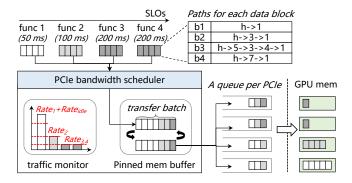
Figure 9. SLO-aware PCIe data transfer scheduling.

6.1 SLO-aware PCIe Transfer Scheduling

To leverage global PCIe connections within GPU servers while avoiding contention, FaaSTube abstracts global PCIe bandwidth and regulates each function's usage similar to rate control in computer networking [47], achieving dynamic PCIe bandwidth partition among functions.

Fig. 9 shows the process of PCIe transfer scheduling in FaaSTube. First, each function's data is divided into smaller chunks (default size is 2MB) to enable fine-grained transfer control. FaaSTube allocates the PCIe bandwidth required by each function to meet its SLO requirements and triggers the transfer of data chunks from different functions in proportion. Finally, FaaSTube employs a circular pinned memory buffer shared among functions to improve transfer efficiency.

Function transfer rate control. FaaSTube 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 each function's SLO. Let L_{slo} denote the function's SLO, and L_{infer} denote its inference computation latency. The $Rate_{least}$ is defined as $data\_size/(L_{slo} - L_{infer})$. The predictability of DNN inference has been extensively studied [5, 10, 48, 53]. Given that FaaSTube utilizes temporal GPU sharing among functions, interference is minimized. Consequently, offline profiling can effectively guide transfer control to meet the latency SLOs for each functions.

FaaSTube monitors the transfer rate of each function's data block in real time to ensure it remains above $Rate_{least}$. FaaSTube 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 PCIe bandwidth in the GPU server, $Rate_{idle} = BW_{all} - \sum_{i=0}^{all\_funcs} Rate_{least}^i$. FaaSTube 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.

Batched transfer triggering. Since data chunk transfers cannot be interrupted once triggered, initiating all transfers simultaneously prevents newly arrived functions from

---

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 $Paths$

1 **while** $path == null$ **do**
2      $path \leftarrow$ next_shortest_path(BW_{nxn}, g_s, g_d);
3   **if** all edges in path is idle **then**
4          $Paths \leftarrow path$;
5   Update($BW_{nxn}, path, func$);
6   **if** $BW_{out}(g_s) == 0 \cup BW_{in}(g_d) == 0$ **then**
7   break;

8 **if** $BW_{out}(g_s) \neq 0 \cap BW_{in}(g_d) \neq 0$ **then**
9   **while** $path == null$ **do**
10          $path \leftarrow$ next_busy_path(BW_{nxn}, g_s, g_d);
11   bandwidth_balancing($path, func, BW_{nxn}$);
12          $Paths \leftarrow path$;
13   **if** $BW_{out}(g_s) == 0 \cup BW_{in}(g_d) == 0$ **then**
14   break;

15 **return** $Paths$;

---

preempting bandwidth. Conversely, triggering each chunk individually incurs additional overhead. Therefore, FaaSTube triggers data chunk transfers in batches (5 chunks by default), allowing newly arrived functions to preempt bandwidth by including their data chunks in subsequent batches.

Circular pinned mem buffer. To minimize the overhead of pinned memory allocation, a naive approach would cache all required pinned memory for each function; however, this requires substantial pre-allocation and may impair host system performance [34]. Instead, FaaSTube employs a circular pinned memory buffer shared among functions, in conjunction with with batched data transfers, enabling different batches to reuse this fixed buffer and reducing the amount of cached pinned memory.

6.2 Topology-aware NVLink Transfer Scheduling

FaaSTube leverages parallel NVLink paths to enhance point-to-point data passing in non-uniform topologies. FaaSTube introduce a contention-aware path selection algorithm that optimizes NVLink usage while minimizing bandwidth contention from other functions (avoiding path duplication).

Given a serverless workflow, FaaSTube first applies the placement strategy in MAPA [37] to assign functions to GPUs and maximise NVLink connections between functions. After the function placement is determined, FaaSTube first allocates direct connection paths between the GPUs included in the serverless workflow. If these direct NVLinks are already occupied by other functions, FaaSTube will force the other functions to release the path and re-plan other paths.

Then, FaaSTube searches for available free NVLink paths for each inter-GPU data transfer in the serverless workflow, starting with the GPU pair with the least bandwidth. FaaSTube maintains a bandwidth usage matrix $BW(g, b)$, where g represents GPUs and b is the available bandwidth between them. FaaSTube continuously monitors and updates global bandwidth usage in real-time on this matrix $BW(g, b)$, which is used to guide path selection.

As shown in Algorithm 1. the selection process involves: 1) FaaSTube 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. 2) If all free paths are exhausted and the outgoing bandwidth of g_s and incoming bandwidth of g_d are not saturated, FaaSTube searches busy paths to see if bandwidth can be balanced between the current function and the one occupying the path (lines 8-14). FaaSTube 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 pipelined method as in PCIe transfers, and batched data transfers to adapt to path changes. However, the difference is that the FaaSTube needs to adapt to the different NVlink bandwidths (24 or 48 GB/s) of each path, so the FaaSTube transmits data chunks proportional to each path's bandwidth.
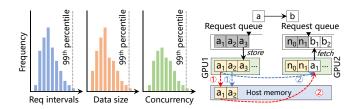
7 Elastic GPU Data Store

To efficiently manage the GPU data store, FaaSTube 1) provides an auto-scaling memory pool on each GPU, which can elastically scale based on the actual demand of functions, and 2) intelligently migrates data between host and GPU memory based on request queues.

7.1 Auto-scaling Memory Pool

As noted in Challenge #3, fluctuating intermediate data sizes necessitate on-demand memory allocation in the GPU data store. However, temporary memory allocation introduces significant overhead in data passing, as native GPU allocations (e.g., cudaMalloc and cudaFree) incur millisecond-level delays. Our experiments show that temporarily allocating memory increases data passing latency by 19% (Section 9.2.1).

Memory pooling can mitigate this issue; however, existing methods lack elasticity, leading to excessive memory consumption in serverless environments. ML frameworks like PyTorch [36] and TensorFlow [3] cache pre-allocated

Figure 10. (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).

memory blocks for later reuse but do not actively release unused memory blocks. When function workloads and intermediate data sizes vary dynamically, this approach leads to up to 4X memory occupation than actual demand in our experiments. While PyTorch permits manual reclamation of memory pools, it reclaims all memory blocks, introducing overhead in future allocations. Recent work, GMlake [16], reduces fragmentation in the memory pool by using CUDA virtual memory and unified 2MB memory chunks, but it still lacks elastic memory reclamation. Moreover, costly IPC operations on each chunk in GMlake introduce large overhead in data passing (up to 45ms in our experiments).

FaaSTube introduces a new memory pre-warming strategy to dynamically rightsize the GPU memory pool for functions. Inspired by keep-alive strategies [40, 49] used in function pre-warming, which track request intervals to define the duration each function stays in memory. However, the new challenge faced by FaaSTube is that the intermediate data size for the same function is fluctuating and may accumulate as function workload increases (*Challenge #3*). To accurately estimate memory requirements (Fig.10(a)), FaaSTube tracks not only request intervals ($R_{window} = Interval^{99th}$), but also the size of intermediate data ($R_{size} = Data_size^{99th}$), and the degree of data accumulation ($R_{con} = Concurrency^{99th}$). 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 $MemPool\_size = \sum_{func} Data\_size \cdot 1_{\{R_{window} \cap t \neq \varnothing\}}$, where 1_A is an index function of events, being 1 when event A is true and being 0 otherwise. To accommodate bursty requests, FaaSTube maintains a minimum memory pool threshold, such as 300 MB in the experiment, instead of reducing it to 0.

7.2 Smart Migration Guided by Request Queues

Efficient data migration is crucial when intermediate data accumulates in GPU memory. Existing migration approaches, such as Unified Memory [9, 21, 35], rely on an LRU eviction strategy that removes earlier-stored (cold) data first. However, this method is ill-suited for managing intermediate

data in serverless workflows, where earlier-stored data is often accessed sooner due to its downstream functions are enqueued earlier. This leads to unnecessary overhead from reloading data from host memory. For instance, as shown by the red line in Fig.10(b), the LRU strategy evicts 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 delays.

Instead, FaaSTube adopts a queue-aware data migration approach, prioritizing the migration of intermediate data whose downstream functions are further back in the request queue. For example, as shown by the blue line in Fig.10(b), the output data of function a_2 is migrated before a_1's output. Moreover, FaaSTube promptly clears intermediate data that is no longer needed and proactively reloads previously migrated data back to the GPU when sufficient memory becomes available. For instance, after a_1's output is processed, a_2's output is reloaded into GPU memory. In FaaSTube, data migrations are automatically triggered by memory pressure, such as reaching the data store's capacity limit (set to 1GB per GPU in our experiments), and are executed asynchronously with function execution.
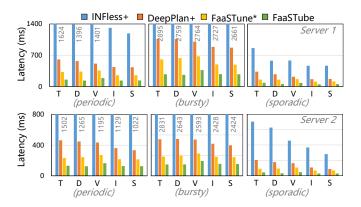
8 Implementation

FaaSTube is built on INFless [49], a state-of-the-art serverless inference system. It comprises 5K lines of C++ code.

Data transfer management: Similar to Pheromone [50], FaaSTube mounts a shared host volume to each function for efficient data and message exchange on the host side. On the GPU side, FaaSTube launches a daemon thread for each GPU to handle data passing requests from functions. This thread uses CUDA IPC to establish a GPU memory mapping with the function, serving as a *transfer buffer*. Since functions run in virtualized environments like containers [31], the underlying GPUs are invisible to them. Data passing between functions is achieved via storing and fetching data through the transfer buffer. The daemon thread is responsible for transferring data between GPUs. The daemon thread manages data transfer between GPUs, utilizing multiple GPU streams to transfer data in parallel across different directions.

Function scheduling: FaaSTube follows the function scheduling in FaasFlow [24] to minimize inter-node transfers in serverless workflow. Intra-node function placement follows existing strategy MAPA [37]. Moreover, To mitigate performance impacts of function cold-starts, FaaSTube pre-warms required functions and models, as in SHEPHERD [52].

9 Evaluation

Setup. Most experiments are conducted on two kinds of AWS GPU servers. **Server 1** (p3.16xlarge): This server has 8 NVIDIA V100 GPUs with NVLinks, a Xeon E5-2686 v4 CPU with 32 virtual CPUs, and 244GB of memory. **Server 2**

Figure 11. Comparison of the end-to-end latency on various workloads. Server 1 has 8xV100 and server 2 has 8xA100.

(p4d.24xlarge): This server has 8 NVIDIA A100 GPUs with NVSwitch, a Xeon Plati. 8275CL CPU, and 1152GB of mem.

Real-world inference workflows. We conduct experiments using seven typical inference applications collected from the latest studies, as detailed below and in Table 1.

- *Traffic* (T): Following Boggard [4], 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 [42], 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 [54], 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 [15], we implement an image classification workflow that first denoises the image, then applies multiple classification models simultaneously, and aggregates the results to improve accuracy.
- *Social* (S): Following InferLine [11], we implement a social media workflow that identifies the text in user posts with an OCR model and analyzes it for sensitive content.
- *Yelp* (Y): Following Astraea [53], we implement a comment generation workflow that first analyzes comments for malicious content and then generates appropriate replies.

All pre-processing and post-processing are performed on the GPU using NVIDIA's CV-CUDA library [33]. The datasets of these workflows are from [42, 53].

Baselines. We compare FaaSTube to the following baselines:

- *INFless+*: This baseline combines INFless [49], a state-of-the-art serverless inference system, with latest data passing optimization [27, 50] that accelerates data exchanges by enabling functions to share host memory.
- *DeepPlan+*: Derived from DeepPlan [19], this baseline uses parallel PCIe connections to accelerate data transfers in

serverless workflows. Since DeepPlan focuses on model loading in monolithic inference systems, we incorporate its core ideas into INFless+. We refer to it as DeepPlan+.

- *FaaSTube**: This baseline utilizes all available connections (NVLinks and parallel PCIe links) in GPU servers to accelerate data passing in serverless inference workflows, but lacks the further optimizations provided by FaaSTube. We refer to it as FaaSTube*.
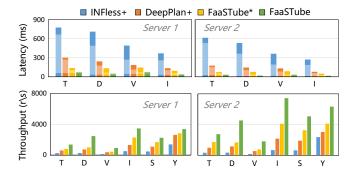
Workloads. We simulate the dynamic invocation of inference workflows, using production traces from Azure Function [40], which is a widely used trace in serverless research [27, 49, 52]. There are three typical types of request arrival patterns in the production trace including sporadic, periodic, and bursty. As in Aquatope [54], the load is scaled according to the resource availability (e.g., number of nodes and GPUs).

9.1 Overall performance of FaaSTube

In this section, we compare FaaSTube to state-of-the-art systems across a wide range of workflows.

End-to-end latency. FaaSTube can reduce the end-to-end latency of workflows by up to 90%. Fig. 11 shows the P99 latency across various workflows under different production workloads. FaaSTube consistently achieves the lowest latency, reducing it by 86%-90% compared to INFless+ and by 62%-79% compared to DeepPlan+. Both INFless+ and DeepPlan+ rely on host-oriented data passing, with DeepPlan+ using parallel PCIe connections, which still incur significant overhead due to redundant transfers between GPU and host memory. Furthermore, FaaSTube reduces latency by 43%-63% compared to FaaSTube*, which merely utilizes all available connections in GPU servers. This improvement is due to FaaSTube's optimized transfer scheduling and memory management during data passing.

Latency breakdown. FaaSTube can reduce the data passing overhead of workflows by up to 98%. To provide a detailed analysis of data passing overhead, we break down the P99 execution latency,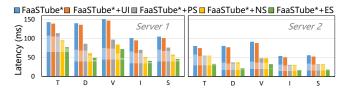 excluding queuing time. As shown in Fig. 12(a), the latency is divided into three parts: host-to-gFunc data passing (top), gFunc-to-gFunc data passing (middle), and computation time (bottom). Results for bursty workloads are presented, as other workloads exhibit similar patterns. FaaSTube achieves the lowest data passing overhead, reducing it by 93%–98%, 90%–94%, and 70%–88% compared to INFless+, DeepPlan+, and FaaSTube*, respectively. DeepPlan+ leverages parallel PCIe data passing, resulting in lower latency than INFless+, which relies on a single PCIe link. FaaSTube* reduces gFunc-to-gFunc data passing overhead by utilizing NVLink. FaaSTube further reduces both types of data passing overhead through topology-aware transfer scheduling and efficient management of pinned memory and GPU memory during data passing.

Throughput. FaaSTube can improve throughput by up to 12X. We measure the maximum throughput of these workflows. Fig. 12(b) shows that FaaSTube achieves the highest throughput, outperforming INFless+, DeepPlan+ and FaaSTube* by 2.4X–12X, 1.7X–3.9X and 1.3X–2.7X, respectively. The improvements are more pronounced in driving and video workflows, where large volumes of media data are transferred between functions and returned to host memory, making both gFunc-to-gFunc and host-to-gFunc data transfers significant bottlenecks. FaaSTube mitigates these bottlenecks by optimizing PCIe bandwidth utilization with circular pinned memory buffers and optimizing inter-GPU data transfers through topology-aware NVLink scheduling and elastic GPU storage. This shows that the throughput of serverless workflows is highly dependent on the efficiency of data transfers within GPU servers.
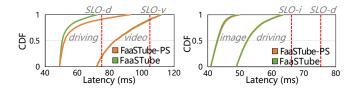
9.2 Optimizations in FaaSTube

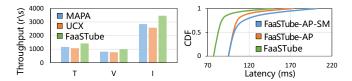Here, we show the effectiveness of each technique in FaaSTube.

9.2.1 Ablation Study. FaaSTube employs several techniques to enhance data passing efficiency: a unified data passing interface (UI), fine-grained PCIe scheduling (PS), topology-aware NVLink scheduling (NS), and an elastic data store (ES). We evaluate each technique by measuring the latency when enabling each one incrementally. As in the previous experiments, we report P99 latency under bursty loads, excluding queuing time. As shown in Fig. 13(a), on the server 1, FaaSTube with all optimizations (rightmost bar)

Figure 12. (a) Latency breakdown and (b) comparison of the maximum throughput.

Figure 13. The latency when enabling each component in FaaSTube one by one.

Figure 14. The effectiveness of FaaSTube's PCIe transfer scheduling.

Figure 15. (a) The effectiveness of FaaSTube's parallel NVLink transfer and (b) The effectiveness of FaaSTube's elastic data store.

reduces latency by 46%-65% compared to FaaSTube*. Specifically, UI, PS, NS, and ES reduce latency by up to 2.5%, 20%, 23%, and 19%, respectively. UI's local messaging channel eliminates RPC overhead, while PS's circular pinned mem buffer reduces the cost of allocation in host-to-gFunc data passing. NS optimizes gFunc-to-gFunc data passing via parallel NVLink paths. Additionally, since host-to-gFunc data passing involves NVLink transfers with neighboring GPUs, NS helps reduce latency in those transfers as well. ES further reduces gFunc-to-gFunc data passing overhead by enabling rapid memory allocation through an auto-scaling memory pool and smartly migrating accumulated data to avoid the overhead of fetching from host (a 1GB limit is set for the GPU data store in the experiments). As shown in Fig. 13(b), on the server 2, FaaSTube achieves a latency reduction of 57%-72%. UI, PS, NS, and ES reduce latency by up to 3.2%, 30%, 0%, and 39%, respectively. The increased computing power of the A100 GPU makes data passing overhead more pronounced, particularly due to the high allocation costs of pinned memory and GPU memory. Additionally, the fully connected GPU topology among A100 GPUs limits the performance gains from NS.

9.2.2 PCIe Transfer Scheduling. To demonstrate the effectiveness of FaaSTube's PCIe transfer scheduling (PS) in achieving performance isolation between functions, we conduct mixed workload experiments using two workflow pairs on server 1. The SLO for each workflow is set to 1.5X its independent runtime. We compare FaaSTube with FaaSTube-PS, which employs native PCIe bandwidth sharing as in Deep-Plan+. Both workflows run under bursty workload, consistent with previous experiments. Fig. 14(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. Under native PCIe bandwidth sharing, driving's latency is increased due to interference from the video workflow. In contrast, FaaSTube reduces latency by 32% and improves SLO compliance through fine-grained control of PCIe bandwidth usage for each function. Fig. 14(b) shows results for a low-contention scenario, where driving is paired with another real-time image workflow. In this case, FaaSTube and FaaSTube-PS performed identically, indicating that FaaSTube introduces no additional overhead in PCIe transfers.

9.2.3 NVlink Transfer Scheduling. To demonstrate the advantage of FaaSTube's parallel NVLink transfer over methods like MAPA [37], which only optimize placement on non-uniform GPU topologies, we evaluate their maximum throughput on server 1. As shown in Fig. 15(a), FaaSTube can improve the throughput of video, image and traffic workflows by 18%, 13%, and 17% respectively compared to MAPA. We also compare a related work UCX [43], which manually optimizes point-to-point transfers for HPC tasks on a 4xG-PUs server. Due to the lack of topology-aware path selection in FaaSTube, its throughput is even lower than MAPA's.

9.2.4 Elastic Data Store. To demonstrate the advantage of FaaSTube's auto-scaling memory pool (AP) and smart migration guided by request queues (SM) in its elastic data store, we first analyze the impact of each component on latency. As shown in Fig. 15(b), the auto-scaling memory pool reduces latency by an average of 19%. For burst requests, smart data migration further reduces tail latency by 14% by migrating non-immediately accessed data based on the request queue, avoiding the overhead of loading from the host memory.

Next, we compare FaaSTube with the latest pooling schemes in PyTorch [36] and GMlake [16]. As shown in Fig. 16(a-b), PyTorch caches all allocated memory blocks, leading to large memory occupation and fragmentation (e.g., a 100MB block cannot accommodate a 120MB request, requiring a new allocation) and does not actively free unused memory. GMlake mitigates fragmentation by using CUDA virtual memory and unified 2MB chunks but still lack active memory release, leading to large GPU memory consumption. In contrast, FaaSTube dynamically scales its memory pool based on function load and intermediate data size, thereby reducing memory usage.
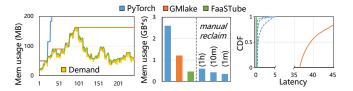
While PyTorch allows manual memory pool reclamation, it reclaims all memory blocks, causing overhead in subsequent allocations. We evaluated memory allocation performance in PyTorch memory pool at varying reclamation frequencies (i.e., every hour, every ten minutes, and every minute). As shown in Fig. 16(b-c), while manual reclamation reduces memory usage, it increase tail latency by up to 4X (blue dashed curve). In contrast, FaaSTube dynamically scales the memory pool, effectively balancing memory usage and performance. The overhead of GMlake arises from the extra

IPC operation costs introduced by each 2MB block during data sharing between the function and GPU data store.
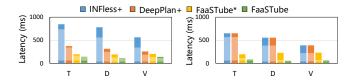
9.3 Scalability

In this section, we evaluate the scalability and applicability of FaaSTube in a cluster and a GPU server without NVLink.
Inter-node performance. Fig. 17(a) shows the latency of deploying workflows on a cluster consisting of four AWS p3.16xlarge servers. As in the previous experiment, we report the P99 latency under bursty load, excluding queuing time. FaaSTube achieves optimal performance, reducing latency by 85%, 63%, and 39% compared to INFless+, Deep-Plan+, and FaaSTube* respectively. The improvement of FaaS-Tube comes from two parts: 1) intra-node transfers. Existing serverless workflow scheduling [24] minimizes cross-node communication, resulting in at most one inter-node data transfer in a function workflow. Consequently, FaaSTube's PCIe and NVLink transfer scheduling remains effective for the remaining intra-node data transfers. 2) Inter-node transfers. INFless+ and DeepPlan+ utilize a host-oriented method, sequentially copying data between the source GPU, host storage on both nodes, and the target GPU, leading to significant cumulative latency. In contrast, FaaSTube maintains a comprehensive view of the cluster's connections, enabling rapid data transfer between GPUs across nodes using a pipelined approach, thereby reducing overhead. Additionally, FaaS-Tube's optimizations in pinned memory and GPU memory allocation further decrease inter-node data transfer overhead.
Other GPU topologies. We evaluate FaaSTube's performance on a server with 4 A10 GPUs, which lack NVLink connections. Fig. 17(b) shows that FaaSTube achieves the lowest latency, reducing it by up to 90%, 90%, and 75% compared to INFless+, DeepPlan+, and FaaSTube*, respectively. On the A10 GPU server, each GPU can transfer data with host memory via only one single PCIe link, so DeepPlan+ and INFless+ perform the same. Additionally, INFless+ and Deep-Plan+ are constrained by their host-oriented data passing methods, where data is sequentially copied to host storage and then to the target GPU, leading to latency accumulation. FaaSTube transfers data in a pipelined manner. FaaSTube*

Figure 16. (a) Tracing the memory usage timeline, (b) Comparison of memory occupation and (c) Comparison of memory pooling efficiency. The blue dashed line refers to the performance of the pytorch memory pool under manual collection at different frequencies (1 hour, 10 mins and 1mins).

Figure 17. (a) Comparison of inter-node performance and (b) Comparison of performance on a 4xA10 GPUs server.

remains constrained by GPU memory and pinned memory allocation overhead during data transfers.

10 Related Work

Multi-GPU communication. Existing multi-GPU communication methods [7, 22, 30, 37, 39] primarily focus on collective communication primitives, such as allReduce, reduceScatter, and allGather, which use parallel rings or trees structures to integrate all GPUs. These methods overlook GPU point-to-point transfers on non-uniform topologies, which is needed by serverless inference workflows. For instance, NCCL's ncclSend/ncclRecv operations only use the direct NVLink path. Moreover, existing methods [7, 37, 45] for non-uniform topologies only addresses the problem partially through placement optimizations. In contrast, FaaSTube enables robust data transfers on non-uniform topologies by leveraging parallel NVLink paths with contention-aware path selection.

GPU memory management. Existing methods focus on pooling memory and unifying multi-level memory. Systems like GMlake [16] use CUDA virtual memory to reduce fragmentation in memory pooling, while CUDA UVM [35], HUVM [9] and DeepUM [21] address GPU memory limits by swapping data between GPU and host memory. However, these methods are designed for long-running training tasks and lack flexible memory recycling, which can lead to large memory occupation in serverless context. In contrast, FaaS-Tube dynamically scales the memory pool according to the function's needs. Moreover, due to the limited visibility of underlying GPU resources, existing memory management techniques are not applicable to serverless functions.

Serverless workflow optimizations. Current research primarily focus on traditional workflows. Systems such as Pheromone [50] and Unum [26] optimize function composition, while Dataflower [25] and Fuyao [27] improve data transfer in host memory, Nightcore [20] minimizes runtime redundancy, and FaasFlow [24] enhances function scheduling. Although these methods are orthogonal to FaaSTube, none address the need for efficient GPU-oriented data transfer in serverless inference workflows. In contrast, FaaSTube fully exploits various connections in GPU servers for serverless functions.

GPU-enabled serverless system. As demand for GPUs grows, many GPU-enabled serverless systems have emerged,

typically categorized by how they share GPUs: temporal sharing (e.g., DGSF [12] and FaaSwap [51]) and spatial-sharing (e.g., StreamBox [48] and Llama [38]). However, these systems do not optimize data transfers between GPU functions. Although FaaSTube follows a temporal-sharing model, its design can be extended to other GPU-enabled serverless systems.

11 Conclusions

We propose FaaSTube, an efficient GPU-oriented data passing framework for serverless inference workflows, enabling direct data exchange in GPU memory. FaaSTube optimizes transfer scheduling over PCIe and NVLinks to accelerate host-to-GPU and GPU-to-GPU data transfers among functions. Additionally, it implements an elastic memory pool on each GPU that automatically scales based on actual data passing demand of serverless functions, reducing GPU memory consumption.

References

[1] 2007. AWS S3. https://aws.amazon.com/ec2/instance-types/p3/.
[2] 2007. ZeroMQ. https://aws.amazon.com/s3/.
[3] 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. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.
[4] Neil Agarwal and Ravi Netravali. 2023. Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 933–951. https://www.usenix.org/conference/nsdi23/presentation/agarwal-neil
[5] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00073
[6] Azure Cloud. [n. d.]. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndv2-series?tabs=sizebasic.
[7] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 62–75. https://doi.org/10.1145/3437801.3441620
[8] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/3342195.3387555
[9] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 625–638. https://www.usenix.org/conference/atc22/presentation/choi-sangjin
[10] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom
[11] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285
[12] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 739–750. https://doi.org/10.1109/IPDPS53621.2022.00077
[13] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu
[14] Google Cloud. [n. d.]. https://cloud.google.com/compute/docs/gpus/create-gpu-vm-general-purpose.
[15] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1041–1057. https://www.usenix.org/conference/nsdi22/presentation/gunasekaran
[16] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (<conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 450–466. https://doi.org/10.1145/3620665.3640423
[17] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han
[18] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A Cost-Effective Deep Learning Inference System. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 624–638. https://doi.org/10.1145/3472883.3486993
[19] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3552326.3567508

[20] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[21] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3575693.3575736

[22] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 2024. TCCL: Discovering Better Communication Paths for PCIe GPU Clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 999–1015. https://doi.org/10.1145/3620666.3651362

[23] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[24] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*. 782–796. https://doi.org/10.1145/3503222.3507717

[25] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2024. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (<conf-loc>, <city>Vancouver</city>, <state>BC</state>, <country>Canada</country>, </conf-loc>) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 57–72. https://doi.org/10.1145/3623278.3624755

[26] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[27] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (<conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 431–447. https://doi.org/10.1145/3620666.3651327

[28] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. https://doi.org/10.1145/3627703.3629568

[29] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[30] NVIDIA Collective Communications Library NCCL. [n. d.]. https://developer.nvidia.com/nccl.

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

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

[33] Nvidia CV-CUDA. [n. d.]. https://github.com/CVCUDA/CV-CUDA.

[34] Nvidia Pinned Memory Performance Issue. [n. d.]. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html.

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

[36] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[37] Kiran Ranganath, Joshua D. Suetterlein, Joseph B. Manzano, Shuaiwen Leon Song, and Daniel Wong. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 99, 14 pages. https://doi.org/10.1145/3458817.3480853

[38] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3472883.3486972

[39] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 593–612. https://www.usenix.org/conference/nsdi23/presentation/shah

[40] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[41] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[42] Sudipta Saha Shubha and Haiying Shen. 2023. AdaInf: Data Drift Adaptive Scheduling for Accurate and SLO-guaranteed Multiple-Model Inference Serving at Edge Servers. In *Proceedings of the ACM SIGCOMM 2023 Conference* (<conf-loc>, <city>New York</city>, <state>NY</state>, <country>USA</country>, </conf-loc>) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3603269.3604830

[43] Amirhossein Sojoodi, Yiltan H. Temucin, and Ahmad Afsahi. 2024. Enhancing Intra-Node GPU-to-GPU Performance in MPI+UCX through Multi-Path Communication. In *Proceedings of the 3rd International Workshop on Extreme Heterogeneity Solutions* (Edinburgh, United Kingdom) *(ExHET '24)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/3642961.3643800

[44] TensorRT Dynamic Shape. [n. d.]. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[45] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 172–186. https://proceedings.mlsys.org/paper_files/paper/2020/file/cd3a9a55f7f3723133fa4a13628cdf03-Paper.pdf

[46] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[47] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[48] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. 2024. StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 59–73. https://www.usenix.org/conference/atc24/presentation/wu-hao

[49] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[50] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1489–1504. https://www.usenix.org/conference/nsdi23/presentation/yu

[51] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *arXiv preprint arXiv:2306.03622* (2023).

[52] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[53] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: towards QoS-aware and resource-efficient multi-stage GPU services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 570–582. https://doi.org/10.1145/3503222.3507721

[54] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3567955.3567960