FaaShare: Enabling Generic Low-Latency and SLO-Aware GPU Sharing in Serverless Computing

Zhuolong Jiang*, Zinuo Cai*, Yumou Liu, Ruhui Ma†, *Member, IEEE*, Rajkumar Buyya, *Fellow, IEEE*

***Abstract*—Serverless computing, also known as Function as a Service (FaaS), is an emerging cloud computing paradigm characterized by high scalability and low maintenance requirements, which has attracted significant attention from academia and industry. However, existing serverless platforms lack effective support for high-performance accelerators such as GPUs, preventing computationally intensive workloads from benefiting from this promising model. This can result in high latency response, failure of Service Level Objectives (SLO), and low GPU utilization for serverless functions. Consequently, inefficient utilization of high-performance accelerators has become a major bottleneck in the performance of serverless computing systems with GPU support.**

In this paper, we propose FaaShare, an efficient serverless computing platform that addresses three challenges: high latency, violation of SLO, and GPU sharing. First, we introduce a new container abstraction called `MSContainer`, which provides a pre-warm environment for function execution to reduce cold start latency. Second, we propose a temporal-spatio priority queue scheduling mechanism that dynamically adjusts function priorities to ensure service level objectives. Finally, we utilize Bayesian optimization to allocate optimal computing resources for serverless functions to enhance GPU sharing performance. Extensive evaluation proves that the proposed framework substantially outperforms state-of-the art systems by 23%, 33%, and 18% on job completion time, makespan, and system throughput, meeting the low latency goals of serverless services.

***Index Terms*—Serverless Computing, Cloud Computing, GPU Sharing.**

I. Introduction

Serverless computing [1]–[7] is rapidly emerging as a promising paradigm for the next generation of cloud computing. Compared with existing cloud paradigms, including Infrastructure as a Service (IaaS) [8], Platform as a Service (PaaS) [9], serverless computing, known as Function as a Service (FaaS), takes function as the minimum execution unit, providing a more lightweight resource abstraction from the perspective of developers. For cloud vendors, serverless computing can improve resource utilization by fully controlling the scheduling of functions and promoting their cloud services for conveniences, such as object storage and logging tools. Due to the enormous potential of serverless in lightweight parallelism, academia and industry also utilize serverless platforms to deploy computation intensive applications like large language models (LLMs) [10],distributed model training [11]–[15], serving [16]–[19], video processing [20] and scientific computing [21].

To facilitate the GPU usage in serverless computing, existing methods have made efforts, but there are still limitations. For the high latency of GPU usage in serverless, INFless [16] reduces the cold start of functions at the container level. DGSF [22] reduces latency at the process level by preloading the CUDA runtime to decrease the execution time of the function's initial invocation, but it overlooks the loading time of other third-party libraries and the container restart time due to program errors. From the perspective of service level objectives (SLO) guarantee, FaaSwap [23] proposes an SLO-aware request queueing policy, but it is only used for model inference. To improve GPU utilization in serverless, Elasticflow [24] specifically considers deep learning training scenarios in serverless and improves GPU utilization by dynamically allocating resources based on task requirements. However, the unit of resource allocation remains a single GPU, which results in low GPU utilization. Although FaST-GShare [25] allows GPU sharing for serverless functions in inference scenarios by adopting Multi-Process Service (MPS), it does not take into account resource contention among tasks. Thus, it is necessary to design a low latency and SLO-aware GPU sharing mechanism for GPU-based high-performance computing tasks in serverless.

However, it is not trivial to enable low latency and SLO-aware GPU sharing in serverless computing. We implement a naive GPU-enabled serverless computing platform based on Apache OpenWhisk* as our baseline and identify three critical challenges. The first challenge is the high latency compared with the prompt execution time improved by high-performance accelerators, especially for latency-sensitive workloads like online inference [19] and real-time data processing [26]. Our experiments in II-C reveal that cold start consists of three components, pre-warm window, CUDA context initialization, library loading latency *e.g.,* cuDNN, PyTorch, Tensorflow. The components can be four times longer than the execution time of CUDA applications and 60× longer than that of the inference latency for deep learning models.

The second challenge is the violation of SLO. Co-location affects the execution of each task, which may lead to exceeding the time limits set by users for their tasks. Thus, it is crucial to ensure that each function meets its time constraints under co-

Zhuolong Jiang, Zinuo Cai, Yumou Liu, Ruhui Ma are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: {jzz19961996, kingczn1314, liiyumou, ruhuima}@sjtu.edu.cn).

Rajkumar Buyya is with the Quantum Cloud Computing and Distributed Systems Laboratory, School of Computing and Information Systems, University of Melbourne, Australia (e-mail: rbuyya@unimelb.edu.au)

*Both authors contributed equally to this work.

†Corresponding author is Ruhui Ma (ruhuima@sjtu.edu.cn).

*https://openwhisk.apache.org/

location. To improve SLO performance in co-location, FaST-GShare [25] divides all functions into two groups based on the likelihood of meeting SLO. It maintains queues with different priorities for each group. However, it overlooks the relationship between memory usage and the execution order of different tasks in co-location under different workloads.

The third challenge is enabling efficient GPU sharing to support the co-location of serverless functions. As the number of concurrent requests increases, more tasks need to be processed simultaneously in the GPU, which can lead to complex resource allocation and performance degradation. Therefore, allocating the optimal GPU resource configuration and reducing resource contention for each function is crucial. NVIDIA provides MPS to allow multiple processes' CUDA kernels to run concurrently on the same GPU, improving GPU concurrency and utilization, but it does not consider optimal resource configuration for co-location. Degradation-aware solution [27] models and predicts the degradation of co-location to configure GPU resources, but it ignores the computing resource competition. Heuristic-based solution [25] identifies the configuration based on sampling function resource quotas, but it ignores the over-subscription of GPU utilization and lacks flexibility in resource configuration.

To overcome these challenges, we propose FAASHARE, an efficient GPU-enabled serverless framework designed to address three goals: low latency, satisfaction of SLO, and high GPU utilization. For low latency, FAASHARE provides pre-warming mechanisms for serverless functions, significantly reducing cold start delays during initial execution. To guarantee SLO, FAASHARE offers dynamic priority adjustments for different workloads in over-subscribed scenarios, ensuring fair execution opportunities for each task to meet co-location SLO requirements. To improve GPU sharing, FAASHARE utilizes a Bayesian Optimization-based (BO-based) GPU sharing strategy to allocate the optimal computing resource configurations for co-location and reduce resource contention.

To be specific, we firstly propose `MSContainer`, a novel container abstraction to achieve low latency for GPU-required serverless functions. Initially, `MSContainer` creates pre-warmer to offer warm-up environments for serverless functions, then injects the functions into the pre-warmer and monitors their entire lifecycle. `MSContainer` backs up the pre-warmer's state before each function execution, enabling the program to recover from abnormal exits. To demonstrate the universality of cold start optimization in `MSContainer`, we further extend its support to other high-level programming languages, like Python and Java.

To ensure SLO guarantee, we propose a temporal-spatial priority queue scheduling mechanism. It predicts task degradation and provides fair execution opportunities for different types of functions in over-subscribed scenarios. It considers both the memory footprint and execution time of tasks, which not only enhances memory utilization but also ensures tasks meet their time constraints. It offers a real-time priority update strategy for task requests, iteratively adjusting the execution order of the task queue to optimize task performance. Moreover, the priority update strategy ensures that the order of execution is not influenced by the order of requests, allowing each task to have a fair chance of execution.

To address the challenge of GPU sharing, we introduce BO and combine it with MPS to propose a serverless-specific BO-based GPU sharing strategy to allocate optimal computing resources for serverless functions to enhance GPU sharing performance. Compared to traditional BO, our approach offers three optimizations. First, it dynamically adjusts the granularity of resource allocation based on the characteristics of tasks in co-location to adapt to varying workloads. Second, it modifies the preferences for exploration and exploitation in BO to expedite the search for the optimal configuration for co-location. Third, it dynamically updates based on existing configurations to generate new resource allocations rapidly to adapt to resource optimization for new tasks swiftly.

To demonstrate the effectiveness of our serverless framework for the abovementioned challenges, we build FAASHARE aims, a low latency, SLO guarantee, and high GPU utilization serverless framework. We conduct extensive evaluation to prove that FAASHARE can substantially outperform state-of-the art systems by 23%, 33%, and 18% on job completion time, makespan, and system throughput, while meeting the low latency goals of serverless services.

Our contributions are as follows:

- We propose FAASHARE, a low latency and efficient SLO-aware GPU sharing serverless framework to enable low latency, SLO guarantee, high GPU utilization for serverless function.
- We design an `MSContainer`, a novel container abstraction to provide a pre-warm environment for serverless functions for low latency challenge to reduce cold start latency.
- We design a temporal-spatio priority queue scheduling to predict task degradation and provide dynamic priority and fair opportunity in oversubscribed scenarios to ensure SLO for serverless function.
- We propose a BO-based GPU sharing approach to allocate the optimal computing resource configuration and mitigate resource competition for co-location to improve GPU utilization.
- We implement FAASHARE based on OpenWhisk and demonstrate its significant improvement in cold start, SLO guarantee, GPU sharing through extensive evaluation.

II. BACKGROUND AND MOTIVATION

A. Serverless Computing

Serverless computing, also known as Function as FaaS, is an emerging paradigm in the field of cloud computing, and it is rapidly growing in popularity. The traditional IaaS model typically revolves around servers, where users are responsible for managing hardware resources to meet specific workload demands. However, serverless computing completely changes this model by outsourcing the responsibility of resource management to cloud service providers. Users only need to program without considering the underlying servers or resource configurations, simplifying code development. Serverless computing also offers high scalability, allowing

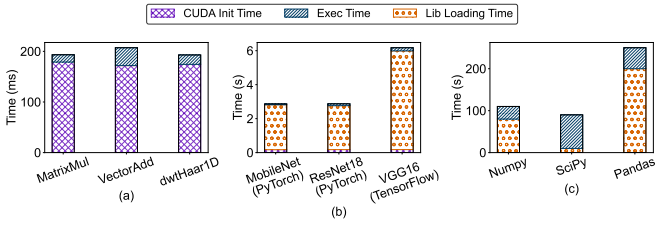


Fig. 1. Runtime analysis of CUDA-based programs.

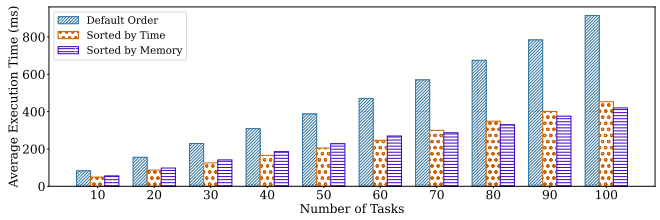


Fig. 2. Performance comparison of different priority strategy.

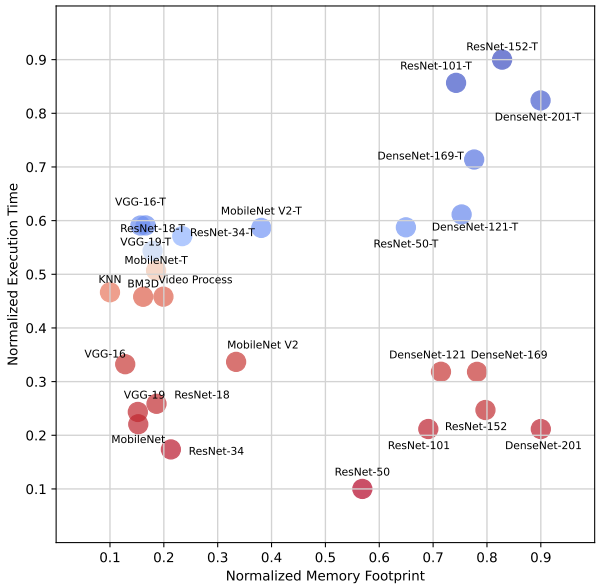


Fig. 3. Distribution of GPU-based workloads.

for dynamic resource allocation based on workload demands, leading to more efficient resource utilization and cost control. These advantages present new prospects for tasks such as deep learning inference [16], [22], [25] and training [24].

To provide high-performance hardware support for computationally intensive tasks, we implement a naive version of GPU-enabled serverless platforms based on Apache OpenWhisk as our baseline, which can support GPU allocation according to the function resource requirements. By modifying the core components of Controller, and Invoker, our baseline can support GPU allocation according to the function resource requirements and memory and CPU provided by typical serverless frameworks. We deploy the GPU-enabled OpenWhisk in a distributed Kubernetes cluster[†], a production-grade container orchestration system and evaluate the baseline with CUDA applications and real-world workloads.

B. Low Latency Challenge

Cold start poses a critical challenge in serverless computing, particularly exacerbating scenarios with GPUs featuring lazy initialization characteristics. Cold start is typically composed of CUDA initialization and library loading, which is intolerable, especially for latency-sensitive workloads like deep learning inference tasks. To demonstrate the effect of cold start on the task, we evaluate cold start and pre-warm performance at baseline. Figure 1 shows the response latency of GPU cold start when handling six typical serverless workloads on the naïve GPU-enabled serverless platforms. We evaluate three pure CUDA applications, Matrix Multiplication (MM), Vector Addition (VA), and DwtHaar1D (DH), and three high-level model serving applications. Our evaluation shows that the cold start latency can be at most 120× the actual execution time, which is highly unacceptable for latency-sensitive workloads. Library loading occurs before CUDA initialization and mainly includes various acceleration libraries for scientific computing, such as PyTorch, TensorFlow, cuDNN, etc. Although these libraries have different latencies, they significantly impact the execution time of the job. Moreover, CUDA initialization extends the delay time for CUDA-based applications in Figure 1 (a).

Challenge I

> *GPU cold start results in longer completion times for serverless tasks using GPU devices and is primarily composed of pre-warm window, CUDA initialization and library loading.*

C. SLO Challenge

SLO guarantee is key to the serverless paradigm to enhance user experience. We first analyze the characteristics of popular GPU-based tasks, and then randomly select different numbers of tasks from Table I to describe the impact of priority execution strategies on performance.

Figure 3 shows the task distribution for different model inference, model training, and native CUDA applications, "-T" represents training task, and reveals four typical performance and resource utilization types. MobileNet and VGG16 exhibit low memory needs and rapid execution. In contrast, DenseNet-201 training shows high memory use but reasonable execution times. KNN tasks use less memory but are computationally demanding. Lastly, some tasks have high memory demands but brief execution times.

To schedule complex tasks in serverless, we perform repeatable random sampling of tasks from Table I and employ default, time-priority, and memory-priority scheduling strategies, with outcomes depicted in Figure 2. Time-priority sorts tasks based on their execution time from shortest to longest, memory-priority sorts tasks from smallest to largest based on memory usage, and default executes tasks in a random order. Both time-priority and memory-priority outperform the default, highlighting their importance in enhancing

[†]https://kubernetes.io/

TABLE I
WORKLOADS USED TO EVALUATE FAASHARE.

Type	Name	Configuration	SLO (ms)
Native CUDA	MatrixMul [a]	51,1024,2048,4096	-
	VectorAdd[a]	10^5,10^6,10^7,10^8	-
	dwtHaar1D[a]	64k,128k,256k,512k	-
	Image encode and decode[a]	1024, 2048,4096, 8192	10,50,280, 600
	PCA [28]	20k, 40k, 80k	-
	KNN [29]	32, 64,128, 256	-
	BM3D [30]	256,512,1024, 2048	40,110,400,1600
	Video Process[b]	720p, 1080p, 4k, 8k	-
DL	ResNet-152 [31]	16, 32, 64, 128	108, 207, 389, 751
	ResNet-101 [31]	32, 64, 128, 256	145, 273, 525, 1020
	ResNet-50 [31]	16, 32, 64, 128	46, 87, 164, 317
	ResNet-34 [31]	64, 128, 256, 512	94, 180, 360, 705
	ResNet-18 [31]	8, 16, 32, 64, 128	9, 15, 29, 54, 102
	DenseNet-121 [32]	16, 32, 64, 128	52, 96, 180, 350
	DenseNet-161 [32]	32, 64, 128, 256	219, 407, 780, 1550
	DenseNet-201 [32]	16, 32, 64, 128	89, 158, 293, 566
	VGG-19 [33]	16, 32, 64, 128	107, 209, 410, 806
	VGG-16 [33]	8, 16, 32, 64, 128	47, 89, 172, 338, 667
	MobileNet [34]	64, 128, 256, 512	60, 117, 232, 459
	Qwen2-7B [35]	1	200
	Llama-3.2-3B [36]	1	160
	DeepSeek-R1-1.5B [37]	1	140

[a] https://developer.nvidia.com/cuda-downloads
[b] https://github.com/NVIDIA/VideoProcessingFramework

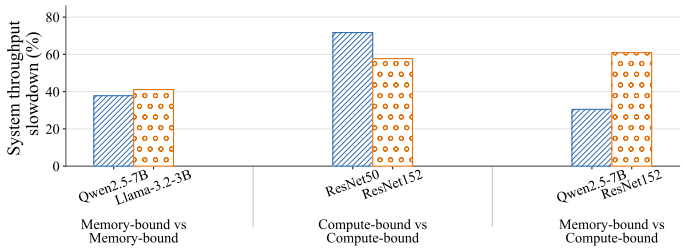


Fig. 4. System throughput slowdown comparison of workloads in MPS mode.

task performance. For instance, time-priority reduces average execution time in scenarios like 60 tasks, while memory-priority excels in others, such as with 100 tasks, showing that time and memory factors should be considered together in scheduling. The results reveal that without optimizing task order, significant performance degradation occurs. While time and memory strategies offer advantages over the default, they do not always ensure optimal execution.

This indicates that task scheduling can lead to significant performance degradation without optimizing the task order. Although the time-priority and memory-priority strategies have significant advantages over the default strategy, it does not provide the optimal execution order.

Challenge II

The co-location of model training, model inference, and native CUDA applications simultaneously impacts the SLO.

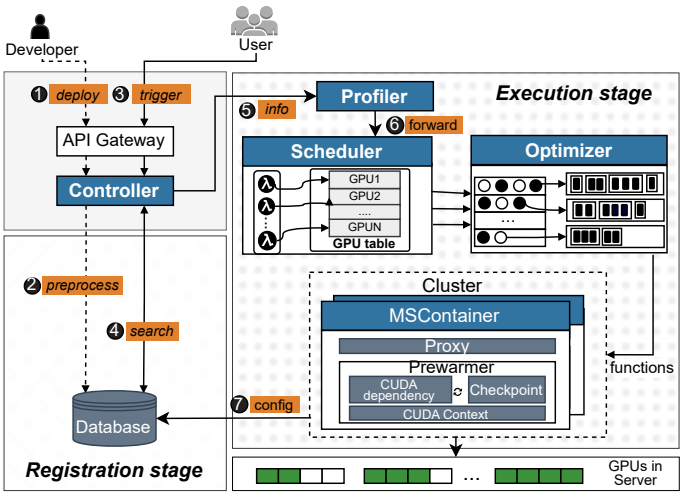


Fig. 5. System overview.

D. GPU Sharing Challenge

We employ the official device plugin in the naïve implementation to support GPU scheduling in large clusters, but its exclusive scheduling pattern does not allow GPU sharing among different serverless functions, which results in low device utilization in the naïve implementation. Fortunately, NVIDIA officially provides MPS to support fine-grained GPU sharing. The MPS mechanism allows parallel execution without context switching, significantly improving GPU efficiency from the software level. However, it cannot allocate suitable configurations for different tasks and alleviate resource competition in co-location.

Furthermore, Figure 4 illustrates the system throughput slowdown comparison across three workload combinations under co-location. In memory-intensive scenarios, contention for memory bandwidth leads to performance degradations of 37% and 41%, respectively. For mixed workloads, ResNet-152 suffers a 60% throughput drop, while Qwen2.5-7B experiences a 30% reduction. This demonstrates that memory-intensive jobs tend to preempt resources and subsequently suppress compute-intensive jobs. Similarly, the compute-intensive combination results in throughput drops of 70% and 55% due to SM resource saturation. These results confirm that diverse workload combinations induce significant performance degradation, highlighting the necessity of optimizing resource configurations in GPU sharing scenarios.

Challenge III

MPS can provide parallel capabilities for serverless function, but the challenges lie in finding the optimal configuration and reducing resource contention while improving GPU utilization for co-execution serverless workloads.

III. DESIGN

We propose FAASHARE, an efficient GPU-enabled serverless framework to achieve low latency, satisfaction of SLO, high GPU utilization. Figure 5 shows an overview

of the system. FAASHARE has four core components: an MSContainer, a controller, a scheduler, and an optimizer. First, we employ MSContainer to provide a pre-warm environment for functions, mitigating cold start latency and enabling function execution on GPUs. Second, the scheduler performs queue-level decisions by determining task ordering and candidate co-location sets according to SLO urgency and memory constraints, so that latency-critical tasks can obtain timely execution opportunities. Third, the optimizer performs resource-level decisions for the tasks selected by the scheduler. Specifically, it refines the MPS quotas of tasks within the selected co-location set, thereby improving GPU utilization while preserving SLO feasibility. In this design, the scheduler decides which tasks run together and in what order, while the optimizer decides how much GPU share each selected task receives. Finally, the optimizer can allocate optimal computing resources for different workloads and task characteristics to achieve the best co-location performance.

Workflow of FAASHARE is similar to the function lifecycle mechanism on existing serverless platforms, and partitions the workflow into two stages: function registration and execution. In the registration stage, developers first deploy their functions that require GPU resources during execution to the serverless platform ❶. The controller accepts the request, performs preprocessing ❷. Then the controller stores the code file in the Database.

In the execution stage, after users trigger function invocation requests to the serverless platform ❸, the controller retrieves the code file and searches for a suitable configuration in the database based on the request information and current GPU load information ❹. If a suitable configuration is found, the controller informs the profiler ❺to forward the requests to the scheduler ❻. Otherwise, the profiler notifies the scheduler to perform the online optimization to generate optimal resource configuration for the task by BO-based GPU sharing strategy and stores the corresponding configuration in the database for next time ❼. Then, the controller forwards the requests to the scheduler. At the same time, the profiler is also responsible for monitoring the status and data in the GPU in real-time, including memory usage, task execution time, and storing the data in the database. Finally, the scheduler arranges fair sharing opportunities and sends the request to the MSContainer for execution to reduce cold start latency.

IV. MSCONTAINER

In this section, we propose MSContainer, a novel container abstraction designed to reduce cold start latency for serverless functions since we observe that cold start is a bottleneck when enabling GPU support for serverless platforms in II-C, especially for latency-sensitive workloads. We conduct the first trial to design an application-agnostic mechanism to enable fast application startup. The core insight of MSContainer is pre-warm beforehand to alleviate shared context initialization latency. The key components of MSContainer are the cold start management, dynamic injection, multi-programming language extension. Cold start management is used to manage the creation and destruction

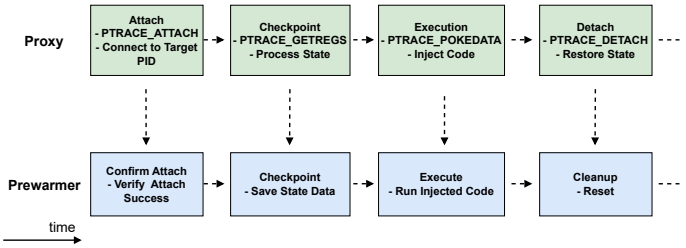


Fig. 6. The process of dynamic injection.

of a pre-warmer that contains a pre-warm context. Dynamic injection allows serverless functions to be attached to the pre-warmer as plugins, monitoring and interacting with the pre-warmer. Multi-programming language extension makes cold start optimization generally support other high-level programming languages.

A. Dynamic Injection

We employ a dynamic injection mechanism to enable serverless functions to access pre-warmer to reduce cold start latency. The core of the dynamic injection mechanism is ptrace. Ptrace is a system call that enables the parent process to observe and control the status and execution of its child processes. We create a proxy to leverage ptrace to inject function into pre-warmer. Figure 6 shows dynamic injection primarily consists of four stages: attach, backup, execution, detach.

Attach. The main purpose of the Attach stage is to obtain the runtime context of the pre-warmer, including memory layout, thread information, variable states, etc. The proxy utilizes ptrace to establish a communication channel with the pre-warmer, enabling interaction in subsequent stages. This includes sending instructions to the target program, retrieving execution results, or receiving updates on the program's status.

Checkpoint. The checkpoint stage involves copying or extracting the memory, register state, files, and other critical data of the target program. Its purpose is to restore the original state of the target program when performing injection operations, if necessary. Because injection operations may modify the pre-warmer's memory layout, register state, and data structures. By creating a copy of the target program during the backup stage using ptrace, it is possible to restore it to its original state in case of problems during the execution phase.

Execution. In the execution stage, the proxy injects a serverless function into the pre-warmer and executes the function. This is achieved by modifying the execution flow of the pre-warmer to redirect it to the entry point of the injected code. Specifically, the proxy first utilizes ptrace to create a memory space within the pre-warmer. Then, using the memory writing capability provided by ptrace copy the function into the previously allocated memory region. Finally, the proxy utilizes ptrace to modify the program counter of the pre-warmer to point to the address where the function is executed.

Detach. In the detach stage, the proxy unloads the serverless function from the pre-warmer and cleans up the associated resources, restoring the pre-warmer to its original state before injection.

B. Multi-Programming Language Extension

To demonstrate the generality of our approach, we further elucidate the cold start optimization principles in Python and Java.

The core of implementing dynamic injection in Python lies in the interoperability between Python and C. First, the code of the serverless function is compiled into a dynamic link library (DLL). Then, a proxy, which is used for Python injection, integrates the DLL into the runtime of the pre-warmer using the `ctypes` or `cffi` modules. Finally, Injection logic is implemented for Python based on C, to perform the attach, checkpoint, execution, and detach operations on the pre-warmer, completing the entire dynamic injection process. This migration of dynamic injection technology from C to Python reduces cold start time. Thus, dynamic injection is enabled to transfer to Python to reduce cold start. For Java, dynamic injection utilizes the Java Virtual Machine's (JVM) capabilities to manipulate and extend the runtime behavior of Java applications. First, We initialize a specialized Java process as a pre-warmer. Then, a Java proxy is created to package the serverless function into a JAR file and perform dynamic injection logic. The proxy utilizes the JVM's API to implement injection logic to execute the injection process into the pre-warmer. This method allows for the dynamic injection of code directly into the Java runtime, significantly reducing cold start and improving execution efficiency.

C. Multi-layer Defense Mechanism

To mitigate the security risks and potential instability inherent in dynamic injection, FAASHARE incorporates a comprehensive multi-layer defense mechanism. This framework spans the entire lifecycle of the injection process, including pre-injection handshakes, code integrity verification, runtime sandboxing, and fault recovery.

Safe-point Handshake. To eliminate risks such as deadlocks or context corruption caused by asynchronous process interruption, we introduce a synchronization mechanism. Specifically, the Pre-warmer sets a `Ready` flag upon completing its initialization. The Proxy is permitted to execute PTRACE_ATTACH only when the Ready flag is asserted, ensuring that injection operations do not interfere with the Pre-warmer's critical sections, such as memory allocation or locking operations.

Integrity Verification and $W \oplus X$ Protection. To safeguard against tampering and overflow attacks, the Proxy performs a SHA-256 checksum verification prior to injection. Furthermore, we adhere to the $W \oplus X$ (Write XOR Execute) principle: memory permissions are transitioned atomically from writable to executable only after the code has passed integrity verification and has been successfully committed to memory.

Post-injection Sandboxing. To prevent user functions from leveraging the injection channel for privilege escalation (Security), we enforce strict system call filtering. The Proxy utilizes ptrace to force-activate a Seccomp-BPF filter, which intercepts and denies dangerous system calls (e.g., ptrace itself). This mechanism effectively constrains the execution environment and restricts any malicious logic within the user code.

TABLE II
DESCRIPTIONS OF THE INPUT FEATURES FOR THE PREDICTOR

Name	Description
Task features	SM utilization, memory bandwidth utilization, and L2 cache utilization when the task runs in isolation.
Contention features	Aggregate SM utilization, memory bandwidth utilization, and L2 cache utilization of the other co-located tasks on the same GPU.
MPS configuration	Assigned MPS percentage limit for the task.

Atomic Rollback Mechanism. To prevent process crashes resulting from injection failures, the Proxy snapshots all register states of the Pre-warmer before any memory modification occurs. If the injection process triggers an exception (e.g., a segmentation fault), the Proxy immediately intercepts the signal and leverages the cached register data to force-reset the CPU registers and instruction pointer. This ensures that the Pre-warmer can be rolled back to its idle state prior to the injection attempt, maintaining system availability.

V. PERFORMANCE PREDICTOR

To accurately estimate performance degradation under co-location and ensure strict SLO satisfaction, we design a performance predictor. By analyzing job resource characteristics and the intensity of resource contention in an NVIDIA MPS-enabled environment, the predictor directly estimates the execution latency under interference.

We choose Random Forest as the predictor because performance degradation under GPU co-location is jointly affected by multiple factors, such as competition for compute resources and contention for memory bandwidth. As a result, the relationship between resource contention and execution latency is typically highly non-linear, making it difficult for simple linear models to characterize accurately. Random Forest can effectively fit such complex mappings through an ensemble of decision trees, and is therefore well suited for our performance prediction task.

The input features are summarized in Table II. We incorporate job features, interference features, and partition configurations as inputs to predict the execution time of a job under co-location. Job features include SM utilization, memory bandwidth, and L2 cache occupancy during standalone execution. Interference features characterize the resource contention for compute and memory bandwidth from co-resident jobs on the same GPU. The MPS configuration represents the resource percentage allocated to the job. By modeling the interplay between job resource utilization and the execution environment, the predictor estimates co-located execution time to proactively prevent SLO violations caused by resource contention. Furthermore, we guarantee SLO compliance by validating the predicted execution time against the job's SLO.

Specifically, we implement the predictor as a Random Forest regressor with 100 regression trees. Each tree is trained on a bootstrap-sampled subset of the original training data, and the maximum depth of each tree is set to 12. During inference, the model averages the outputs of all trees as the final

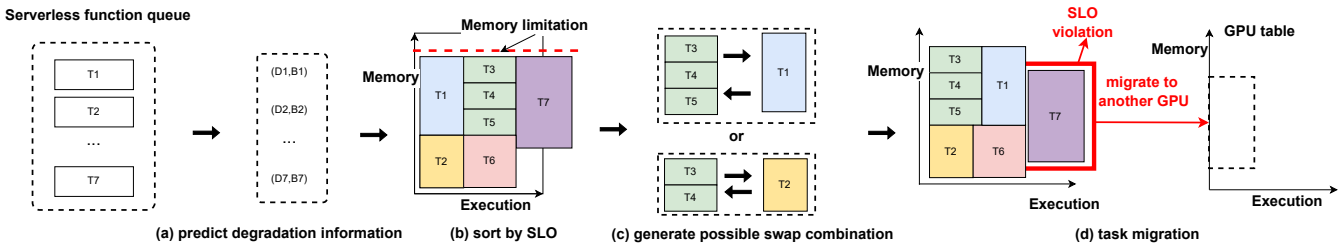


Fig. 7. Spatio-temporal priority queue scheduling.

prediction, thereby reducing variance and improving prediction stability and generalization ability. In our implementation, we empirically measure that a single predictor inference takes approximately 0.5 ms, indicating that the predictor introduces only negligible runtime overhead.

We construct a dataset comprising 1,000 samples using the tasks listed in Table I. These foundational tasks encompass 16 native CUDA workloads and 32 deep learning workloads. Specifically, we generate the samples by randomly co-locating 2 to 10 concurrent tasks and sweeping the MPS proportional limits from 10% to 100% in 10% increments. Finally, we split the dataset into an 80% training set and a 20% testing set. During training, we use Mean Absolute Error (MAE) as the splitting criterion for tree construction to improve robustness against abnormal interference, and we ultimately adopt Mean Absolute Percentage Error (MAPE) as the primary metric to evaluate the model's generalization and prediction accuracy. Based on this metric, our model achieves an overall prediction error of 3.7%. For different GPU types, we use the same feature format and data collection pipeline, but train an independent Random Forest predictor for each GPU type. At runtime, FAASHARE selects the corresponding predictor according to the target GPU type.

VI. Spatio-Temporal Priority Queue Scheduling

We design spatio-temporal priority queue scheduling to provide fair execution opportunities for different types of serverless task queues in over-subscribed scenarios, ensuring SLO guarantees. Specifically, we propose a temporal-spatio priority task scheduler to provide fair execution opportunities for scheduling different types of serverless tasks in over-subscribed scenarios, thereby ensuring SLO guarantees.

The input of the algorithm includes the SLO and memory requirements of all tasks in the task queue, as well as the performance predictor, and the output is the optimal task execution order. The detailed algorithmic flow is shown in the Algorithm 1. We first sort the oversubscribed job queue according to SLO urgency to prioritize latency-critical jobs (Line 1). Here, decoupling means that execution time is removed from the primary sorting criterion, rather than from the entire scheduling process. After the initial order is fixed, execution-time estimates are only used as a feasibility signal for SLO validation. Then, the job queue is partitioned into several sub-queues based on GPU memory capacity (Line 2). To quickly determine whether concurrent tasks within a sub-queue can collectively meet their respective SLOs under MPS co-location, we tentatively assign each task the minimum MPS configuration that just satisfies its own SLO and invoke the performance predictor. If interference still causes any task to violate its SLO, we continuously migrate the task at the tail of the sub-queue to another GPU until the prediction shows that all remaining tasks can safely meet their SLOs (Lines 3-6). We perform space optimization to further maximize the number of concurrently executable tasks within a fixed GPU memory size, thereby reducing the average execution time of the entire queue. Specifically, we sequentially select tasks that satisfy the constraints from different queues for order swapping until we have traversed the entire queue (Lines 7-9). The purpose of swapping task orders is to use the same amount of memory to substitute in more tasks that can be executed in parallel while satisfying the constraints, thereby achieving task priority updates and reducing the average execution time of the task queue. The task order swapping considers SLO guarantees and average task execution as constraints (Lines 13-18). To further address potential SLO violations caused by interference arising from consolidating multiple workloads on an MPS-enabled GPU, we similarly utilize the performance predictor during the execution exchange phase. For each task involved in a potential swap, the predictor evaluates its actual execution time within the current MPS co-location context. If the predicted execution time exceeds the task's SLO requirement, indicating an SLO violation due to resource contention, the scheduler intervenes by migrating the affected task to another available GPU to prevent severe interference (Lines 23-29). If a task swap can improve the average task completion time while meeting SLO requirements, pre-verified by the predictor, we perform the swap to maximize memory utilization and reduce the average completion time (Lines 31-34). In this stage, the scheduler determines which tasks should run first and which tasks can be grouped together, while the detailed resource allocation is handled in a subsequent optimization stage. Figure 7 also illustrates the process of spatio-temporal priority queue scheduling. We first generate possible task swap combinations, and then, with the assistance of the performance predictor, determine the optimal task execution order by comparing the performance under different task combinations. Additionally, we migrate tasks in the queue that violate their SLOs to a new GPU.

VII. Bayesian Optimization for GPU Sharing

Co-location inevitably introduces resource contention and job performance degradation. Effective resource allocation is the key to improving co-location performance. Bayesian

Algorithm 1: Spatio-Temporal Priority Queue Scheduling Algorithm with Performance Predictor

Input: Task queue A with attributes mem, $time$, and SLO, Performance Predictor
Output: Optimized task queue A

```
SortBySLO(A);
SubQueues ← PartitionByMemory(A, M_GPU);
foreach Q in SubQueues do
    Assign minimal MPS to each t ∈ Q satisfying t.SLO;
    while ∃t ∈ Q, PerformancePredictor(t, MPS_Context) > t.SLO do
        Transfer Q.tail() to another gpu;
    end
end
for i ← |SubQueues| − 2 to 0 do
    for j ← i + 1 to |SubQueues| − 1 do
        ExecuteExchange(SubQueues[i], SubQueues[j])
    end
end
Function ExecuteExchange(Q1, Q2):
    G1 ← ∅, G2 ← ∅;
    foreach t1 in Q1 do
        foreach t2 in Q2 do
            if Match(t1, t2) then
                G1 ← G1 ∪ {t1};
                G2 ← G2 ∪ {t2};
            end
        end
    end
    if |mem(G1) − mem(G2)| ≤ ϵ then
        foreach t in G1 ∪ G2 do
            t_exec ← PerformancePredictor(t, MPS_Context);
            if t_exec > t.SLO then
                foreach gpu in GPUs do
                    if gpu ≠ CurrentGPU(t) and mem(t) ≤ M_gpu then
                        Transfer t to gpu;
                        return;
                    end
                end
            end
        end
        SwapGroups(G1, G2);
        if Avg(t_Q1, t_Q2) < before then
            Apply(G1, G2);
        end
    end
return
```

Optimization is an efficient global optimization algorithm that performs a global search by constructing a probabilistic model of the objective function. In our system, BO functions exclusively as a resource-level decision module. Once the upstream scheduler determines the execution order and candidate co-location set, BO takes over to finely tune the MPS quota for each task within this set. It does not alter task scheduling, but rather determines the optimal GPU share each selected task should receive to maximize overall GPU utilization while strictly satisfying SLO constraints

Thus, we propose a GPU sharing strategy that integrates BO with MPS. Specifically, BO takes jobs and their features as input to generate candidate MPS configurations. Then, the system executes jobs under these candidate configurations and selects the optimal MPS configuration based on the observed data. Finally, BO learns the functional mapping between jobs and MPS configurations to output the optimal configuration for any given set of jobs. In the following sections, we first describe the problem model and then detail the BO-based optimization strategy for serverless computing.

A. Problem Model

Specifically, the input to the optimization model is a 7-dimensional feature vector, and the output is the optimal MPS configuration for a job. Consider a queue containing N concurrent jobs. For any job i, the input features consist of three components: job features v_i, contention features c_i (detailed in Table II), and a candidate MPS configuration s_i for job i. We define the actual execution time of a job under configuration s_i as a function f_i of the input features and the configuration, i.e., $f_i(v_i, c_i, s_i) \rightarrow t_i$. The objective of BO is to explore the objective function space to identify the optimal configuration s_i for each job in the queue, thereby minimizing the job execution time. Meanwhile, the execution time of all jobs must satisfy their respective SLOs. The optimization problem for finding the optimal MPS configuration can be formulated as follows:

$$\min_{s_1,\ldots,s_N} \quad \sum_{i=1}^{N} f_i(v_i, c_i, s_i) \tag{1}$$

$$s_i^* = \arg\min_{s_i \in \mathcal{S}} f_i(v_i, c_i, s_i) \tag{2}$$

$$\text{s.t.} \quad f_i(v_i, c_i, s_i) \leq SLO_i, \quad \forall i \in \{1, ..., N\} \tag{3}$$

B. Bayesian Optimization

To adapt BO for the serverless environment, we incorporate several key design enhancements: offline and online optimization, adaptive exploration-exploitation, and a safety guardrail mechanism.

1) Offline and Online Optimization: We decouple the BO process into offline and online phases. In the offline phase, we generate job queues based on the workloads listed in Table I and profile 1000 data points in a real GPU environment, capturing job features, MPS configurations, and the corresponding execution times. We employ a Gaussian Process (GP) with a Matern kernel as the surrogate model to fit high-dimensional non-linear relationships. To evaluate candidate configurations, we utilize Expected Improvement (EI) as the

acquisition function. The surrogate model is pre-trained on offline data, and the inference results are cached in an SM configuration table. For different GPU types, FAASHARE adopts the same BO workflow, but maintains independent profiling data, GP surrogate models, and MPS configuration tables for each GPU type. If a job queue cannot be matched in the table, online optimization is triggered. The system then assigns an initial configuration for execution and continuously fine-tunes the GP model with real-time feedback to derive and update the optimal configuration. During training and online adaptation, all explored configurations and their measured outcomes are continuously recorded in the SM configuration table for future reuse. Based on our empirical measurements, when a query hits the table, the BO overhead is only 0.2 ms. ~~If the queried task configuration is not found in the table, the system invokes online BO optimization. Since our implementation supports optimizing multiple tasks simultaneously, the average optimization overhead for a new task is about 4 ms and the optimizer typically converges within 7 iterations.~~When the query misses the configuration table, FAASHARE selects the closest historical configuration as the decision result and performs asynchronous BO updates for the missed task using idle GPUs. The measured results generated during this asynchronous optimization process are recorded in the SM configuration table, so that future similar requests can reuse them.

2) Optimization of Exploration and Exploitation: We use the Gaussian process (GP) as a surrogate model for Bayesian optimization, which efficiently deals with complex nonlinear relationships and provides precise uncertainty estimates. We use Expected Improvement (EI) as the acquisition function because it provides an intuitive and effective way to balance the needs of exploration and utilization, which allows the optimization process to use existing information to find regions of performance improvement and explore regions with higher uncertainty to discover potentially better solutions.

To make EI adapt to the optimization of SM configuration for complex tasks in serverless, we design an adaptation optimization for BO, which dynamically identifies the extent of exploration and exploitation in the EI converge process. Specifically, we dynamically adjust the ratio of exploration to utilization by iteratively setting exploration parameters to quickly find the optimal configuration to reduce the number of iterations. We can calculate as follows.

$$\mathrm{EI}(x) = \left(\mu(x) - f\left(x^{+}\right) - \xi\right)\Phi(Z) + \sigma(x)\phi(Z) \tag{4}$$

$$Z = \frac{\mu(x) - f\left(x^{+}\right) - \xi}{\sigma(x)} \tag{5}$$

where $\mu(x)$ is the mean of the prediction at the point x, $\sigma(x)$ denotes the predicted standard deviation at the point x, $f(x^{+})$ is currently the best known observation, ξ is the ratio of exploration and exploitation used to balance the exploration and exploitation weights. Besides, ξ directly affects Z and in turn affects the size of EI. EI tends to improve exploration when ξ is large, and when ξ is small, it tends to develop known good regions, which can adjust the balance between exploration and exploitation at different stages of optimization by dynamically adjusting the values of ξ.

To prevent BO from generating invalid MPS configurations, we introduce a safety guardrail mechanism based on the lower bound of SLOs. Upon detecting an invalid configuration, the system intercepts the request and bypasses the surrogate model, directly assigning the minimum MPS resources required to satisfy the job's SLO. This mechanism ensures that SLOs remain inviolate even if the BO model fails in extreme cases.

VIII. IMPLEMENTATION

The prototype is implemented in approximately 3K–5K lines of code, with Python as the primary control language, combined with CUDA/MPS for fine-grained device-side resource control, and lightweight RPC mechanisms for state synchronization and command dispatch between the control plane and node agents.

In the current prototype implementation, an `MSContainer` runs as a privileged container to allow the Proxy to use `ptrace`. We do not configure additional Linux capabilities separately. To limit the trusted boundary, user code in the Prewarmer is executed under a seccomp filter, which blocks unsafe kernel-level operations, such as `ptrace`. An MSContainer can contain multiple Proxy–Prewarmer pairs, which share the same PID namespace inside the MSContainer. The `MSContainer` maintains the mapping between Proxy and Prewarmer PIDs to ensure that each Proxy only attaches to its corresponding Prewarmer. Therefore, FAASHARE does not require hostPID or cross-container PID visibility in the Kubernetes deployment.

To extend the system from a single GPU to multiple GPUs, we mainly make three modifications. First, we extend the original local GPU status table into a cluster-wide resource table that uniformly records the load, memory usage, and MPS quotas of GPUs across all nodes. Second, we extend the existing scheduling flow into a two-level scheduling process: the system first selects the target node and target GPU, and then reuses the existing queue scheduling and BO optimization logic within the selected GPU. Third, we maintain an independent local MPS control instance for each GPU, while node agents continuously report device status and receive configuration commands.

Regarding the applicability to MIG, since MIG can partition a physical GPU into multiple hardware-isolated GPU instances, each with independent compute, memory, and cache resources, FAASHARE can treat each MIG instance as an individual GPU and apply the same BO-based MPS quota optimization method to the tasks within each instance. Therefore, the method of FAASHARE is also applicable to MIG-capable GPUs.

IX. PERFORMANCE EVALUATION

A. Experimental Environment and Setup

a) Testbed: We use two servers equipped with GPUs for our evaluations. Specifically, one server is configured with 8 NVIDIA RTX 3090 GPUs (24 GB memory per GPU), and the

other server is configured with 8 NVIDIA RTX 4090 GPUs (48 GB memory per GPU), for a total of 16 GPUs across both servers.

b) Workloads: We follow the workload settings in MISO [38] and synthesize a job trace inspired by Helios [39]. In our testbed experiments, we instantiate 120 jobs in total, including 60 SLO-constrained (latency-critical) jobs and 60 best-effort jobs, uniformly sampled from Table I. For best-effort jobs, we treat DL workloads as model training tasks, while for native workloads we use those marked with "-" in Table I (i.e., without SLO). Job arrivals are generated by a Poisson process with an average inter-arrival time of $\lambda = 30$ seconds. To evaluate SLO compliance, we follow [40] to set the SLO for each task to 4 times its standalone execution time. Following the methodology in DVABatch [41], we define low, medium, and high loads as 25%, 60%, and 90% of the peak throughput, respectively. These load levels are determined by subjecting each benchmark to a stepping load to identify its maximum processing capacity.

c) Metrics: We evaluate the system performance using three key metrics [38]: Average Job Completion Time (JCT), System Throughput (STP), and Makespan. Average JCT represents the duration from the beginning of a request's execution to its completion, which serves as an indicator of the system's parallel execution optimization. STP measures the number of tasks completed or the amount of data processed per unit of time, reflecting the overall performance efficiency. Finally, Makespan refers to the completion time of the longest-running job, characterizing the total time required to finish the entire workload.

d) Baselines: **TGS** [42] achieves task sharing by dynamically adjusting the kernel priority of tasks. **DGSF** [43] transparently enables serverless functions to use GPUs through general-purpose APIs. **ElasticFlow** [24] is an elastic serverless training platform for distributed deep learning that provides performance guarantees by elastically scaling resources to meet job deadlines. **FaST-GShare** [25] is an efficient FaaS-oriented spatio-temporal GPU sharing architecture that utilizes a dedicated manager to isolate resources and a profiler to ensure SLO compliance for deep learning inference. **FluidFaaS** [44] provides a dynamic pipelined solution for serverless computing that achieves strong isolation and improved throughput through fine-grained GPU sharing. **Priority-time queue scheduling (PT)** schedules serverless tasks based on execution time priority, assigning higher priority to tasks with shorter execution times. **Priority-memory queue scheduling (PM)** schedules serverless tasks based on memory footprint, where tasks with a smaller memory footprint have higher priority.

B. Cold Start Optimization

a) Performance Analysis of Cold Start: We evaluate the performance of dynamic injection under different workloads as shown in Figure 8. For workloads implemented in C language, the composition of cold start time primarily includes three key components: CUDA initialization, execution time, and injection time.

FAASHARE reduces the latency by 10 × compared to the baseline, as FAASHARE can preload and initialize the necessary CUDA environments before program execution to eliminate CUDA initialization time. FAASHARE also achieves a 5% performance improvement over DGSF because FAASHARE does not need to consider synchronization operations of operator execution. For native CUDA applications with typically short execution times (*e.g.,* MatrixMul) and low-load model inference tasks (*e.g.,* ResNet18), FAASHARE significantly reduces the overall cold start latency. Additionally, the injection time for FAASHARE is 12ms.

b) Multi-programming Language Extension: For the programming languages Python and Java, we breakdown the execution times of workloads into compilation time, CUDA initialization, library initialization, execution time, and injection time. For native CUDA applications, FAASHARE can further reduce latency compared to baseline and DGSF, achieving performance improvements of 5x and 10x respectively. Because FAASHARE not only eliminates compilation time (20 ms) and CUDA initialization but also optimizes library latency.

For ML applications, FAASHARE offers even more significant performance enhancements. Resnet18 relies not only on PyTorch but also on cuDNN, which introduces delays of approximately 0.6s and 1.2s respectively. FAASHARE can completely remove the loading overhead of these libraries and improve performance by 10x and 10x, achieving low latency. Notably, the time cost of dynamic injection based on Python and Java ranges from 5-10 milliseconds.

c) Sensitivity Analysis of Checkpoint for Cold Start: When errors occur in the program, causing it to terminate abruptly, as mentioned in IV, the termination of the program also leads to the termination of the pre-warming process, which requires the pre-warming process to be restarted to load the necessary environment in advance. To demonstrate the effectiveness of checkpoint for cold start, we set the same error probability for 10 tasks and evaluate the average execution time of tasks under different error probabilities, with experimental results shown in Figure 9. When the program error probability is 0.5, FAASHARE achieves 5× improvement compared to baseline because only a few programs need to be restarted. As the error probability increases, the improvement of FAASHARE relative to baseline gradually increases. When the error probability is 0.9, FAASHARE reduces the latency by 2.3× compared to baseline. The primary time cost is spent on continuously compiling and loading dependency packages. At the same time, FAASHARE can restore the context through ptrace and re-run the modified files by the user without needing a complete restart.

C. Overall Performance Comparison

a) Overall Performance Comparison: FAASHARE significantly improves GPU co-location performance through MSContainer, spatio-temporal scheduling, and Bayesian Optimization (BO). Figure 10 shows the performance of the different methods in terms of average JCT, makespan, and STP. The overall performance, from lowest to highest, is: ElasticFlow, TGS, FluidFaaS, FaST-GShare, and FAASHARE.

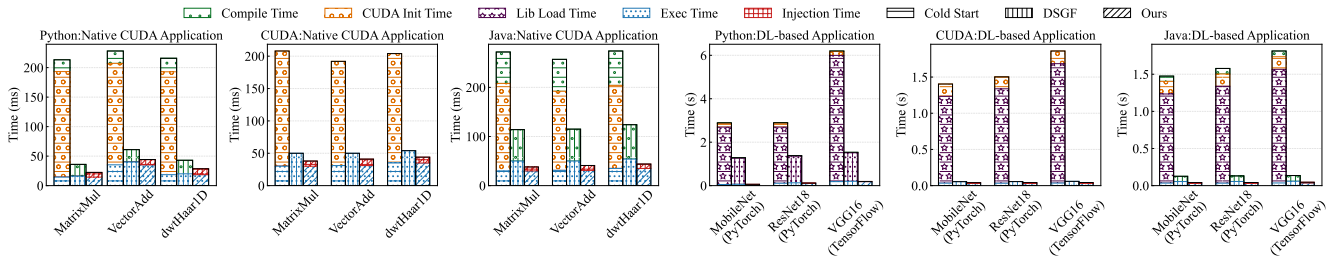


Fig. 8. Breakdown of each step of our workloads when running with unoptimized and dynamic injection.

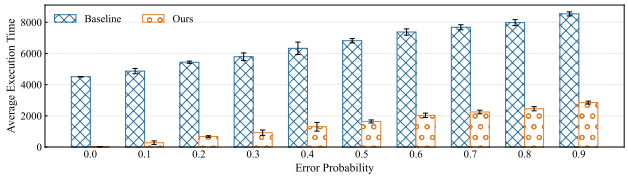


Fig. 9. Error sensitivity analysis for cold start.

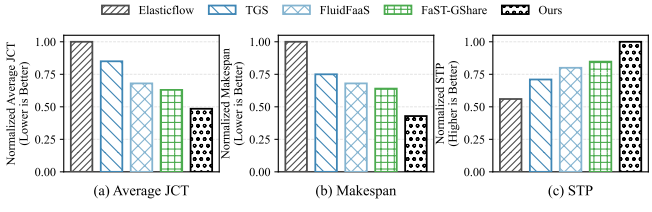


Fig. 10. Performance comparison between different strategies of GPU sharing.

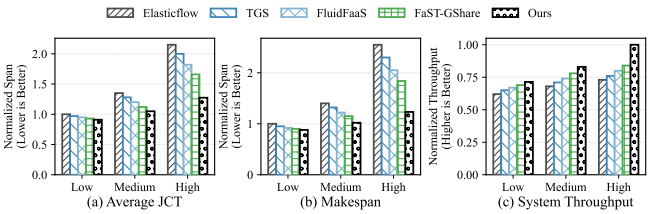


Fig. 11. Multi-scenario performance evaluation.

First, as shown in Figure 10(a), FAASHARE reduces the average JCT by 51%, 43%, 29%, and 23% compared to ElasticFlow, TGS, FluidFaaS, and FaST-GShare, respectively. ElasticFlow and TGS perform the poorest due to their lack of spatial sharing, but TGS slightly outperforms ElasticFlow by leveraging operator rate control. FluidFaaS and FaST-GShare improve performance by introducing spatial isolation, with the MPS-based FaST-GShare outperforming the MIG-only FluidFaaS. In contrast, FAASHARE achieves the best results by minimizing environment initialization latency through MSContainer.

In Figure 10(b), FAASHARE reduces the makespan by 57%, 43%, 37%, and 33% compared to the aforementioned methods, respectively. The concurrency efficiency of spatial sharing methods is significantly better than that of time-sharing methods, and FaST-GShare surpasses FluidFaaS owing to the throughput advantage of MPS. FAASHARE further maximizes the concurrency opportunities for all jobs through spatio-temporal scheduling and utilizes BO to allocate computing resources reasonably and avoid outliers, thereby taking a substantial lead.

In Figure 10(c), FAASHARE achieves the highest STP, improving upon ElasticFlow, TGS, FluidFaaS, and FaST-GShare by 1.79×, 1.41×, 1.25×, and 1.18×, respectively. By configuring the most cost-effective resources for different workloads using BO, FAASHARE maximizes GPU resource utilization and efficiently achieves task parallelism, enabling each job to attain high execution efficiency.

b) Performance Comparison Across Multi-Load : We compare FAASHARE against Elasticflow, TGS, FluidFaaS, and FaST-GShare under low, medium, and high load scenarios. As shown in Figure 11, FAASHARE consistently outperforms the baselines across all scenarios. Under medium load, the system throughput of FAASHARE reaches 1.19× that of Elasticflow. Under high load, FAASHARE reduces the average JCT by 40%, 36%, 30%, and 23% compared to the four methods, respectively. This improvement is attributed to its superior spatial isolation over traditional time-sharing, as well as the reduction in initialization latency by MSContainer. Overall, FAASHARE achieves significant performance improvements across all load scenarios, and notably optimizes execution efficiency, especially in highly competitive environments.

c) Exploration and Exploitation Sensitivity Study: The exploration degree of the acquisition function in BO determines the confidence in finding the optimal value in the target region, primarily reflected in the variable ξ.

Figure 12 presents a sensitivity analysis of different values in the convergence process of BO under different workload scenarios. First, in the low workload scenario, ξ values of 0.01 and 0.05 exhibit better performance and smaller bias compared to ξ values of 0.1 and 0.2. Moreover, smaller ξ values enable BO to converge faster in finding the optimal point. Because when ξ is small, the acquisition function tends to further search for the optimal point in the known better region, reducing the search time. On the other hand, when ξ is large, the acquisition function tends to explore unknown regions, which results in more time wasted on exploration.

In the medium and high workload scenarios, smaller ξ values still demonstrate better performance and smaller bias. Additionally, the convergence process generally exhibits a smaller bias compared to the low workload scenario, and the impact of different ξ values on the convergence process gradually diminishes. This is attributed to the gradual reduction in the number of tasks executed in parallel in the medium and

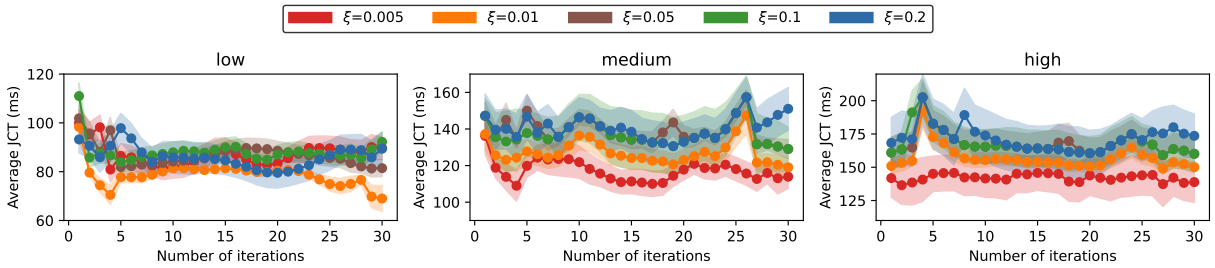


Fig. 12. Exploration and exploitation evaluation.

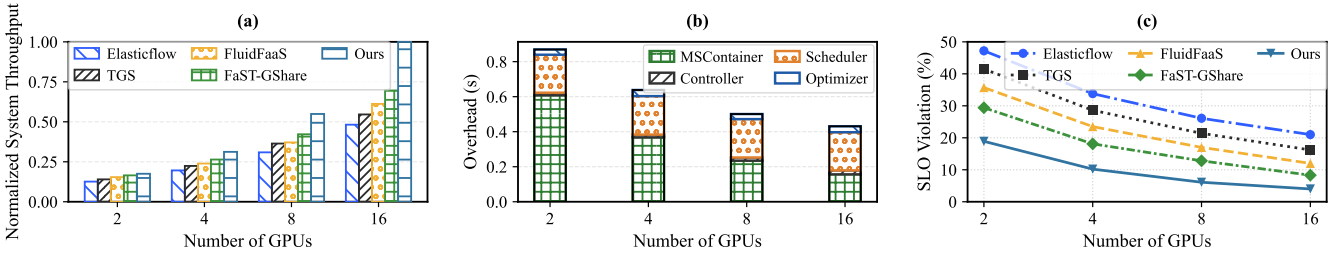


Fig. 13. Performance comparison and overall component-wise overhead in multi-GPU environments. (a) normalized throughput comparison; (b) breakdown of component overheads; (c) comparison of SLO violation rates.

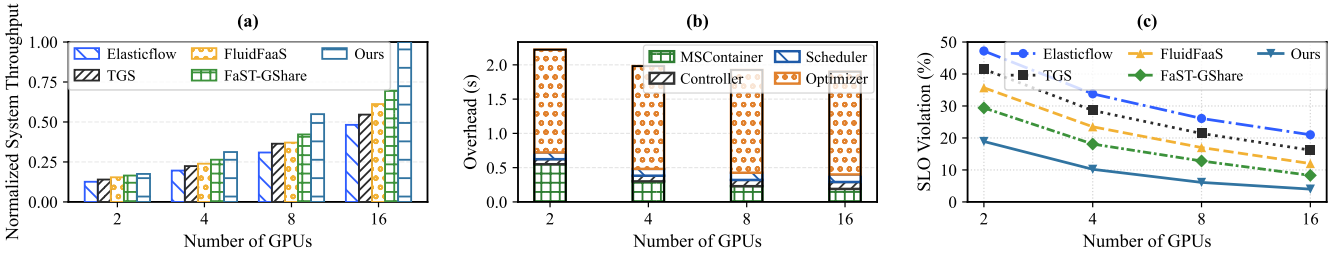


Fig. 14. ~~Performance comparison in multi-GPU environments. (a) normalized throughput comparison; (b) breakdown of component overheads; (c) comparison of SLO violation rates.~~

high workload scenarios, and the corresponding decrease in the need for BO to optimize SM configurations, resulting in faster convergence.

D. Performance Scaling with Different Numbers of GPUs

To evaluate the performance of our approach in multi-GPU environments, we sequentially report four sets of results: multi-GPU throughput, system overhead breakdown, SLO violation rate, and P99 tail latency. Throughput reflects the scaling trend, overhead illustrates the cost of the control plane and optimization modules, and SLO alongside P99 tail latency characterize the changes in SLO under co-located and scaling conditions.

To evaluate the throughput performance in multi-GPU environments, we measure the normalized system throughput across different cluster sizes. As shown in Figure 14(a), the throughput of all methods increases with the number of GPUs. Our method exhibits the best scalability, improving throughput by 52%, 45%, 36%, and 30% compared to ElasticFlow, TGS, FluidFaaS, and FaST-GShare respectively under the 16-GPU scale. ElasticFlow suffers from coarse-grained allocation, TGS lacks spatial sharing, FluidFaaS misses multi-GPU coordination and has limited parallelism, and FaST-GShare experiences a certain degree of computing resource idleness. In contrast, our framework dynamically searches for the optimal MPS configuration via BO and leverages a spatiotemporal priority mechanism to achieve highly efficient parallelism for heterogeneous workloads, thereby substantially increasing the overall throughput.

Figure 13(b) illustrates the execution overheads of the four core components of our system across different cluster sizes, including MSContainer, Controller, Scheduler, and Optimizer. We report the total overhead of each component throughout the entire experimental workflow, rather than the per-task component overhead. As the number of GPUs increases from 2 to 16, the overhead of MSContainer, which is responsible for function environment warm-up and dynamic injection, steadily decreases from ~~0.55s to 0.19s~~0.608s to 0.157s. This is primarily attributed to the accelerated parallel initialization brought by the larger cluster scale. Meanwhile, the Controller, which is responsible for global state maintenance, introduces only ~~extremely low overheads across all scales, fluctuating only between 0.074s and 0.101s~~0.015s to 0.023s of overhead across different GPU scales. The Scheduler, which performs spatio-temporal priority scheduling and predictor-based feasibility checks, remains almost stable, with an overhead of about ~~0.098s to 0.110s~~0.217s to 0.220s. ~~Furthermore, the Bayesian~~

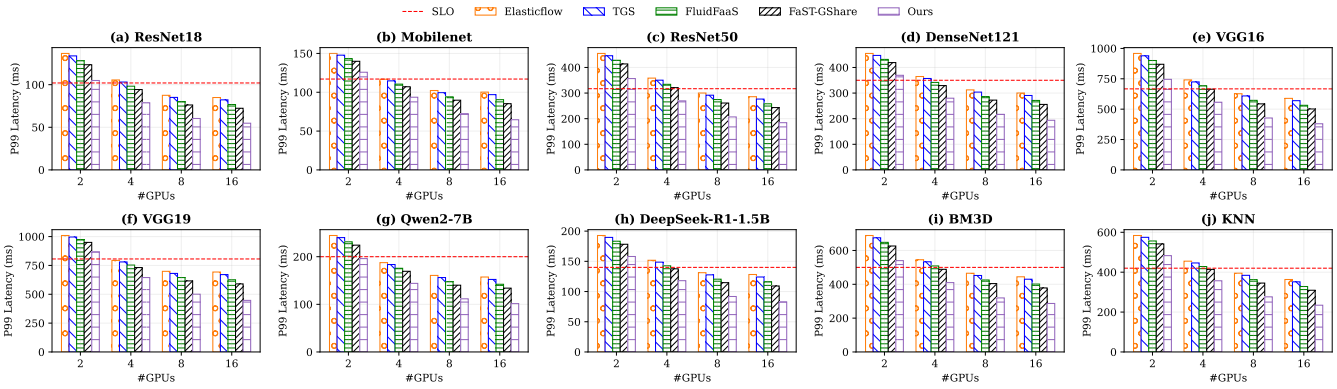


Fig. 15. Performance comparison of P99 latency.

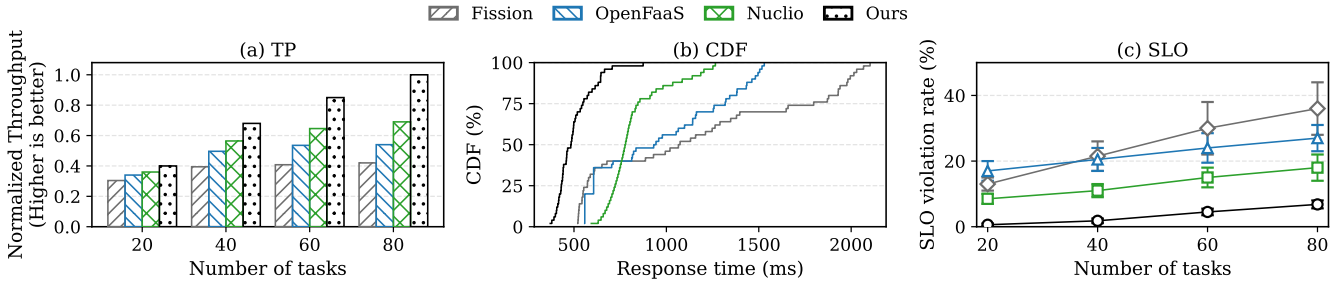


Fig. 16. Performance comparison of various open-source serverless frameworks. (a) Normalized throughput across different numbers of tasks; (b) Cumulative Distribution Function (CDF) of response latency; (c) Comparison of SLO violation rates under different numbers of tasks.

~~optimizer maintains a constant overhead of 1.50s across all test scenarios.~~ For BO, since the overhead mainly comes from the SM configuration table lookup, and the configuration query for each task is independent of scenario changes, its overhead remains within a relatively stable range of approximately 0.02s to 0.03s across different scenarios. Overall, each component introduces only limited control-plane overhead, demonstrating that FaaShare has good scalability in multi-GPU scenarios.

To evaluate the SLO guarantee capabilities in multi-GPU environments, we test the SLO violation rate across different cluster sizes. As shown in Figure 14(c), as computing capacity expands, the SLO violation rates of all methods exhibit a downward trend. Although all methods face compute bottlenecks in the 2-GPU scenario, our method rapidly converges the violation rate under 4-to-16 GPU configurations, further suppressing it from 10.2% to an extremely low level of 4.0%.

Furthermore, we evaluate the P99 tail latency across 10 different workloads in Figure 15. Overall, ElasticFlow and TGS are more prone to queueing tails due to their limited resource allocation and sharing capabilities. FluidFaaS and FaST-GShare lack dynamic coordination and adaptive adjustment during multi-GPU scaling, resulting in more pronounced tail jitter in scenarios involving co-location contention and fragmentation. Our approach combines spatiotemporal priority scheduling with Bayesian-optimized SM quota selection to mitigate co-location interference and stabilize tail latency. The performance gap between the methods widens as the cluster size expands. Under the 16-GPU configuration, our method achieves lower P99 tail latency on most workloads, and this advantage becomes more prominent with increased cluster scales.

E. Comparison with Open-Source FaaS Frameworks

We evaluate the performance of the platforms in scaling tests ranging from 20 to 80 concurrent tasks. As shown in Figure 17(a) and Figure 17(b), when the number of concurrent tasks reaches 80, the throughput of FaaShare leads Fission, OpenFaaS, and Nuclio by 58%, 46%, and 31%, respectively, while its SLO violation rate remains at 6.8%. In contrast, the SLO violation rates of Fission and OpenFaaS increase to 36.0% and 27.0%, respectively, and to 18.0% for Nuclio. The cumulative distribution function (CDF) of response time in Figure 17(c) further corroborates this trend; the 99th percentile latency of FaaShare is reduced by 62%, 48%, and 37% compared to the three aforementioned frameworks, respectively. The performance of traditional frameworks is primarily constrained by their general-purpose architectural designs. Fission's frequent routing allocation and OpenFaaS's dual-layer gateway queuing mechanism introduce additional network hops and lead to long-tail latency. Although Nuclio eliminates external communication overhead by leveraging an in-pod multi-worker concurrent model, it lacks GPU-specific optimization. To address this, FaaShare uses MSContainer dynamic injection to eliminate the cold start bottleneck, incorporates spatio-temporal priority queue scheduling to guarantee the SLO of co-located tasks, and employs Bayesian optimization combined with MPS to allocate SM resources, which mitigates resource contention, compresses queuing latency, and improves overall system throughput and GPU utilization.

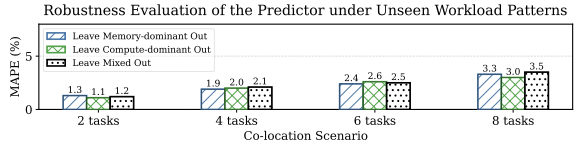


Fig. 17. Performance comparison of different open-source serverless framework.

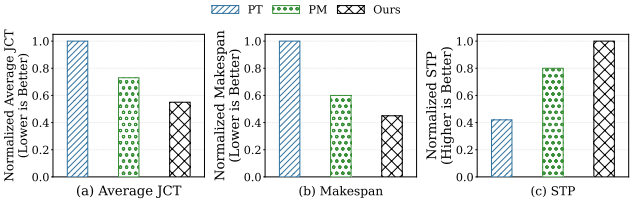


Fig. 18. Performance comparision of priority queue.

F. Robustness Evaluation of the Performance Predictor

To evaluate the robustness of the predictor under various workload combinations, we conduct supplemental generalization experiments. We first construct co-located queues for four scenarios with 2, 4, 6, and 8 jobs, respectively. Based on the ratio of memory-intensive tasks (R_{mem}), we categorize these workloads into three patterns: Memory-dominant ($R_{mem} > 50\%$), Compute-dominant ($R_{mem} < 50\%$), and Mixed ($R_{mem} = 50\%$). We sample 300 instances for each scenario from the dataset in §V and adopt a leave-one-pattern-out validation strategy, where one pattern is reserved as the test set and the remaining two are used for training. Evaluation results show that across the four scenarios, the average MAPE of the predictor for the three unseen combinations is 1.2%, 2%, and 3.2%, respectively. Although the error exhibits an upward trend as the number of jobs increases and combinations become more complex, the predictor maintains low error performance and demonstrates strong generalization capability.

G. Load-aware Priority Queue Optimization

a) Tail Latency Analysis of Priority Queue : Time-priority (TP) and memory-priority (MP) can affect the execution order of the task queue. To investigate the impact of priority strategies on the task queue, we conduct performance analysis experiments, as shown in Figure 18. In Figure 18 (a), compared to TP and MP, we achieve 41% and 23% improvement in average JCT. This is due to the dynamic task priority adjustment strategy, which allows different tasks to have a fairer chance of execution. TP focuses too much on execution time and overlooks the memory space occupancy, resulting in a limited number of tasks executed in parallel. MP fails to provide fair opportunities for execution time. In Figure 18 (b), we continue to provide 51% and 19% improvement in Makespan. TP leads to delays in tasks with long execution time but low memory occupancy, while MP leads to delays in tasks with short execution time but high memory occupancy. Ultimately, in Figure 18 (c), we still achieve 2.5 × higher and 20% improvement in STP compared to TP and MP.

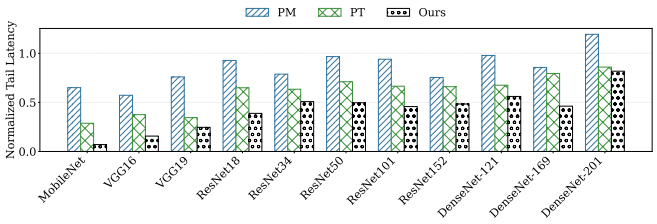


Fig. 19. Normalized P99 tail latency.

b) Tail Latency Analysis of Priority Queue: Figure 19 shows the results of different strategies on p99 latency compared to baseline. First, benefit from the dynamic adjustment of the priority queue, we can significantly reduce the tail latency of application requests. PT reduces the p99 tail latency by 5× compared to PM and PT. As the application workload increases, the gains obtained gradually diminish. This is because PT focuses only on the execution time of tasks while ignoring the impact of memory occupancy, resulting in high tail latency for longer-running tasks. Similarly, PM only considers memory occupancy and prioritizes tasks with lower memory usage, but it overlooks the factor of execution time. By dynamically adjusting the execution time and memory occupancy, we achieve a trade-off between execution time and memory occupancy.

X. Related Work

GPU-enabled Serverless Computing. Molecule [45] is the serverless computing system utilizing heterogeneous computers and enabling both general-purpose devices and domain-specific accelerators for serverless applications. Kim *et al.* develop an API that connects the open source framework to the NVIDIA-Docker and commands that enable GPU programming, then proposes a GPU-supported serverless computing framework that can deploy services faster [46]. Satzke *et al.* propose to extend the open-sourced KNIX high-performance serverless framework to enable executing functions on shared GPU cluster resources [47]. DGSF [43] transparently enables serverless functions to use GPUs through general-purpose APIs. Kernel-as-a-service (KaaS) [48] implements a GPU interface for stateless functions to access GPU instances.

Cold Start Optimization. Shahrad *et al.* [49] manage a pre-warming window and a keep-alive window for each stateless function to alleviate performance degradation caused by cold starts. INFless [16] proposes long-short term histogram, which simultaneously tracks and calculates the histograms of idle time for long-term and short-term applications to find the appropriate warm-up window and duration, but it reduces the cold start of functions at the container level.

Agarwal *et al.* [50] propose a reinforcement learning agent setting to analyze the identified factors such as function CPU utilization, ascertain the function-invocation patterns, and reduce the function cold start frequency by preparing the function instances in advance. FaasCache [51] realizes a caching-inspired greedy-dual keep-alive policy to reduce the cold-start overhead.

GPU Sharing. Prior works have tried to support GPU sharing at different levels. Gandiva [52] proposes GPU time-

sharing in shared GPU clusters through checkpointing at low GPU memory usage of the training job. Harmony [53] applies an RL model to make placement decisions to minimize interference and maximize the throughput for bin-packing DL workloads in a GPU data center. Salus [54] focuses on fine-grained GPU sharing with two primitives: fast job switching and memory sharing. Ali-MLaaS [55] comprehensively analyzes large-scale workload traces in Alibaba and discloses the benefit of GPU sharing in production GPU data centers. FaST-GShare [25] considers task and resource allocation under non-computational resource contention, which is orthogonal to our works.

XI. Conclusion

In this paper, we present FaaShare, a low-latency, satisfaction of SLO, high GPU utilization serverless framework. For low latency, we desgin MSContainer to provide pre-warm context to reduce cold start for serverless function. For the satisfaction of SLO, we propose a temporal-spatio priority queue scheduling mechanism. For GPU sharing, we propose a BO-based resource configuration strategy, which significantly improves the performance of the system. The proposed framework proves its effectiveness in extensive experiments. In terms of cold start, we significantly reduce the delay compared to the unoptimized framework. In terms of GPU sharing, compared with the unoptimized scheme, we improve the average JCT, makespan, and STP by 23%, 33%, and 18%, respectively.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[2] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys*, vol. 54, pp. 1 – 34, 2021.

[3] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, pp. 1 – 32, 2019.

[4] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

[5] L. Jin, Z. Cai, H. Wang, Z. Zhang, R. Ma, H. Guan, and Y. Liu, "Ephemera: accelerating i/o-intensive serverless workloads with a harvested in-memory file system," *ACM Transactions on Architecture and Code Optimization*, vol. 22, no. 3, pp. 1–24, 2025.

[6] J. Liu, Z. Cai, Y. Liu, H. Li, Z. Zhang, R. Ma, and R. Buyya, "Smore: Enhancing gpu utilization in deep learning clusters by serverless-based co-location scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[7] Z. Cai, Z. Chen, R. Ma, and H. Guan, "Smss: Stateful model serving in metaverse with serverless computing and gpu sharing," *IEEE Journal on Selected Areas in Communications*, vol. 42, no. 3, pp. 799–811, 2023.

[8] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures," *Computer*, vol. 45, pp. 65–72, 2012.

[9] E. V. Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, pp. 8–17, 2018.

[10] X. Han, Z. Cai, Y. Zhang, C. Fan, J. Liu, R. Ma, and R. Buyya, "Hermes: Memory-efficient pipeline inference for large models on edge devices," in *Proceedings of International Conference on Computer Design*. IEEE, 2024.

[11] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proceedings of the IEEE International Conference on Computer Communications*, 2019.

[12] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *Proceedings of the International Conference on Management of Data*, 2021.

[13] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[14] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λdnn: Achieving predictable distributed dnn training with serverless architectures," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.

[15] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, "Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementations*, 2021.

[16] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "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*, 2022.

[17] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *Proceedings of the USENIX Annual Technical Conference*, 2022.

[18] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *Proceedings of the International Conference on Parallel Processing*, 2021.

[19] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[20] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[21] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers 4*. Springer, 2018, pp. 154–168.

[22] H. Fingler, Z. Zhu, E. Yoon, Z. Jia, E. Witchel, and C. J. Rossbach, "Dgsf: Disaggregated gpus for serverless functions," *2022 IEEE International Parallel and Distributed Processing Symposium*, pp. 739–750, 2022.

[23] M. Yu, A. Wang, D. dong Chen, H. Yu, X. Luo, Z. Li, W. Wang, R. Chen, D. Nie, and H. Yang, "Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping," *ArXiv*, vol. abs/2306.03622, 2023.

[24] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu, "Elasticflow: An elastic serverless training platform for distributed deep learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[25] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023.

[26] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[27] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on multi-gpu servers with spatio-temporal sharing," in *2022 USENIX Annual Technical Conference*, 2022.

[28] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[29] S. Zhang, X. Li, M. Zong, X. Zhu, and R. Wang, "Efficient knn classification with different numbers of nearest neighbors," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 5, pp. 1774–1785, 2017.

[30] K. Dabov, A. Foi, V. Katkovnik, and K. O. Egiazarian, "Image denoising by sparse 3-d transform-domain collaborative filtering," *IEEE Transactions on Image Processing*, vol. 16, pp. 2080–2095, 2007.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[32] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[34] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[35] Q. Team *et al.*, "Qwen2 technical report," *arXiv preprint arXiv:2407.10671*, 2024.

[36] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv e-prints*, pp. arXiv–2407, 2024.

[37] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[38] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters," in *Proceedings of Symposium on Cloud Computing*, 2022.

[39] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[40] J. Wang, Y. Wang, M. Han, and R. Chen, "Colocating {ML} inference and training with fast {GPU} memory handover," in *in Proceedings of the USENIX Annual Technical Conference*, 2025.

[41] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "{DVABatch}: Diversity-aware {Multi-Entry}{Multi-Exit} batching for efficient processing of {DNN} services on {GPUs}," in *in Proceedings of USENIX Annual Technical Conference*, 2022.

[42] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin, "Transparent gpu sharing in container clouds for deep learning workloads," in *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023, pp. 69–85.

[43] H. Fingler, Z. Zhu, E. Yoon, Z. Jia, E. Witchel, and C. J. Rossbach, "Dgsf: Disaggregated gpus for serverless functions," in *in Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2022.

[44] X. Hui, Y. Xu, and X. Shen, "Fluidfaas: A dynamic pipelined solution for serverless computing with strong isolation-based gpu sharing," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, 2025, pp. 1–14.

[45] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[46] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "Gpu enabled serverless computing framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2018.

[47] K. Satzke, I. E. Akkus, R. Chen, I. Rimac, M. Stein, A. Beck, P. Aditya, M. Vanga, and V. Hilt, "Efficient gpu sharing for serverless workflows," in *Proceedings of the Workshop on High Performance Serverless Computing*, 2020.

[48] N. Pemberton, A. Zabreyko, Z. Ding, R. Katz, and J. Gonzalez, "Kernel-as-a-service: A serverless interface to gpus," *arXiv preprint arXiv:2212.08146*, 2022.

[49] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the USENIX Annual Technical Conference*, 2020.

[50] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2021.

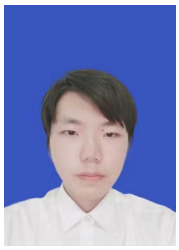
[51] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[52] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementations*, 2018.

[53] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *Proceedings of the IEEE International Conference on Computer Communications*, 2019.

[54] P. Yu and M. Chowdhury, "Fine-grained gpu sharing primitives for deep learning applications," *Proceedings of Machine Learning and Systems*, 2020.

[55] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. 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*, 2022.



Zhuolong Jiang is currently a doctoral student in Information Security at Shanghai Jiao Tong University, China. His research interests are focused on serverless GPU optimization.

Zinuo Cai is currently a graduate student in Computer Science at Shanghai Jiao Tong University, China. He obtained the bachelor's degree in Software Engineering at Shanghai Jiao Tong University. His research interests are focused on resource schedule and system security in cloud computing.

Yumou Liu is currently pursuing an undergraduate degree in Computer Science at Shanghai Jiao Tong University, Shanghai, China. His research interests are focused on cloud computing.

Ruhui Ma is currently an associate professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. He received his Ph.D. degree in computer science from Shanghai Jiao Tong University. His research interests include cloud computing systems, AI systems, and machine learning.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry distinguished professor and director of the Quantum Cloud Computing and Distributed Systems Laboratory, School of Computing and Information Systems, University of Melbourne, Australia. He has authored more than 625 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide.