FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping

Minchen Yu[†‡]   Ao Wang[§]   Dong Chen[†]   Haoxuan Yu[†]   Xiaonan Luo[†]   Zhuohao Li[†]

Wei Wang[†]   Ruichuan Chen[*]   Dapeng Nie[§]   Haoran Yang[§]

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

[§]Alibaba Group   [*]Nokia Bell Labs

Abstract

Serverless computing has become increasingly popular for machine learning inference. However, current serverless platforms lack efficient support for GPUs, limiting their ability to deliver low-latency inference. In this paper, we propose FaaSwap, a GPU-efficient serverless inference platform. FaaSwap employs a holistic approach to system and algorithm design. It maintains models in main memory and dynamically swaps them onto GPUs upon request arrivals (i.e., late binding), thereby enabling a large number of inference functions to efficiently share a node's GPUs. FaaSwap uses various techniques, including asynchronous API redirection, GPU runtime sharing, pipelined model execution, and efficient GPU memory management, to achieve the optimal performance. We also develop an interference-aware request scheduling algorithm that allows FaaSwap to meet the latency SLOs for individual inference functions. We have implemented FaaSwap as a prototype on a leading commercial serverless platform. Experimental evaluations demonstrate that, with model swapping, FaaSwap can concurrently serve hundreds of functions on a single worker node with 4 V100 GPUs, while achieving inference performance comparable to native execution (where each function runs on a dedicated GPU). When deployed on a 6-node production testbed, FaaSwap meets the latency SLOs for over 1k functions, the maximum that the testbed can handle concurrently.

1 Introduction

The remarkable advances in machine learning (ML) and its widespread adoption in various domains have fueled a surging demand for cloud-based ML inference services [20, 25, 40, 51, 52]. However, the prevailing "serverful" model used by existing cloud-based inference services has led to numerous challenges. For instance, users are required to rent GPU virtual machines (VMs) and manually configure various system parameters. Additionally, they must specify the number of VMs needed and dynamically scale them as the inference load changes. This serverful model not only imposes considerable configuration and management burdens, but also leads to resource overprovisioning and GPU underutilization.

Severless computing offers a compelling cloud model for inference serving [15, 45, 49]. In a serverless cloud, users can publish ML models as inference functions, and delegate resource provisioning and scaling responsibilities to the cloud. Serverless computing is also economically appealing as users only pay for the resources actually consumed by their functions (i.e., pay-per-use billing), eliminating the resource idling cost.

However, today's serverless computing platforms, such as AWS Lambda [6] and Alibaba Function Compute [1], lack efficient support for GPUs. They simply run an ML inference model as a container and bind it to a specific GPU when launched, which follow "serverful" model serving practices, such as Nexus [40], INFaaS [38], and MArk [51]. This early-binding approach results in both performance deterioration and cost inefficiency in serverless inference: GPU functions exhibit high startup overhead (e.g., around 10 seconds as shown in Table 1), which is unacceptable for inference services with millisecond-scale service-level objectives (SLOs) [40, 51, 53]. To mitigate this issue, a function must be kept long-lived on its host GPU [4, 7] which requires users to pay for occupied GPUs even during idling, thus incurring high cost inefficiency. It also results in low GPU utilization and load imbalancing, making it inefficient for cloud providers.

We believe that an efficient serverless inference platform should provide four desirable properties. First, it should enable *pay-per-GPU-use billing* for users, with charges incurred only when the functions are invoked and running on GPUs. Second, the platform should achieve optimal GPU utilization by means of efficient *GPU sharing* for concurrent inference functions, minimizing resource provisioning costs for cloud providers. Third, the platform should be aware of the user-specified latency SLOs and strive to meet them for all inference requests, if possible. Lastly, the platform should achieve the aforementioned three properties without detailed knowledge about inference models due to business-critical confidentiality reasons. We notice that there have been a number of relevant systems [19, 25, 27, 28, 37, 38, 40, 45, 50], none of which however provide all these properties. They often suf-

fer from cost inefficiency, SLO violations, or require model-specific knowledge (more discussion in §2.2 and §8).

In this paper, we exploit *late binding* to achieve the desired properties of a serverless inference platform. We argue that such a platform should keep inference models in host memory and dynamically bind them to a pool of GPUs through *model swapping* upon request arrivals. In contrast to maintaining long-lived models on specific GPUs (i.e., early binding), this approach incurs no GPU memory footprint when idling, thereby enabling pay-per-GPU-use billing. By keeping models in the host and late binding, each GPU can accommodate more functions beyond the capacity of its memory and can also balance load across GPUs, leading to efficient GPU sharing and improved resource utilization. Moreover, we assume no detailed knowledge about inference models (e.g., model structure), and carefully explore the design space of model swapping between host and GPU via PCIe, as well as between GPUs via NVLink connections whenever beneficial, yielding significantly lower latency compared with function cold starts and facilitating SLO compliance.

We present FaaSwap, a GPU-enabled serverless inference platform that realizes the above design rationale. Specifically, FaaSwap employs a GPU pooling architecture wherein each worker node manages a pool of local GPUs and allows its inference functions to access GPU resources through CUDA API redirection. This enables seamless model swapping within a GPU pool, being transparent to users. FaaSwap proposes three key designs that exploit the characteristics of inference tasks to systematically optimize the model swapping and execution with GPU pooling. First, it proposes asynchronous API redirection to avoid frequent synchronizations between the inference functions and the GPU pool, thus effectively eliminating high communication overheads for model inference. Second, FaaSwap leverages pipeline execution to overlap the host-to-GPU model swapping and the inference execution, thereby concealing model swapping costs and reducing end-to-end latency. It also leverages high-speed NVLink connections between GPUs for fast model swapping when feasible and beneficial. Combined with low-latency API redirection, FaaSwap can efficiently execute models on any available GPUs. We intentionally do not leverage the detailed information (e.g., structure) about the models to meet the typical business requirement for model confidentiality. Third, FaaSwap designs an efficient GPU memory management system to facilitate model swapping and inference execution. It automatically tracks the addresses of models as they are swapped even across multiple GPUs, and adjusts each memory access of CUDA APIs accordingly during inference execution. It also efficiently organizes and shares memory blocks to avoid high memory allocation overheads, improving overall performance of model swapping. Moreover, FaaSwap ensures resource and fault isolation within its GPU pool.

To meet latency SLOs for inference functions while maintaining low GPU costs, we further propose three policies.
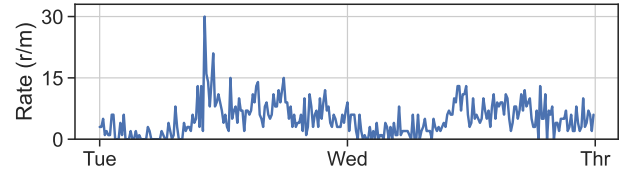

Figure 1: A two-day request trace of a typical GPU inference function in Aliyun Function Compute.

First, FaaSwap designs a request scheduling algorithm that minimizes model swapping overheads, resulting in reduced end-to-end inference latency. It categorizes models into two groups, heavy or light, based on whether these models incur high overhead during swapping via PCIe. Then, in actual request scheduling, FaaSwap prioritizes NVLink over PCIe for transmitting heavy models across GPUs, thus effectively reducing interference caused by concurrent model swapping. Second, FaaSwap leverages model heaviness to guide eviction when GPU memory is insufficient. It tends to cache heavy models in GPUs and evict light ones. Together with request scheduling, this approach significantly minimizes model swapping overhead. Third, FaaSwap proposes an SLO-aware request queuing policy which prioritizes requests to functions that have a higher likelihood of meeting SLOs, thereby effectively increasing the total number of SLO-compliant inference functions.

We have implemented and evaluated FaaSwap atop Aliyun Function Compute, one of the world's largest commercial serverless platforms. Evaluation results show that FaaSwap achieves low-latency model inference, comparable with native executions. FaaSwap can share a single GPU across hundreds of inference functions and load-balance GPUs with model swapping, resulting in over 10× cost reduction compared with current GPU offering in Aliyun Function Compute. With its efficient SLO-aware scheduling and queuing policies, FaaSwap can serve 480 functions on a 4-GPU worker node while achieving low tail latency and satisfying millisecond-scale SLOs for all functions. Cluster experiments further show that FaaSwap effectively scales with the number of inference functions at low resource cost, and meets per-function latency SLOs for thousands of functions.

2 Background and Motivation

In this section, we first give an overview of serverless inference. We then describe the inefficiency of existing solutions to enabling GPUs in serverless platforms, and highlight four key requirements in this regard.

2.1 Serverless Inference

As a prominent serverless platform with a global presence, our Aliyun Function Compute has observed a growing adoption among enterprise customers who opt to deploy their inference

services using serverless functions, known as *serverless inference*. In comparison to existing inference services based on a "serverful" cloud model, such as AWS SageMaker [5], serverless inference significantly alleviates the burden of server management for cloud users. Specifically, the serverful approach requires users to manually configure various system-level parameters (e.g., VM types, GPUs, CPU cores, etc.) and manage resource provisioning (e.g., scaling the number of VMs up or down according to demand changes). In contrast, serverless inference enables users to simply publish models with inference code as functions, and then cloud providers automatically handle resource provisioning, autoscaling, scheduling, and fault tolerance. Furthermore, compared with the serverful approach, serverless inference also offers substantial cost savings as users do not pay for idle resources under the pay-per-use pricing model [15, 45, 49, 51]. In Aliyun Function Compute, the requests to a function typically exhibit dynamic, bursty arrival patterns as shown in Fig. 1, consistent with previous research findings [20, 21, 25, 26, 33, 34, 38, 40, 53]. By leveraging the high elasticity of a serverless platform, inference functions can quickly scale in response to the changing workload, while users are billed based on the actual function runtime at a fine granularity, such as 1 ms [6, 8].

2.2 GPU Support in Serverless Platforms

Despite the benefits of the serverless inference model, existing serverless platforms, including Aliyun Function Compute and other leading platforms, currently lack efficient support for GPUs, which impedes their ability to achieve high-performance serverless inference. In fact, numerous Aliyun Function Compute users have expressed a compelling need to execute their models in GPU-enabled functions.

Existing solutions and their inefficiency. A number of recent systems have been proposed to support GPUs in serverless platforms [1, 23, 24, 45]. They, however, still follow the approach of existing serverful model serving systems (e.g., Nexus [40] and INFaaS [38]), and deploy inference models as long-running containers where each container, when created, is bound to a specific GPU (i.e., early binding). In this approach, the deployed model remains in the memory of a designated GPU to handle future requests, and the occupied GPU resources can only be reclaimed after the model serving terminates.

However, the early-binding approach deviates from the serverless paradigm and is costly for both cloud users and providers. First, binding inference functions to GPUs occupies resources for extended duration, even when idling. Thus, users are obligated to pay for the allocated GPUs regardless of actual usage [3], leading to high expenses that undermine the cost-saving benefits of serverless inference. Second, this approach results in severe GPU underutilization, considering the low average request rates of most inference functions and the cross-GPU load imbalancing. Fig. 2 (left) depicts the distribu-
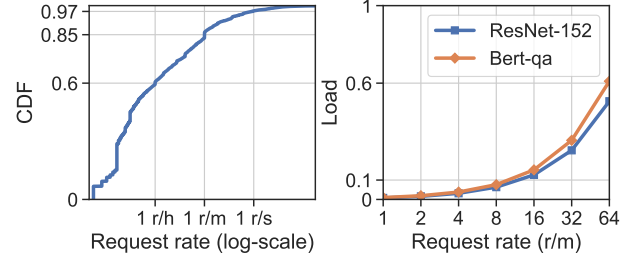

Figure 2: CDF of average function request rate from a one-week production trace (left) and the expected GPU load under various per-function request rates when running multiple functions on a `V100` GPU to saturate its 32 GB memory (right).

Table 1: Model execution time when the inference functions are warm- and cold-started on `V100` GPU, respectively.

Model	Warm-start	Cold-start	Mem. footprint
ResNet-152	24 ms	8 s	1.6 GB
Bert-qa	45 ms	11 s	2.4 GB

tion of the average request rates of Aliyun Function Compute functions in a one-week trace, revealing that 85% (97%) of functions were invoked only once per minute (second) on average[1]. These findings align with the observations from other production traces [8, 39]. Fig. 2 (right) further illustrates that consolidating multiple models to fill GPU memory can still lead to low GPU load. Meanwhile, packing models into a GPU can cause temporary overloading due to the bursty request patterns (Fig. 1), thus inevitably leading to hotspots and load imbalancing in a multi-GPU setting. The impact of load imbalancing will be shown in Fig. 7 with details given in §7.1.

To reduce costs, current systems need to frequently reclaim GPU resources when functions are idle, so that users do not pay for unused GPUs and other functions can access available GPUs for improved utilization. However, this solution can lead to frequent function cold starts, the overhead of which is unacceptable for model inference. Table 1 provides a comparison of model execution times when the inference functions are warm- and cold-started on V100 GPUs in Aliyun Function Compute.[2] Unsurprisingly, cold starts are two orders of magnitude slower due to the need for GPU container setup, ML framework startup (PyTorch in our case), GPU runtime creation, and model loading, all of which lead to extremely long latency that far exceeds the typical SLO requirement of model inference.

Table 2 summarizes current serverless systems and our FaaSwap system. Here, Aliyun Function Compute and

---

[1]For confidentiality reasons, we only depict the request rates of CPU functions, which exhibit similar patterns as those running on GPUs (see Fig. 1).

[2]For cold start, we exclude the delay of fetching a remote container image or a model file, which can take extra seconds to minutes to complete [43].

Table 2: A comparison of FaaSwap and existing solutions that offer GPU support in serverless platforms.

Solution	GPU pay-per-use	GPU efficient	SLO compliance	Model agnostic
Aliyun [1]	×	×	×	✓
Molecule [23]	×	×	×	✓
DGSF [24]	×	×	×	✓
INFless [45]	×	×	*	×
FaaSwap	✓	✓	✓	✓

Alibaba Function Compute [1] are leading commercial serverless platforms that provide GPU-enabled functions; Molecule [23] introduces a serverless platform that supports GPUs and other hardware devices; DGSF [24] enables serverless functions to access GPUs in a remote cluster. All these systems adopt the early-binding approach as previously discussed, which neither enables pay-per-GPU-use billing model for users nor achieves high GPU utilization for cloud providers. Moreover, they are oblivious to the semantics of model inference and unable to meet latency SLOs. Among current systems, INFless [45] is the closest to FaaSwap, which presents a serverless system specifically designed for model inference and supporting GPU functions. Though it proposes a function scheduling scheme that aims to satisfy latency SLOs, it incurs function cold starts, resulting in significant performance degradation and SLO violations (see details in §7.2). Moreover, INFless requires model knowledge for operator-level profiling.

Requirements of GPU-enabled serverless inference. Based on the discussion above and our operational experiences with Aliyun Function Compute customers, we conclude the key requirements of an efficient GPU-enabled serverless inference platform.

R1: Pay-per-GPU-use. One of the main advantages of serverless computing is its pay-per-use billing model. Therefore, users should be billed only when their functions are invoked and running on GPUs (i.e., pay-per-GPU-use).[3] This is crucial for achieving substantial cost savings in the presence of dynamic inference workloads (as shown in Fig. 1), considering the high cost of GPUs.

R2: GPU-efficient inference. For serverless providers like Aliyun Function Compute, minimizing the resource provisioning cost is the key to maintaining market competitiveness. Therefore, the platform should serve as many inference functions as possible using a minimum number of GPUs, thereby attaining the highest GPU utilization. This essentially requires efficient, fine-grained GPU sharing.

R3: Compliance to latency SLOs. The platform should allow users to specify their latency SLOs, e.g., ensuring that at least 99% of inference requests are served within 200 ms [51].

---

[3]Note that, in our experiences, enterprise customers are willing to pay a nominal fee to retain idle functions in host memory for substantially improved performance, similar to the current function keep-alive charge meant to avoid cold-start overheads [3, 7, 29].

The platform should strive to meet the latency SLOs for all requests, if possible.

R4: No detailed model knowledge needed. Today's ML models contain lots of intellectual properties and are of high business value. For model confidentiality, the serverless inference platform should not look into the detailed model structure.

3 Key Insight and Challenges

Key insight. As described in §2.2, the existing early-binding approach retains inference models in expensive and limited GPU memory, resulting in substantial idling costs and GPU underutilization. Thus, an efficient serverless inference platform needs to enable *late binding* that GPUs are managed as a pool and can be dynamically shared by multiple functions, as well as that the inference models reside in host memory when idling and can be swapped into available GPUs on demand when activated by requests.

This approach can effectively satisfy the key requirements identified in §2.2. **First**, keeping models in host memory leads to no GPU memory footprint when idling, which enables the pay-per-GPU-use billing and achieves cost savings for cloud users. **Second**, host memory is much larger than GPU memory (e.g., a few TB vs. tens of GB), allowing consolidation of many functions with low request rates in a single GPU for high GPU utilization. With GPU pooling, it also enables flexible, fine-grained function scheduling by dynamically swapping models to GPUs based on their loads, which effectively load-balances multiple GPUs and further improves overall resource efficiency. **Third**, model swapping can be efficiently performed from host to GPU through PCIe, as well as between GPUs via high-speed NVLink connections when needed. Compared with function cold starts, this approach incurs much less overhead, sustaining low inference latency and facilitating SLO compliance. **Finally**, late binding can be seamlessly performed within a pool of GPUs, which can manage models with a holistic view of memory usage, without requiring specific model knowledge from users.

Challenges. Realizing GPU pooling and late binding in the serverless platform presents three challenges.

C1: Efficient GPU pooling and model swapping. GPU pooling necessitates inference functions to interact with a GPU pool, resulting in high communication overhead compared with native executions (i.e., running on local GPUs). Hence, the platform must deliver high-performance GPU pooling and model swapping to enable low-latency inference.

C2: GPU memory management. The platform needs an effective GPU memory management system to enable each function to execute seamlessly on various GPUs. Since the platform should not hold the detailed model knowledge (e.g., model structure and parameters) of inference functions due to confidentiality reasons, it is required to automatically track
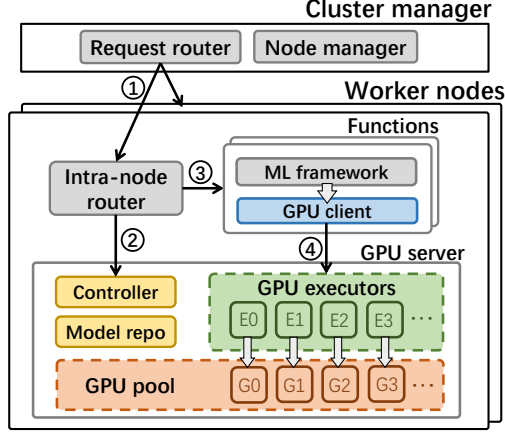
Figure 3: Architecture overview of FaaSwap. A request arriving in the FaaSwap cluster is first routed to the worker node hosting its target function ①. The router in the worker node synchronizes with the GPU server to query the executor for this request ②, and then routes it to the function instance with the target executor ID ③. The function instance next processes the request and uses a GPU client to automatically redirect CUDA API calls to this executor ④, and finally returns the result to the user after request completion.

memory footprint of individual functions and make model swapping transparent to them.

C3: SLO compliance and resource efficiency. The platform must strive to satisfy latency SLOs for inference functions and achieve resource efficiency. It is crucial to understand how model swapping affects end-to-end inference performance, and to develop effective request scheduling and model management algorithms accordingly.

In this paper, we present FaaSwap, a GPU-enabled serverless platform designed to address the aforementioned three challenges. FaaSwap tackles the first two challenges by redesigning existing serverless platforms to enable low-latency GPU remoting and model swapping, and to support efficient GPU memory management (§4). Furthermore, we propose effective algorithms to satisfy latency SLOs and improve resource efficiency (§5).

4 FaaSwap System Design

4.1 Architecture overview

Fig. 3 shows the architecture overview of FaaSwap which contains two main components: cluster manager and worker nodes. The cluster manager takes charge of cluster-level tasks, including request routing, node allocation, and resource scaling. At each worker node, FaaSwap employs GPU pooling, where a GPU server manages all local GPUs as a pool and enables functions to access any available GPUs dynamically. Within the GPU server, a model repository manages models in host memory; GPU executors handles CUDA execution, per-

forms necessary model swapping, and manages GPU memory on associated GPUs; the controller holds a global view of the GPU memory and the executor status, and decides how to schedule requests to executors. Each worker node also runs an intra-node router to signal the GPU server about request arrivals and route requests to local inference functions. When a request arrives at the target function, it interacts with the scheduled executor through a GPU client by remoting CUDA API calls. The GPU server, router, and functions in a worker node all run as containers.

Key to FaaSwap is to build an efficient GPU server that can enable low-latency model inference and address the first two challenges discussed in §3. In particular, we will elaborate upon FaaSwap's designs to answer these questions: 1) how FaaSwap enables low-latency GPU pooling (§4.2) and model swapping (§4.3), i.e., Challenge *C1*, 2) how FaaSwap tracks memory footprint of functions and manages GPU memory (§4.4), i.e., Challenge *C2*, and 3) how FaaSwap ensures the isolation and handles failures (§4.5).

4.2 GPU Remoting

The common approach to GPU pooling is to redirect individual CUDA API calls from functions to GPU executors, transparent to the inference program [13, 36]. However, this approach necessitates synchronization for each API call, resulting in high synchronization overhead that is unacceptable to model inference. Based on our measurement, each inference needs thousands of CUDA API calls (e.g., over 4k calls for ResNet-152), which incur an additional delay of several hundred milliseconds in communication (Table 4). To address this issue, FaaSwap proposes the asynchronous API redirection.

Asynchronous API redirection. We observe that intermediate steps in an inference execution are typically performed asynchronously in GPU — the intermediate data gets generated and consumed in GPU memory without requiring any data transfer to the host until the execution is completed. Therefore, a function can redirect intermediate CUDA calls to the GPU executor asynchronously without waiting for their results, and perform synchronizations only for the final output. This approach does not affect the execution order, and thus can still ensure the correctness of the model inference.

Following this observation, we perform asynchronous redirection for CUDA APIs. In particular, we divide the set of CUDA APIs into two categories based on their semantics: 1) synchronous, blocking APIs, and 2) asynchronous, non-blocking APIs. The former needs the host to wait for their completion and use the outputs in the following steps, e.g., cudaMalloc, and thus we perform synchronizations by default. The latter does not change the runtime state in the host, e.g., cudaLaunchKernel, allowing asynchronous API redirection without blocking. FaaSwap supports the common CUDA runtime APIs and CUDA libraries, e.g., cuDNN. We show

the category of each such API in Appendix A.1.

With asynchronous API redirection, we can fuse multiple consecutive CUDA API calls into a single group and send them together, which further reduces communication costs. We design an effective grouping strategy to improve the performance of API redirection. We note that fusing too many calls into one group (e.g., all intermediate API calls) can mitigate communication costs, which however needs the function to wait until all calls are issued. On the other hand, having too few calls in one group (e.g., one call per group) has no extra delay but incurs considerable communication costs with frequent API redirections. We observe that various models exhibit similar API call generation patterns as they are constructed wih common blocks (e.g., convolutional layers). Therefore, we leverage representative API call patterns to profile the performance of API redirection, where we vary the group size to find an optimal one that can generally deliver good performance.

Table 4 in §7 shows the performance benefits of our asynchronous, group-level API redirection. Compared with synchronous API redirection, FaaSwap can cut the inference latency of popular models by up to an order of magnitude, owing to significantly reduced communication costs. Interestingly, FaaSwap can even outperform the native execution (i.e., using local GPU without GPU remoting) for many evaluated CNN models (e.g., ResNet). Serving these models requires configuring many cuDNN descriptors (e.g., cudnnSetConvolutionNd-Descriptor), where the relevant CUDA APIs are executed on the CPU side and do not require GPU resources. As a result, redirecting these APIs effectively distributes CPU-side workloads across functions and the GPU server, enabling functions to access more CPUs and perform parallel computation. For Bert models that trigger no cuDNN descriptor-related APIs, our approach yields performance comparable to native execution. Note that, CPU resources are not the bottleneck despite the utilization of more CPUs in GPU remoting. Based on our measurements (see testbed in §7), when GPUs in a node are fully occupied for serving requests, the CPU utilization of FaaSwap (native execution) for ResNet-152 and Bert-qa remains at 27.4% (8.8 %) and 17.2% (9.2%), respectively.

GPU runtime sharing. GPU programs need GPU runtime to manage GPU-side states, which can account for a considerable portion of memory footprint, e.g., about 1 GB for models in Table 1. To improve memory efficiency, in FaaSwap each GPU executor shares a single GPU runtime across functions it hosts. This dramatically reduces GPU memory footprint, and alleviates the need of creating a new runtime after model swapping which can take a few seconds. FaaSwap also preloads all CUDA kernels on each GPU to avoid extra loading overheads. We will discuss fault isolation of FaaSwap in §4.5.

4.3 Model Swapping

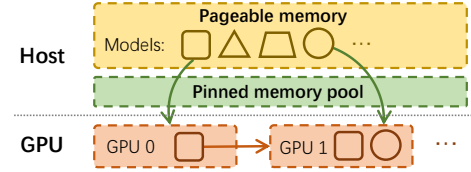Since serverless platforms should not look inside each model

Figure 4: An example of model swapping. Models can be swapped from host to GPU through PCIe (green arrows), or across GPUs through NVLink (red arrow).

to gain the detailed model structure due to business and confidentiality reasons, FaaSwap tracks the general memory footprint of a model to enable model swapping during runtime. FaaSwap capitalizes on the observation that the memory access pattern of a model generally remains consistent across requests, including the addresses and access order of model parameters. Therefore, FaaSwap only needs to track the first function execution (i.e., cold start), the access pattern of which can be applicable to future request executions (see memory tracking details in §4.4). FaaSwap then performs model swapping on demand at the request level, and more importantly, enhances swapping performance through a model pipeline execution.

Model swapping and pipeline execution. Fig. 4 illustrates the model swapping in FaaSwap, which swaps models from host memory to GPUs through PCIe and also supports fast model transmission to neighboring GPUs through NVLink. Specifically, by utilizing pinned memory, the direct memory access (DMA) data transfer is enabled to facilitate the host-to-GPU model swapping. Model swapping can be performed on demand — a GPU executor can load a target model to its associated GPU when notified by request arrivals (② in Fig. 3). Combined with CUDA API redirection (§4.2), FaaSwap can easily execute a model on different GPUs across requests, thus enabling late binding.

In addition, FaaSwap exploits the pipelining opportunity to hide the overhead of model swapping. We note that model inference only involves a forward pass and is typically executed layer by layer, which creates an opportunity for overlapping the transmission of subsequent layers with the computation of previous layers. Key to model pipelining is to determine the number of layers to group for swapping. Grouping too few layers necessitates a frequent waiting for the transmission completion to trigger the computation, leading to high synchronization overhead. Conversely, grouping too many layers results in less overlap between transmission and computation, impairing pipelining efficiency.

Previous systems have proposed various model pipelining strategies, such as PipeSwitch [17] and DeepPlan [28], but they are not applicable to serverless inference. Existing strategies assume that the model is provided in advance, and they conduct extensive profiling to obtain the execution time of each model layer to find the optimal pipelining strategy. However, in practice, serverless platforms should not access the

6

detailed model structures for confidentiality reasons. Therefore, FaaSwap employs a model-agnostic approach to determine the group size in model pipelining. We observe that the transmission performance, concerning group sizes, can experience an "elbow point": increasing the amount of data in a group improves the overall transmission throughput, but the improvement becomes marginal after a certain point. We therefore select this elbow point as the group size, which can achieve good swapping performance without significantly impacting pipelining efficiency. Since the optimal size only depends on hardware configurations, such as PCIe bandwidth, we can easily find it by profiling transmission performance of various-sized data (e.g, about 2 MB based on our testbed). This approach requires no detailed model knowledge or profiling and can be directly applied to various models.

We show the performance of FaaSwap's pipelining strategy in Table 4 in §7. Compared with separate model swapping and inference execution, i.e., non-pipelining, FaaSwap's pipelining execution achieves better end-to-end performance, reducing latency by about 50%. Model pipelining via high-speed NVLink further improves the performance due to the reduced swapping overhead, which is comparable to the inference execution only ("Remote Async.").

Model eviction. We observe that model transmission from GPUs to host can result in significant overhead and interfere with concurrent inference executions. Therefore, FaaSwap always maintains a copy of the model in the host and only invalidates its GPU memory region during eviction. By utilizing the cheap host memory, FaaSwap can effectively eliminate the overhead of model eviction.

4.4 Memory Management

Memory management is critical to make model swapping and late binding transparent to users' inference functions. In FaaSwap, a model should be dynamically executed on various GPUs without requiring users to incorporate any "swapping logic" in their inference code, and users should remain oblivious to swapping actions.

Existing solutions to GPU memory management [9, 19, 27, 35] are not directly applicable to FaaSwap for two reasons. First, late binding requires all GPUs in a pool to share the same logical memory space as inference functions do not recognize backend GPUs and consistently access models using identical memory addresses, even across different GPUs. However, current unified memory management solutions, such as HUVM [19], only allow individual GPUs to utilize memory in the host or other devices, failing to support seamless task migration across GPUs. Second, model swapping necessitates FaaSwap to frequently allocate and deallocate memory, but native APIs (e.g., cudaMalloc) can introduce substantial overhead and significantly impair overall performance. We therefore design a GPU memory management system for FaaSwap to address these issues.

Memory address management. When model swapping occurs, either from host to GPU or between GPUs, the actual memory addresses of model parameters can differ from the original ones. To enable seamless late binding, FaaSwap meticulously manages memory layout and automatically translates each memory access of CUDA calls to actual physical addresses on a target GPU. FaaSwap leverages the memory layout of ML frameworks to facilitate the memory address management. ML frameworks, such as PyTorch and Tensor-Flow, typically organize data into blocks for simplified management, wherein each GPU memory block can contain multiple parameters. This practice enables FaaSwap to perform memory mapping at the block level.

按照 pytorch 内存管理的 block level 进行

Specifically, FaaSwap monitors memory blocks for each function and maintains a mapping to their actual physical addresses after model swapping. The internal data layout within each block, such as the offsets of its parameters, remains unchanged, allowing FaaSwap to easily obtain the physical address of a particular parameter using its associated block address and the corresponding offset. As a result, FaaSwap does not need to maintain extensive metadata for individual data pointers, effectively handling address translation without incurring significant management overhead.

Memory allocation and block management. Based on our measurements, using native APIs to allocate memory blocks for a single model can result in a high overhead of up to hundreds of milliseconds (Fig. 10), which hinders overall swapping and inference performance. FaaSwap hence pre-allocates all GPU memory and internally manage all blocks to avoid relying on native APIs for block allocation. Key to block management is to efficiently reduce memory fragments. To address this issue, a classic approach is the Buddy memory allocation [32], which divides and merges idle blocks based on power-of-two multiples. We revise this approach by exploiting two characteristics of FaaSwap.

First, we leverage block usage patterns of ML frameworks for reduced memory fragments. For instance, PyTorch employs fixed-size blocks (e.g., 20 MB) to host small- and moderate-sized data, resulting in high popularity and easy sharing of these blocks across various models. Motivated by this observation, FaaSwap consolidates these fixed-size blocks to reduce fragmentation. In particular, FaaSwap divides all GPU memory into a number of *memory partitions* during bootstrap, which are created via the native allocation API and have consistent sizes. FaaSwap consolidates the fixed-size blocks into a few memory partitions, which require only simplified memory layouts and can be generally shared across models. Irregular-sized blocks are managed separately in different partitions, where we adopt the classic Buddy allocation to reduce memory fragments. Second, we note that all blocks of a single model are usually accessed entirely during model swapping and execution, as well as reclaimed together after model eviction. This observation inspires us to package blocks from the same model as tightly as possible, e.g., by

collocating them into a single memory partition. This approach enables model eviction to easily free entire memory partitions, making them available for future block allocation. In addition, FaaSwap also periodically consolidates blocks to reduce overall fragments.

4.5 Isolation and Fault Handling

We next discuss how FaaSwap handles resource and fault isolation across function instances.

Resource isolation. With GPU pooling, FaaSwap provides container-level isolation for CPU and memory resources[4], similar to existing serverless platforms [1, 42, 48]. For GPUs, FaaSwap performs software-based isolation at its GPU server, which ensures GPU compute resources, e.g., streaming multiprocessors, are exclusively allocated to function instances at a request granularity. It also isolates GPU memory by prohibiting functions from accessing memory regions of others, which can be easily achieved by GPU memory management (§4.4).

Fault handling and isolation. FaaSwap sustains various system component failures. In case of function failures, FaaSwap simply restarts them to resume the execution, which has no side effect due to the stateless nature of inference. For executor failures at a GPU server, FaaSwap migrates affected models to other active executors via swapping, and then restarts the failed ones. The GPU server also persists runtime states in local storage (e.g., models and metadata) to allow fast recovery from the failure of an entire server. Therefore, FaaSwap can effectively handle faults occurring during function execution, and isolate them across various functions.

At the cluster level, FaaSwap persists metadata of individual nodes in a database, and thus the cluster manager can easily retain these states and recover from failures. It also keeps periodic health checks with the router of each worker node, and handles node failures by launching a new node and migrating all relevant functions.

5 FaaSwap Policy Design

We present how FaaSwap meets the latency SLOs for inference functions and delivers resource efficiency (i.e., Challenge *C3* in §3). We start with the design overview, followed by individual policies.

5.1 Design Overview

Objective. The overall objective of FaaSwap is to meet latency SLOs for all inference functions while minimizing resource cost. We define a function to comply with SLOs if its tail request latency is not longer than a user-specified deadline,

---

[4]FaaSwap makes no assumption on sandboxes and can also support microVMs [14].

Table 3: Latency (ms) of model pipelining execution when concurrently swapping other models through PCIe. The diagonal values indicate the latencies without concurrent models.

Model	DenseNet-169	ResNet-152	Bert-qa
DenseNet-169	**27**	27 (+0%)	27 (+0%)
ResNet-152	31 (+7%)	**29**	43 (+48%)
Bert-qa	166 (+11%)	240 (+61%)	**149**

and meter the resource cost by the number of worker nodes. Key to achieving this goal is to *maximize the number of SLO-compliant functions* at each worker, such that FaaSwap can efficiently exploit per-worker GPU resources to host as many functions as possible, which in turn reduces the total number of workers required.

Challenges. Previous inference systems have proposed various schemes to meet latency SLOs [25, 45, 51, 53]; however, their policies do not apply to FaaSwap for two reasons. First, existing systems such as INFless [45] and Shepherd [53] assume sufficient GPU memory and employ early binding, which schedule model serving instances to GPUs and then batch and route requests to them. In contrast, FaaSwap focuses on late-binding the lower-frequency functions to a pool of memory-constrained GPUs, requiring a joint design of model management (i.e., model swapping and eviction) and request scheduling. Second, existing systems assume a stable model inference latency [25, 38, 45, 51, 53], but this assumption does not hold in our settings — model swapping can cause unpredictable end-to-end performance due to PCIe bandwidth contention [16, 30]. As show in Table 3, concurrently swapping two models through PCIe increases individual model inference latency compared with running them alone, especially for large models (e.g., Bert-qa).

We propose three policies to address the aforementioned challenges. First, considering that packing many functions together can cause short-term overloading and request queueing, FaaSwap introduces a request prioritization policy to maximize the number of SLO-compliant functions (§5.2). Second, FaaSwap designs a request scheduling and model swapping policy to reduce bandwidth contention across concurrent models, thereby improving overall inference performance (§5.3). Lastly, by leveraging the characteristics of model swapping, FaaSwap proposes an effective model eviction policy to reduce bandwidth footprint in model swapping; combined with the request scheduling policy before, FaaSwap minimizes the interference among concurrent model executions (§5.4).

5.2 Request Queueing

Late binding enables FaaSwap to consolidate multiple inference functions on a worker node, thereby improving GPU utilization. However, it also introduces a potential short-term node overloading as inference functions exhibit dynamic request arrival patterns (Fig. 1). To maximize the number of

SLO-compliant functions, FaaSwap should monitor the SLO compliance of individual functions at a worker node and determine the prioritization of their requests. Intuitively, FaaSwap prioritizes functions according to their likelihood of meeting SLOs, where functions with a higher probability of compliance get executed earlier to mitigate queueing delays.

Following this insight, FaaSwap divides all functions on a node into two groups based on their potentials to meet SLOs, and respectively maintains two queues for their requests, i.e., high- and low-priority. Only when the high-priority queue is empty does FaaSwap dispatch low-priority requests. Functions can be moved between the two groups dynamically, depending on their likelihood of SLO compliance. However, realizing this approach requires answering two questions: 1) how to quantify the likelihood of SLO compliance for a function, and 2) how to effectively partition functions into high- and low-priority groups.

Metric for function prioritization. The prioritization metric should reflect the "degree of needed effort", where functions requiring less effort are more likely to achieve SLOs. We therefore propose *required request count* (RRC), which measures the expected number of future requests to a function that must be served within specified deadlines to meet SLOs. Let n be the current number of requests to a function, and m be the number of requests served within deadlines from the total n requests. The RRC of the function can then be defined as $\frac{pn-m}{1-p}$, where p is the tail percentile specified in SLOs, e.g., 98%. This is simply derived from the equation: $\frac{m+RRC}{n+RRC} = p$. RRC values for various functions can be normalized by the average request latency. Thus, functions with negative RRCs should have already met SLOs up to now, while functions with larger RRCs have a lower likelihood of meeting SLOs. This metric allows us to prioritize function requests with smaller RRC values.

Divide functions into two priority groups. With RRCs, we can divide functions into high- and low-priority groups, and then determine the request execution order. Determining the RRC boundary between the two groups can be challenging — having too many (few) high-priority functions can be too aggressive (conservative) to enable more SLO-compliant requests. In FaaSwap, we use a threshold $\alpha \in [0, 1]$ to indicate the boundary and determine how functions are prioritized: we prioritize more functions by increasing α; when α is 1, all functions are put in the high-priority group. In particular, consider a node with N functions sorted by RRCs, and let RRC_i be the RRC of function i. We put the first k functions in the high-priority group, where k is the largest integer such that $\sum_{j=1}^{k} \max(RRC_j, 0) \leq \alpha \cdot \sum_{i=1}^{N} \max(RRC_i, 0)$. FaaSwap automatically configures α at runtime based on the overall load and function SLOs. When there is a short load surge and the number of SLO-compliant functions decreases, FaaSwap turns to be conservative with a small α; otherwise, α increases to prioritize more functions. We defer the detailed algorithm
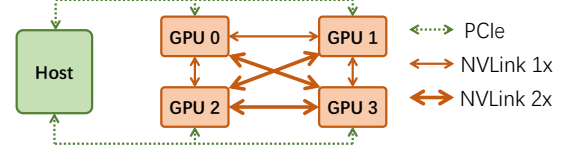

Figure 5: Topology of a 4-GPU worker node in Aliyun Function Compute.

of the α auto-configuration to Appendix A.2.

The request execution order in each priority queue is also determined using function RRCs. In the high-priority queue, requests are prioritized in a *reverse* order of RRCs. This strategy favors functions with a high likelihood of SLO compliance (i.e., with small positive RRCs) over those already meeting SLOs (i.e., with negative RRCs), which effectively increases the number of SLO-compliant requests whenever feasible. In contrast, requests in the low-priority queue are executed following the order of RRCs.

5.3 Scheduling and Model Swapping

FaaSwap aims to minimize the inference latency for individual requests as they are dispatched from the queues. Specifically, when a request is received by FaaSwap's GPU server (② in Fig. 3), the controller decides which GPU (executor) should load the designated model and handle the request. Each GPU processes one function request at a time to ensure resource isolation.

Bandwidth contention in model swapping. While model execution latency is often stable [25], FaaSwap's model swapping can incur unpredictable overhead due to the PCIe bandwidth contention [16, 30]. Fig. 5 shows the topology of a worker node in Aliyun Function Compute, where each pair of GPUs shares a PCIe switch and GPUs are inter-connected via NVLinks with various bandwidths, e.g., faster NVLinks delivering 2× higher throughput than slower ones. The performance slowdown caused by bandwidth contention can vary among models as shown in Table 3, where larger models like Bert-qa require more intensive data transmission and exhibit more pronounced performance degradation. Moreover, concurrently swapping a target model with light ones (e.g., DensNet-169) leads to reduced contention and lower latency compared with bandwidth-intensive ones (e.g., Bert-qa). Therefore, we propose to leverage this characteristic to reduce interference during the model swapping and enhance the overall performance.

Interference-aware scheduling. For each request, FaaSwap aims to minimize its interference with concurrent workloads, which in turn reduces inference latency. We propose two designs to achieve this. First, FaaSwap avoids concurrent swapping for bandwidth-intensive models whenever possible. It divides models into two categories based on their bandwidth intensiveness in swapping, i.e., heavy and light models, which can be easily done via basic model profiling: if model pipelin-

9

Algorithm 1 Interference-Aware Request Scheduling

1: **function** SCHEDULE(req)
2:     $A \leftarrow$ set of available GPUs   ▷ $A \neq \varnothing$, otherwise queueing req
3:     $M \leftarrow$ set of GPUs hosting the target model
4:     **if** $M \neq \varnothing$ **then**
5:         $G \leftarrow M \cap A$
6:         **if** $G \neq \varnothing$ **then**
7:             $g \leftarrow$ any GPU in G
8:             **Execute** *req* on g ▷ Without swapping
9:         **else**
10:            $(g, m) \leftarrow$ GPU pair with fastest NVLink, $g \in A, m \in M$
11:            **Execute** *req* on g; **Swap** model from m ▷ GPU-to-GPU swapping
12:    **else**
13:        $g \leftarrow$ a GPU whose neighbor is not loading models, $g \in A$
14:        **if** g not found **then**
15:            $g \leftarrow$ a GPU whose neighbor is loading a light model, $g \in A$
16:        **if** g not found **then**
17:            $g \leftarrow$ any GPU in A
18:        **Execute** *req* on g; **Swap** model from host   ▷ Host-to-GPU swapping

---

ing significantly slows down the inference execution, the data transmission is the bottleneck and thus the model is heavy (see Table 4). We do not need accurate swapping performance under concurrency, which in fact is hard to obtain. Second, FaaSwap exploits the direct NVLink connections between GPUs to reduce PCIe contention. FaaSwap prioritizes GPU-to-GPU over host-to-GPU model swapping such as to enable faster model transmission and avoid interference with concurrent PCIe traffic.

Algorithm 1 shows FaaSwap's scheduling and swapping policy. For a request, FaaSwap first checks whether the target model is loaded on an available GPU, and if so, directly executes it without the swapping overhead (line 8). If the model is hosted by busy GPUs, FaaSwap then schedules the request to perform the GPU-to-GPU swapping, in that the source and target GPUs should have a fast NVLink connection (line 11). Otherwise, FaaSwap resorts to the host-to-GPU swapping and prioritizes target GPUs whose neighbors are idle or running light models in order to reduce PCIe contention (line 18). In a nutshell, FaaSwap minimizes the interference and overhead of model swapping for each request, thereby providing low inference latency.

5.4   Model Eviction Policy

Model eviction plays a critical role in reducing bandwidth contention and enhancing the overall inference performance, in conjunction with FaaSwap's request scheduling policy. Unlike the traditional cache eviction strategies, which primarily aim to minimize the miss rates, FaaSwap's model eviction policy considers the performance implications of model swapping for different models to facilitate future model loading.

We notice that swapping light models leads to negligible overhead for end-to-end performance compared with heavy ones (Table 3 and Table 4). Therefore, we tend to evict models that have almost no impact on performance when swapping, which allows for caching more heavy models in GPUs for reduced host-to-GPU data transmission and interference. Following this insight, we divide models into two groups: heavy models that are hosted on only a single GPU should be prioritized, and the rest are of lower priority and can be evicted earlier including light models and the heavy ones with copies on multiple GPUs. As a result, FaaSwap only needs to frequently perform the host-to-GPU and cross-GPU swappings for light and heavy models respectively, which lead to negligible or no PCIe bandwidth contention and thus achieve low swapping overhead. In each group, we adopt a common Least-Recently-Used (LRU) policy to determine the eviction order for its models.

5.5   Put All Together

With the aforementioned policies, FaaSwap effectively maximizes the number of SLO-compliant functions at each worker node. At the cluster level, FaaSwap monitors the request load at individual nodes and the SLO compliance of individual functions, aiming to eliminate any potential SLO violations. Note that, a worker node may discard hot functions when it becomes overloaded and cannot meet latency SLOs for functions (§5.2). In this case, FaaSwap's cluster manager can migrate these functions onto other nodes with available resources and provision new nodes when needed, which further maximizes the SLO compliance for functions at low resource cost.

6   Implementation

We have implemented FaaSwap atop Aliyun Function Compute, one of the world's leading commercial serverless platforms. FaaSwap's GPU server and GPU client are implemented in 4k and 1.5k lines of C++ code, respectively. Intra-node router and cluster manager are directly implemented atop the relevant components in Aliyun Function Compute. We also implement a demo router in 530 lines of Python code to provide the basic functionalities for single-node tests. FaaSwap's cluster manager can maintain and track a resource pool of GPU nodes to ensure fast node allocation and scaling, following the existing practice in Aliyun Function Compute. We provide a container image as a function template based on PyTorch, where the original CUDA libraries (e.g., libcudart.so) are replaced by our GPU clients to enable GPU remoting. This requires no modification to the PyTorch framework.

7 Evaluation

In this section, we evaluate FaaSwap using production traces from Aliyun Function Compute. Our evaluation ansewrs the following questions:

- Can FaaSwap enable efficient GPU remoting and model swapping (Table 4)?

- How much benefit FaaSwap's model swapping can bring in terms of overall performance (§7.1)?

- Can FaaSwap maximize the number of SLO-compliant functions at node and how does its individual design policies contribute to overall performance gain (§7.2)?

- Can FaaSwap satisfy per-function latency SLOs and improve resource utilization at cluster (§7.3)?

Settings. We deploy FaaSwap in a Aliyun Function Compute cluster following production environments. FaaSwap runs on a cluster with up to 6 workers. Each worker node has 48 vCPU cores, 384 GB memory, and 4 NVIDIA V100 GPUs, each with 32 GB memory. We use 8 popular ML models in evaluation, as shown in Table 4, and distribute them across inference functions in a round-robin manner. Table 4 also shows the performance of GPU remoting and model pipeline, which we discuss in §4.2 and §4.3, respectively. We warm up all functions before running test workloads to exclude cold starts.

Metrics We focus on the ratio of functions meeting SLOs and GPU load in evaluation. A function can comply with SLOs if its tail request latency is less than a deadline. By default, we use 98th tail latency, and set the deadlines for CV models and Bert-qa to 80 ms and 200 ms, respectively. The load is measured by the proportion of duration when the GPU processes inference requests.

7.1 FaaSwap's Model Swapping

We first discuss the benefits of FaaSwap's model swapping.

Host-to-GPU model swapping. With host-to-GPU model swapping, FaaSwap substantially reduces memory footprint of inference functions and results in high GPU utilization. Fig. 6 illustrates the latency and throughput at various request rates under FaaSwap and native execution (Native). Native simply employs early binding and keeps functions alive in GPUs, which is constrained by limited GPU memory and can only host a small number of functions. Consequently, Native exhibits poor overall throughput under a rate of tens of requests per second, the pattern of which cover a majority of serverless functions (Fig. 2). In contrast, FaaSwap supports a significantly larger number of functions using host memory, resulting in a substantial improvement in throughput, e.g., over 10× under 10 r/m. Only at high request rates, such as 120 r/m per function, where GPU memory is no longer
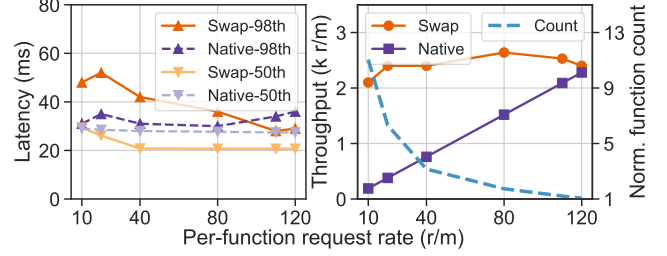

Figure 6: Performance of executing multiple ResNet-152 functions on a single GPU under FaaSwap's model swapping (Swap) and native execution (Native), which packs as many as possible functions on the GPU to saturate its memory. We show the median and 98th tail latencies under various per-function request rates (left), and the aggregate throughput and the function count in Swap normalized to Native (right).
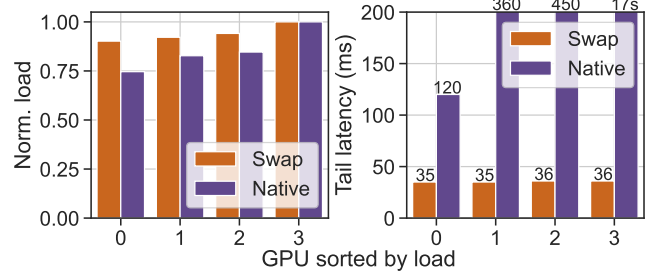

Figure 7: Per-GPU load normalized to the maximum (left) and 98th tail latency of requests on each GPU (right) under FaaSwap's model swapping (Swap) and Native.

the bottleneck, Native can achieve a throughput similar to FaaSwap. For inference latency, FaaSwap delivers comparable performance with Native even under model swapping. For instance, under a request rate of 10 r/m, the tail inference latency with model swapping can sustain at about 50 ms, effectively satisfying latency SLOs. Under high request rates, FaaSwap results in lower latency compared with Native due to its efficient implementation of GPU remoting (§4.2 and Table 4).

GPU-to-GPU model swapping. Fig. 7 compares FaaSwap with Native on a 4-GPU worker. In Native functions are bound to specific GPUs, which can easily cause GPU hot spots. In contrast, FaaSwap enables GPU-to-GPU swapping and can effectively migrate models for load balancing. Fig. 7 (left) shows per-GPU load normalized to the maximum, where FaaSwap can lead to less variance across 4 GPUs compared with Native. Moreover, load imbalance in Native can greatly impair the performance of model inference due to severe request queueing. Fig. 7 (right) shows the 98th tail latency of requests executed on each GPU, where Native leads to extremely long tail latency, e.g., over seconds. Unlike Native, FaaSwap can distribute load across GPUs via efficient GPU-to-GPU model swapping, which consistently achieves fast model inference and cuts the tail latency to around 35 ms for all GPUs.

Table 4: ML models and the latency (ms) of GPU remoting and model swapping. Bert-qa [22] is a popular transformer-based NLP model and others are popular CV models. Models are marked heavy if the swapping (Pipeline PCIe) significantly slows down the inference (Remote Async.).

Model	GPU Remoting			Model Swapping in Execution			Heavy?
	Native	Remote Sync.	Remote Async.	Non-pipeline	Pipeline PCIe	Pipeline NVLink	
ResNet-50	11	82	9	23	13	11	Yes
ResNet-101	20	157	14	35	22	16	Yes
ResNet-152	27	236	19	45	29	21	Yes
DenseNet-169	30	262	25	34	27	26	No
DenseNet-201	36	331	28	39	30	30	No
Inception-v3	19	151	14	27	17	16	No
EfficientNet	17	101	12	17	13	13	No
Bert-qa	45	92	45	190	149	48	Yes

7.2 FaaSwap at Node

We next evaluate the performance of FaaSwap at a node. We evaluate FaaSwap using real-world workloads sampled from production traces (Fig. 2 left). The function request rates range from 5 to 30 r/m, which are representative for GPU inference functions as we discuss in §2.

Performance under varying functions. We first compare FaaSwap with two baselines, native execution and INFless [45]—a state-of-the-art serverless inference system. Unlike model swapping, INFless introduces a function keep-alive policy to balance the GPU cost and inference latency, which learns from historical traces to set the lifespan of individual functions. For a fair comparison, we implement the keep-alive policy of INFless (INFless-KA) in the Native system. Note that INFless also proposes a scheduling algorithm that aims to optimize function placement and resource allocation (i.e., early binding) to meet their latency SLOs. However, it is not directly applicable in our evaluation: limited GPU memory results in a queue of functions waiting to be launched; whenever an idle function is reclaimed, it must immediately schedule a waiting function to the reclaimed slot, leading to a constrained decision space. Fig. 8 shows the ratio of functions meeting SLOs in Native, INFless-KA, and FaaSwap (left) and further reveals the actual function count in each case when running 160 functions (right). With model swapping, FaaSwap can efficiently support 160 functions and deliver low inference latency, consistently meeting SLOs for all functions. In contrast, Native experiences a decreasing ratio as the number of functions increases (e.g., a maximum of 72 functions), which results from the limited GPU memory. Compared with Native, INFless-KA can reclaim GPU memory by cleaning up idle functions, thereby enabling the execution of more functions (e.g., 112 functions). However, this approach inevitably incurs function cold starts, which significantly degrades overall inference performance and results in an extremely low ratio of SLO-compliant functions, with only 7 out of 160 achieving SLO compliance.

Benefits of FaaSwap's policies. To understand the benefits
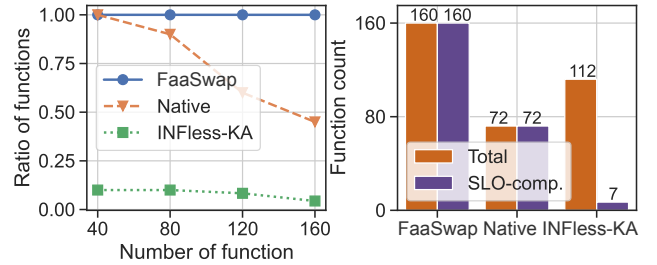

Figure 8: Ratio of SLO-compliant functions under Native, keep-alive policy of INFless (INFless-KA), and FaaSwap (left). We show the actual numbers of total and SLO-compliant functions in each case under 160 functions (right).

of policy designs, we use four baselines that disable individual policies in FaaSwap. (1) FaaSwap-*FIFO* uses a FIFO policy in request queueing compared with our SLO-aware policy (§5.2). (2) FaaSwap-*Random* disables interference-aware scheduling (§5.3), which randomly schedules a request to an idle GPU if the target model is not loaded, and triggers model swapping through PCIe. (3) FaaSwap-*LRU* directly adopts a LRU policy in model eviction rather than prioritizing models according to swapping overhead (§5.4). (4) FaaSwap-*Block* disables block management policy (§4.4), which simply caches released memory blocks of various sizes in a single pool. When model loading requires a new block, it directly returns a cached one in the pool if the requested size can be satisfied, otherwise it will free existing idle blocks until the required memory space is available.

Fig. 9 shows the ratio of SLO-compliant functions using FaaSwap and four baselines. Compared with FaaSwap, all the baselines suffer from various limitations and fail to effectively support a large number of functions. In particular, FaaSwap-FIFO is oblivious to SLOs and unable to properly prioritize functions, leading to serious SLO violations under many functions, such as over 50% of total 560 functions cannot satisfy the SLOs. FaaSwap-Block cannot reuse various-size blocks and forces frequent memory allocation via native CUDA API, which incurs long delay in block allocation and
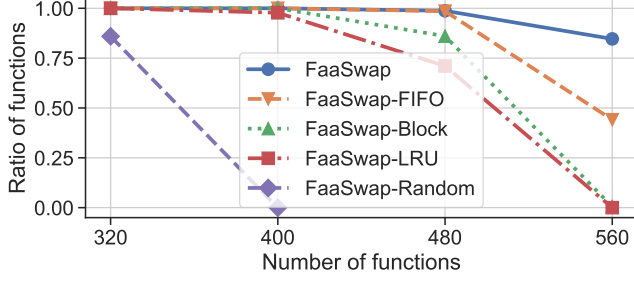
Figure 9: Ratio of SLO-compliant functions under FaaSwap and various policies.

significantly harms overall performance as we will describe in Fig. 10. FaaSwap-LRU frequently evicts heavy models, leading to PCIe bandwidth contention during future swapping and impaired inference performance. Therefore, the function ratios quickly drop to 0 for both FaaSwap-Block and FaaSwap-LRU. FaaSwap-Random leads to the worst performance due to its inefficient scheduling and swapping policy, which does not exploit NVLink across GPUs and is oblivious to model heaviness. Consequently, it easily violates SLOs even under 320 functions. Compared with baselines, FaaSwap can successfully support over 80% functions under 560 functions, effectively maximizing the number of SLO-compliant functions.

Policy behaviors. Fig. 10 further shows the behaviors of FaaSwap's block management and model eviction policies. In particular, we compare the latency of per-model block allocation under FaaSwap-Block and FaaSwap (Fig. 10 left). FaaSwap incurs only negligible overhead due to efficient block sharing (§4.4), which requires no native GPU memory allocation. In contrast, FaaSwap-Block can easily trigger many CUDA allocation calls when swapping models and thus cause a long delay, e.g., up to hundreds of milliseconds. Fig. 10 (right) breaks down the proportion of three swapping cases under FaaSwap-LRU and FaaSwap, which include non-swapping and host-to-GPU (Swap PCIe) and GPU-to-GPU (Swap NVLink) swapping. FaaSwap's eviction policy tends to keep heavy models in GPUs such as to reduce overall swapping overhead. For example, over 90% of requests to heavy models incur no host-to-GPU swapping, which effectively avoids PCIe bandwidth contention. While swapping through PCIe is required by most requests to light models, this leads to negligible impact on overall performance (Table 4). On the other hand, FaaSwap-LRU is oblivious to model knowledge and thus both light and heavy models observe similar patterns.

SLO-aware request queueing. We next evaluate FaaSwap's SLO-aware request queueing policy (§5.2). We compare FaaSwap with FaaSwap-FIFO under 560 ResNet-152 functions and vary their deadlines from 60 ms to 80 ms. Fig. 11 shows the ratio of SLO-compliant functions using FaaSwap-FIFO and FaaSwap. FaaSwap's policy is designed to satisfy various user-specified SLOs and thus we vary its target deadline accordingly, by which FaaSwap adjusts request execution
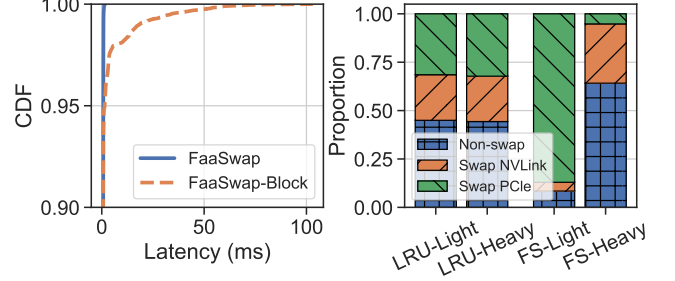

Figure 10: Behaviors of FaaSwap's block management and eviction policies: the latency of block allocation under FaaSwap and FaaSwap-Block (left), and the proportion of three swapping cases under LRU and FaaSwap's eviction policies (right).
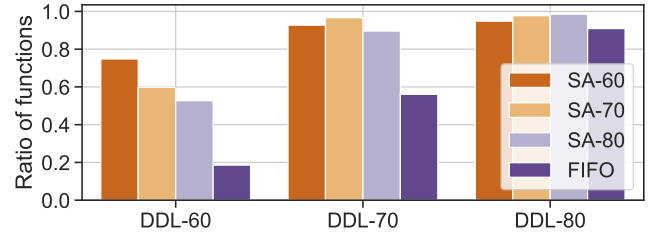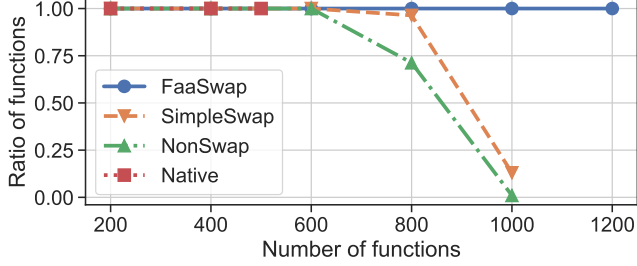

Figure 11: Ratio of SLO-compliant functions using FIFO and FaaSwap's SLO-aware (SA) policies. We set deadlines from 60 ms to 80 ms, and vary the targets in SA accordingly.

order to support as many functions as possible. In particular, SA-60, SA-70, and SA-80 can achieve the best performance when setting deadlines to 60 ms, 70 ms, and 80 ms, respectively. All of them can significantly outperform FaaSwap-FIFO in either deadline.

7.3 FaaSwap at Cluster

We next evaluate FaaSwap on a cluster deployment with 6 GPU workers. As running FaaSwap on a Aliyun Function Compute cluster incurs additional system overhead, we relax the SLOs and set deadlines for CV models and Bert-qa to 150 ms and 250 ms, respectively.

Baselines. We exclude INFless-KA from our cluster deployment due to its extremely poor performance (Fig. 8). We use three baselines: *(1) Native* uses native GPU containers bound to specific GPUs, which is an existing practice in Aliyun Function Compute. Each GPU worker can only host a fixed, small number of functions due to limited GPU memory (§7.1). *(2) NonSwap* allows GPU remoting similar to FaaSwap, but disables model swapping. Compared with Native, NonSwap can share GPU runtime across functions, which reduces memory footprint and enables each GPU to host more function. *(3) SimpleSwap* enables model swapping compared with NonSwap. However, this approach does not incorporate policies implemented in FaaSwap, but supports simple strategies discussed in §7.2, including FIFO request queueing, random

(a) Ratio of SLO-compliant functions under FaaSwap and baselines.
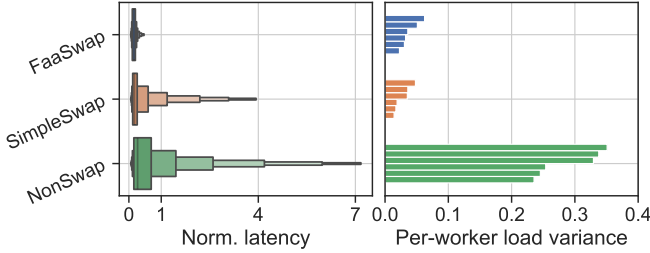
(b) Distribution of request latencies normalized to corresponding deadlines (left) and Per-worker variance of GPU load normalized to the maximum (right) when running 1000 functions. Boxes (left) depict the 1/128, 1/64, ..., 1/2, ..., 63/64, 127/128 quantiles.

Figure 12: Cluster evaluation of FaaSwap.

scheduling, and LRU model eviction.

Cluster evaluation. Fig. 12 compares the performance of FaaSwap and three baselines. We first show the ratio of SLO-compliant functions under various number of functions. As shown in Fig. 12a, only FaaSwap can consistently satisfy per-function latency SLOs under a large number of functions, e.g., over 1000. In particular, Native can easily saturate all GPU memory and only supports up to 500 functions, which leads to low GPU utilization. Compared with Native, NonSwap can relax the constraint of GPU memory and enables more functions. However, it still fixes the binding between functions and GPUs, which can cause a number of GPUs overloaded by requests and lead to long tail latency. For example, the ratio of SLO-compliant functions under NonSwap dramatically drops from 800 functions. While SimpleSwap can outperform NonSwap with model swapping, it still suffers from severe SLO violations under 1000 functions. Since model swapping of SimpleSwap is inefficient with high overhead, it can result in long end-to-end latency.

Fig. 12b further compares the behaviors of FaaSwap, SimpleSwap, and NonSwap under 1000 functions. We show the per-request latency normalized to corresponding deadlines (left). In FaaSwap almost every request can be served within its deadline, leading to a normalized latency less than 1. However, both SimpleSwap and NonSwap suffer from long tail latency, which can be over $4\times$ and $7\times$ of the deadline, respectively. We also compare per-worker GPU load of the three solutions. For each node, we normalize loads of its four GPUs to the maximum, and calculate the variance. Lower variance

indicates better load balancing. Fig. 12b (right) plots per-worker load variance under FaaSwap, SimpleSwap and Non-Swap, where there are 6 workers in each one. Compared with NonSwap, FaaSwap and SimpleSwap can effectively balance GPU load across workers with model swapping, achieving much less load variance.

8 Related Work and Discussion

Host-to-GPU data swapping. There have been a number of systems proposed for host-to-GPU data swapping [17, 19, 25, 27, 28, 31, 37, 44, 50]. In particular, host-to-GPU model swapping has been explored in several inference systems, such as Clockwork [25] and DeepPlan [28]. However, they require model-specific knowledge to perform model swapping and share the execution runtime across models, failing to meet the key requirements of serverless inference (§3). Other systems focus on executing deep learning jobs with large memory footprints exceeding GPU capacity, such as vDNN [37], Salus [50], and SwapAdvisor [27], or optimizing data access for general-purpose workloads, such as Batch-aware [31], HUVM [19]. Compared with FaaSwap, they do not consider the semantics of model inference and are not designed to meet its performance requirements, i.e., latency SLOs.

GPU remoting. GPU remoting and API redirection techniques have been employed in different layers to achieve GPU virtualization [13, 36, 47]. Existing solutions like rCUDA [13] and AvA [47] primarily focus on general-purpose workloads. In contrast, FaaSwap applies GPU remoting in the context of serverless inference, leveraging its unique characteristics to enable low-latency API redirection.

Spatio-temporal GPU partitioning. Existing techniques have investigated spatial and temporal partitioning of GPU resources to improve overall utilization [2, 10–12, 20]. These techniques are orthogonal to FaaSwap and can be directly applied, where a physical GPU can be partitioned into multiple virtual instances to enable more target executors in function late binding. This approach further improves GPU efficiency, which we leave as a future work.

Large models. To reduce the swapping overhead of large models, we can cache their partial parameters in GPU memory, which reduces transmission latency at the cost of increased memory footprint. Our future work aims to balance the tradeoff between performance and GPU memory cost. In addition, large language models have gained much popularity and can exceed the memory capacity of a single GPU, necessitating careful designs of model parallelism and pipeline [18, 41, 46]. Supporting these models in serverless cloud can be extremely challenging, which we leave as a future work.

Model swapping from local disk. For functions with extremely low request rates (e.g., a few requests per hour in

14

Fig. 2), keeping many models in host may even saturate the memory and still lead to resource inefficiency. Therefore, the platform can further move those models to local disk, trading the swapping performance for lower keep-alive cost, which we leave for a future work.

9 Conclusion

We present FaaSwap, a GPU-enabled serverless platform for SLO-aware, resource-efficient ML model inference. FaaSwap employs late binding, which maintains models in main memory and dynamically swaps them to a pool of GPUs for execution. This approach effectively enables the pay-per-GPU-use billing and the GPU resource efficiency, without detailed model knowledge. We also design FaaSwap to enable low-latency GPU pooling and model swapping, and propose various scheduling and model management policies to meet latency SLOs for inference functions at low cost. We have implemented FaaSwap atop a leading commercial serverless platform. The evaluation shows that FaaSwap can effectively comply with per-function latency SLOs and improve GPU efficiency.

References

[1] Alibaba Cloud Function Compute. `https://www.alibabacloud.com/product/function-compute`.

[2] Aliyun cGPU. `https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/cgpu-overview`.

[3] Aliyun Function Compute Billing Scheme. `https://www.alibabacloud.com/help/en/function-compute/latest/billing-billing`.

[4] Aliyun Function Compute Instance Types and Modes. `https://www.alibabacloud.com/help/en/function-compute/latest/instance-types-and-instance-modes`.

[5] Amazon SageMaker. `https://aws.amazon.com/sagemaker/`.

[6] AWS Lambda. `https://aws.amazon.com/lambda/`.

[7] AWS Lambda Provisioned Concurrency. `https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html`.

[8] Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[9] Memory Management on Modern GPU Architectures. `https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf`.

[10] Nvidia Multi-Instance GPU. `https://www.nvidia.com/en-us/technologies/multi-instance-gpu/`.

[11] Nvidia Multi-Process Service. `https://docs.nvidia.com/deploy/mps/`.

[12] Nvidia Virtual GPU. `https://www.nvidia.com/en-us/data-center/virtual-solutions/`.

[13] rCUDA. `http://www.rcuda.net/`.

[14] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. USENIX NSDI*, 2020.

[15] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. ACM/IEEE Supercomputing*, 2020.

[16] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proc. ACM SC*, 2017.

[17] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.

[18] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[19] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Myeongjae Jeon, Youngjin Kwon, Rachata Ausavarungnirun, and Jeongseob Ahn. Memory harvesting in multi-GPU systems with hierarchical unified virtual memory. In *Proc. USENIX ATC*, 2022.

[20] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *Proc. USENIX ATC*, 2022.

[21] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*, 2017.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proc. ACM ASPLOS*, 2022.

[24] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *Proc. IEEE IPDPS*, 2022.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.

[26] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.

[27] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proc. ACM ASPLOS*, 2020.

[28] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proc. ACM EuroSys*, 2023.

[29] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. ACM ASPLOS*, 2021.

[30] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proc. USENIX OSDI*, 2020.

[31] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in GPUs for irregular workloads. In *Proc. ACM ASPLOS*, 2020.

[32] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[33] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.

[34] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proc. USENIX OSDI*, 2018.

[35] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *Proc. ACM ASPLOS*, 2020.

[36] C. Reaño, A. J. Peña, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *Proc. IEEE HiPC*, 2012.

[37] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. ACM/IEEE MICRO*, 2016.

[38] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proc. USENIX ATC*, 2021.

[39] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. USENIX ATC*, 2020.

[40] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proc. ACM SOSP*, 2019.

[41] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

[42] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proc. ACM SoCC*, 2022.

[43] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.

[44] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[45] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*, 2022.

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *Proc. USENIX OSDI*, 2022.

[47] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated virtualization of accelerators. In *Proc. ACM ASPLOS*, 2020.

[48] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proc. USENIX NSDI*, 2023.

[49] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.

[50] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[51] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[52] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*, 2021.

[53] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *Proc. USENIX NSDI*, 2023.

Table 5: Primary CUDA APIs supported in FaaSwap, which we divide into asynchronous and synchronous APIs according to their semantics.

CUDA library	API
CUDA Runtime API (Async.)	cudaMemcpyAsync
	cudaMemsetAsync
	cudaLaunchKernel
	cudaFree
CUDA Runtime API (Sync.)	cudaMalloc
	cudaMemcpy
	cudaStreamCreate
	cudaStreamCreateWithFlags
	cudaStreamCreateWithPriority
	cudaStreamSynchronize
	cudaEventCreateWithFlags
	cudaEventQuery
	cudaGetDeviceCount
	cudaGetDeviceProperties
	cudaDeviceSynchronize
cuDNN (Async.)	cudnnSetStream
	cudnnCreateFilterDescriptor
	cudnnSetFilterNdDescriptor
	cudnnDestroyFilterDescriptor
	cudnnCreateConvolutionDescriptor
	cudnnSetConvolutionGroupCount
	cudnnSetConvolutionNdDescriptor
	cudnnSetConvolutionMathType
	cudnnDestroyConvolutionDescriptor
	cudnnCreateTensorDescriptor
	cudnnSetTensorNdDescriptor
	cudnnDestroyTensorDescriptor
	cudnnConvolutionForward
	cudnnBatchNormalizationForwardInference
cuDNN (Sync.)	cudnnCreate
	cudnnGetConvolutionForwardAlgorithm_v7
cuBLAS (Async.)	cublasSetStream
	cublasSetMathMode
	cublasSgemm
	cublasSgemmStridedBatched
cuBLAS (Sync.)	cublasCreate
	cublasGetMathMode

A Appendix

A.1 CUDA API

FaaSwap performs asynchronous CUDA API redirection to reduce communication overhead for efficient GPU remoting (see §4.2). We divide CUDA APIs into two categories, i.e., asynchronous and synchronous APIs, according to whether they require GPU-to-host data transfer and update state in host. Table 5 lists the primary CUDA APIs supported in FaaSwap and their categories. CUDA APIs issued by intermediate steps during model inference are generally asynchronous.

In addition to listed APIs, model inference can also trigger a few other CUDA APIs in our experiments, e.g., cudaGetDevice. These APIs do not affect inference execution and thus FaaSwap can cache their results in GPU clients without repeatedly querying the executor, which further reduces communications.

Algorithm 2 α Auto-configuration.

1: *scalar* — scale factor larger than 1 that controls the rate of α change
2: *threshold* — threshold to trigger α change
3: **function** PERIODICCONFIG
4:     $\alpha \leftarrow \alpha$ in last period
5:     *last_ratio* $\leftarrow$ ratio of SLO-compliant functions in the last period
6:     *new_ratio* $\leftarrow$ ratio of SLO-compliant functions in this period
7:     **if** *new_ratio* $-$ *last_ratio* $> |threshold|$ **then**
8:         $\alpha \leftarrow \min(\alpha \cdot scalar, 1)$ ▷ Increase α
9:     **else if** *new_ratio* $-$ *last_ratio* $< -|threshold|$ **then**
10:         $\alpha \leftarrow \alpha / scalar$ ▷ Decrease α
11:     **else**
12:         $\alpha \leftarrow \alpha$ ▷ Keep α unchanged

A.2 Auto-configuration in Request Queueing

FaaSwap can automatically configure α based on overall load such as to maximize the number of SLO-compliant functions per node (see §5.2). Intuitively, when the load is low and an increasing number of functions can satisfy SLOs, α should grow to prioritize more functions to enable more SLO-compliant functions. On the contrary, FaaSwap should be conservative to prevent functions to violate their SLOs by decreasing α when a node is overloaded. Therefore, we propose an auto-configuration algorithm for α, which is inspired by TCP congestion control. Algorithm 2 shows the pseudo code, where *scalar* and *threshold* are two parameters to determine how much and when α should change. We by default set *scalar* to 2 and *threshold* to 0.04, which is able to properly adjust α according to our profiling.