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 [‡]Alibaba Group [§]Nokia Bell Labs

Abstract

The dynamic request patterns of machine learning (ML) inference workloads have driven an increasing trend towards exploiting serverless computing for scalable ML model serving. However, today's serverless platforms lack efficient support for GPUs — provisioning functions on GPUs incurs extremely high overhead, forcing them to keep long-running even when idling for reduced cold starts. This leads to significant resource waste to perform ML inference and hinders the pay-per-use billing for GPUs.

In this paper, we present FaaSwap, a serverless platform enabling fine-grained, request-level GPU sharing for resource-efficient ML inference. FaaSwap leverages model swapping to support fast inference execution at low resource cost. It keeps models in a host which has a large amount of cheap memory and quickly swaps models to GPUs when requested, reducing per-function keep-alive cost and enabling efficient GPU sharing across much more functions. FaaSwap also supports swapping models between GPUs for load balancing and improved inference performance. In FaaSwap, we design sophisticated request scheduling and memory management algorithms that efficiently exploit model swapping to reduce GPU cost and meet latency service-level objectives (SLOs) for all inference functions. We have implemented and integrated FaaSwap into Alibaba Cloud Function Compute (FC), one of the world's largest commercial serverless platform. Evaluation results show that FaaSwap can achieve low-latency model swapping, efficiently share a GPU across hundreds of functions, and satisfy per-function latency SLOs at scale.

1 Introduction

Machine learning (ML) models have been increasingly deployed in the cloud to deliver ML inference services and boost real-world applications [17, 22, 32, 40, 41]. Model inference is typically performed in real-time under dynamic, bursty request arrival patterns, and thus needs to accommodate changing demands. Serverless computing offers a compelling approach to enabling scalable model inference: users can simply package models as stateless functions, let cloud providers handle resource provisioning and autoscaling, and be charged by per-request resource usage at a fine granularity (e.g., 1 ms [5]). Mainstream serverless platforms, such as AWS Lambda [5], Azure Functions [7] and Alibaba Cloud Function Compute [1], have reported model inference as a popular use case.

However, today's serverless platforms lack efficient support for GPUs, thus exposing a hard tradeoff between inference performance and cost. Model inference typically has stringent request-level latency service-level objectives (SLOs) such as tens of milliseconds [32, 40, 41], while starting an inference function on a GPU can take a few or tens of seconds (Table 1). Therefore, one has to keep functions alive in GPUs for a long duration, or even use provisioned instances [4, 6], to avoid cold starts and meet strict latency requirements, the practice of which is costly and violates the pay-per-use billing of serverless computing. In addition, inference requests can be highly dynamic [31, 40], therefore the long-running functions are often idle, leading to significant waste of expensive GPU resources.

An ideal serverless platform should allow fine-grained, efficient GPU sharing to achieve *cost-effectiveness for both cloud users and providers*, while meeting latency SLOs for inference functions, i.e., *being SLO-aware*. For cloud users, desired GPU functions should follow a pay-per-use billing model without incurring high overhead to serve inference requests and, if needed, resume from idling. This ensures low-latency, SLO-aware inference, and allows users to easily reap economic benefits under dynamic request patterns. For cloud providers, GPUs are much more expensive than CPUs and should be efficiently shared across functions to improve resource efficiency and reduce overall inference cost.

In this paper, we propose to leverage *model swapping* to enable fine-grained, request-level GPU sharing and efficient model inference. Our key insight is to keep inference functions alive in host, and only swap their models to GPUs when they are activated to serve arriving requests; a GPU is shared by multiple functions across requests. This incurs no GPU memory footprint when functions are idle, which in turns enables pay-per-use billing for GPUs (i.e., cost-effective for users). As host memory is much larger than GPU memory, it substantially increases the number of functions each GPU can accommodate, leading to efficient GPU sharing and improved GPU utilization (i.e., cost-effective for providers). Swapping models between host and GPU can be performed efficiently through PCIe, and thus leads to much lower latency than function cold starts and can be easier to meet request-level latency SLOs (i.e., SLO-aware). In addition, whenever desired, we also swap models between GPUs via fast NVLink connections for lower latency and load balancing.

However, enabling model swapping in serverless platforms poses both systematic and algorithmic challenges. First, a serverless platform must efficiently perform model swapping and make it transparent to users, who should not be required to write "swapping logic" in their inference functions and should be unaware of underlying swapping actions. Since model swapping enables fine-grained GPU sharing, the platform also needs to ensure proper isolation across multiple functions. Second, model swapping can incur considerable PCIe traffic and cause bandwidth contention during concurrent inference executions of multiple functions, which leads to increased end-to-end latency. Hence, the platform must design efficient request scheduling and model management algorithms to exploit model swapping such as to meet latency SLOs for all inference functions at low resource cost.

To address these challenges, we present `FaaSwap`, a GPU-enabled serverless platform with efficient model swapping. `FaaSwap` adopts an architecture of GPU pooling, where each worker manages a pool of local GPUs and lets its functions access this pool for inference execution (e.g., via CUDA API redirection), such that model swapping can be easily performed in a GPU pool and be transparent to users. Specifically, to solve the aforementioned systematic challenges, `FaaSwap` proposes three key designs that exploit the characteristics of inference to deliver low-latency model swapping and execution. **First**, it proposes asynchronous API redirection to avoid frequent synchronizations between the functions and the GPU pool, which effectively eliminates high communication overhead for model inference. **Second**, `FaaSwap` leverages pipeline execution to overlap model swapping and inference execution, which hides the latency of model swapping and results in reduced end-to-end latency. It also leverages high-speed NVLink between GPUs for fast model swapping whenever possible. Together with the low-latency API redirection, `FaaSwap` can efficiently execute models on any idle GPU. **Third**, `FaaSwap` designs an efficient GPU memory management system to facilitate model swapping and inference execution. It automatically tracks the addresses of models when they get swapped even across multiple GPUs, and easily adjusts each memory access of CUDA APIs accordingly during inference execution. It also effectively organizes and shares memory blocks to avoid high memory allocation overhead, improving overall performance of model swapping. In addition, `FaaSwap` ensures resource and fault isolation in its GPU pool.

To further address the aforementioned algorithmic challenge to meet latency SLOs for all inference functions at low GPU cost, we propose three policies: **First**, `FaaSwap` designs a request scheduling algorithm to reduce model swapping overhead, leading to low end-to-end inference latency. It divides models into two categories, i.e., heavy or light, according to whether they cause high overhead of swapping through PCIe. In request scheduling, `FaaSwap` prioritizes NVLink over PCIe to transmit heavy models across GPUs, and effectively reduces interference caused by concurrent model swapping. **Second**, `FaaSwap` also exploits model heaviness to guide the eviction when GPU memory is insufficient. It tends to cache heavy models in GPUs and evicts light ones; together with request scheduling, it can substantially minimize swapping overhead. **Third**, `FaaSwap` proposes a SLO-aware request queueing policy, which prioritizes requests to functions that have higher chance to meet SLOs and thus effectively improves the total number of SLO-compliant functions.

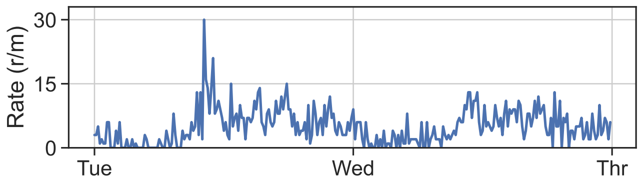


Figure 1: A two-day request trace of a typical GPU inference function in FC.

We have implemented and evaluated `FaaSwap` atop Alibaba Cloud Function Compute (FC) [1], one of the world's largest commercial serverless platforms. Evaluation results show that `FaaSwap` achieves low-latency model inference and swapping in its GPU pool, which leads to comparable performance with native execution. `FaaSwap` can share a GPU across hundreds of functions and load-balance GPUs with model swapping, resulting in over 10× cost reduction compared with current GPU offering in FC. With its efficient policies, `FaaSwap` can enable serving 480 functions at a single 4-GPU worker while achieving low tail latency and satisfying millisecond-scale SLOs for all functions. Cluster experiments further show that `FaaSwap` can effectively scale with function numbers at low resource cost, which meets per-function latency SLOs for thousands of functions using 6 GPU workers.

2 Background and Motivation

In this section, we motivate the need of having a GPU-enabled serverless platform for high-performance inference and identify three key requirements in this regard. We also discuss the inefficiency of existing solutions.

2.1 Serverless Inference and the Need of GPU

As a prominent serverless platform with a global presence, FC has observed a growing adoption among enterprise customers who choose to provision their inference services using serverless functions, known as "*serverless inference*." In this approach, users package models and inference code into containers and publish them as serverless functions, which can be dynamically invoked to make predictions. With serverless inference, users are relieved of the burden of server management as it is automatically handled by FC, such as provisioning, autoscaling, scheduling, and fault tolerance. Serverless inference can also enable significant cost savings as users do not pay for idle resources under the pay-per-use pricing model [12, 36, 39, 40]. In FC, a function's requests typically exhibit dynamic, bursty arrival patterns as shown in Fig. 1, consistent with previous research findings [17, 18, 22, 23, 28–30, 32, 42]. Leveraging the high elasticity of serverless computing, inference functions can quickly scale in response to the changing workload, while users are billed based on the function runtime, with billing granularity as fine as 1 ms [5, 7].

However, both FC and other leading serverless platforms currently lack efficient support for GPUs, impeding their ability to achieve high-performance inference. In fact, numerous FC users have expressed a compelling need to execute their models on GPU-enabled functions, indicating the strong market demand for GPU-accelerated inference in current FaaS platforms.

2.2 Key Requirements

Based on our operational experiences and interactions with FC customers, we have identified three key requirements for building an efficient GPU-enabled serverless inference platform.

Compliance to latency SLOs. Enterprise users often have stringent latency requirements for online inference, which is the key driver behind their demand for GPU support in FC. Therefore, our platform should allow users to specify their latency requirements as SLOs, such as ensuring that at least 99% of inference requests are served within 200 ms [40]. The platform should strive to meet the latency SLOs for all functions, if possible.

Pay-per-GPU-use. Compared to the traditional "serverful" approach, one of the key advantages of serverless computing is its pay-per-use billing model. Users hence requires that their inference functions are billed based on the actual GPU usage, with charges incurred only when the functions are invoked and running on GPUs (pay-per-GPU-use).[1] This is crucial for achieving substantial cost savings in the presence of dynamic inference workloads (Fig. 1), considering the high cost of GPUs.

GPU-efficient inference. For serverless providers like FC, minimizing the resource provisioning cost is the key to maintaining market competitiveness. Given the significantly higher cost of GPUs compared to other resources, 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 fine-grained and efficient GPU sharing.

[1]Note that in our experience, 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 overhead [3, 6, 25].

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

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

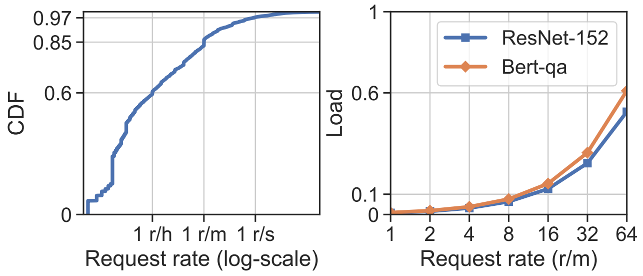


Figure 2: CDF of function average request rate from one-week production trace (left) and expected GPU load under various per-function request rates when running multiple functions on a `V100` GPU to saturate its 32 GB memory (right).

2.3 Existing Solutions and Their Inefficiency

Inefficiency of existing solutions. Achieving the three requirements presents non-trivial challenges. Compared to CPU functions, running inference functions on GPUs incurs considerable startup overhead. Table 1 provides a comparison of model execution times when the inference functions are warm- and cold-started on V100 GPUs in FC.[2] Cold-start results in a two orders of magnitude slowdown due to the need for GPU container setup, ML framework startup (PyTorch in our case), GPU runtime creation, and model loading. This leads to extremely long latency that far exceeds the SLO requirement of model inference.

To avoid cold-start, a common approach is to maintain provisioned functions that remain active on GPUs [4, 6]. However, this approach deviates from the serverless paradigm and is costly for both cloud users and providers. **First**, as provisioned functions, even when idling, occupy GPUs for extended duration, 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**, it results in severe GPU underutilization, considering that the majority of functions exhibit low to medium request rates. Fig. 2 (left) depicts the distribution of the average re-

[2]For cold-start, we exclude the delay of fetching a remote container image or model file, which can take extra seconds to minutes to complete [35].

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

Solution	SLO-aware	Pay-per-GPU-use	GPU-efficient
FC [1]	×	×	×
Molecule [20]	×	–	×
DGSF [21]	×	–	×
INFless [36]	✓	–	×
FaaSwap	✓	✓	✓

quest rates of FC functions in a one-week trace, revealing that 85% (97%) of functions were invoked only once per minute(second) on average[3]. These findings align with observations from other production traces [7, 31].

Table 2 provides a comparison of existing solutions that offer GPU support in serverless platforms and our system, FaaSwap. Alibaba Cloud Function Compute (FC) [1], as a prominent commercial serverless platform, fails to meet latency SLOs and achieve resource efficiency for GPU functions. Molecule [20] introduces a serverless platform that supports GPUs and other hardware devices, while DGSF [21] enables serverless functions to access GPUs in a remote cluster. However, both works primarily target general-purpose workloads and suffer from GPU inefficiency. INFless [36] presents a serverless system specifically designed for model inference with GPU function support. Although it aims to minimize inference latency, it still results in GPU idling and cost inefficiency.

Request-level GPU sharing and its limitations. To enhance resource efficiency, it is intuitive to implement finer-grained GPU sharing, whereby multiple functions can be consolidated onto a single GPU. Each function exclusively utilizes the GPU when activated and relinquishes it upon completion, allowing other functions to execute requests. This approach can potentially improve overall GPU utilization but has two limitations.

First, it still requires to cache a large number of idle inference functions, which ultimately saturates GPU memory and compromises resource efficiency. Fig. 2 (right) illustrates the expected GPU load, measured as the proportion of the GPU busy period, under varying per-function request rates when multiple functions are running on a single V100 GPU that fully occupies its 32 GB memory capacity. A higher GPU load signifies improved utilization. However, the GPU load consistently remains low due to the limited GPU memory. Even when the request rate per function exceeds 1 r/s (the 97^{th} percentile in Fig 2 left), the GPU load still remains below 60%.

Second, packing multiple functions into a GPU can make it overloaded for a short period due to the bursty request patterns. In a multi-GPU machine, this can inevitably result in hot spots and load imbalance across GPUs, which cannot meet request-level latency SLOs nor achieve high GPU utilization. The impact of load imbalance is demonstrated in Fig. 7, with details given in §7.1.

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

3 Key Insight and FaaSwap Overview

We next discuss our solution to aforementioned limitations.

Key insight. As described in §2.3, existing solutions have to keep inference function alive in expensive, limited GPU memory, which leads to not only high function idling cost but also GPU underutilization even under fine-grained resource sharing. Therefore, to enable efficient request-level GPU sharing, a serverless platform must support fast inference execution and low function keep-alive cost, without incurring GPU memory footprint when idling.

We propose to leverage *model swapping* for low-latency, resource-efficient serverless inference. We keep functions alive by caching the models in host memory, and swap models into GPUs only when requested. Since host memory can be cheaper with much larger amount than GPU memory (e.g., a few TB vs. tens of GB), our solution not only avoids charging users for GPU resources during function idling (i.e., pay-per-GPU-use), but also significantly increases the number of functions each GPU can serve, thereby improving overall resource efficiency. Moreover, model swapping can be efficiently performed through PCIe and incurs much less overhead than function cold starts, which can sustain low inference latency and be easier to satisfy request-level millisecond-scale SLOs. We also swap models between GPUs via high-speed NVLinks, which further improve swapping performance and effectively mitigates load imbalance across GPUs.

Challenges and overview. Following this insight, we present FaaSwap, a serverless platform to enable model swapping for low-latency inference and GPU resource efficiency. Achieving so in FaaSwap can pose both systematic and algorithmic challenges.

First, it is non-trivial to enable efficient model swapping and ensure isolation in serverless platforms. In serverless paradigm, users only deliver their inference functions, while the platform holds no knowledge of their models, e.g., model structure and parameters. This requires the platform to automatically track memory footprint of each function, efficiently transmit models, and make it transparent to users. That is, users are not required to write "swapping logic" in their inference code and should be unaware of swapping actions taken by the platform. Since model swapping enables fine-grained GPU sharing across a large number of functions, the platform should also carefully protect in-memory models across various functions and ensure the isolation.

Second, it remains challenges to efficiently perform request scheduling such as to exploit model swapping to satisfy latency SLOs for all functions while achieving high resource utilization. Unlike existing serverless platforms [34, 36, 38], model swapping allows a function instance to run on various GPUs across requests, requiring the platform to carefully design scheduling and memory management policies around GPUs. In addition, model swapping can incur bandwidth contention across functions during concurrent model swapping through PCIe, which impacts end-to-end latency and request-level SLOs. Therefore, the platform needs to judiciously design request scheduling, model swapping, and GPU memory management policies to meet the properties in Table 2.

Therefore, FaaSwap should first enable efficient model swapping in serverless platforms, and then design effective algorithms to exploit it for SLO-aware inference and resource efficiency. We will next discuss how FaaSwap addresses systematic challenges in §4, and defer its algorithm design to §5.

4 FaaSwap System Design

In this section, we present the system design of FaaSwap.

4.1 Architecture overview

FaaSwap adopts an architecture of *GPU pooling* to enable efficient GPU sharing and effectively make model swapping transparent to users. FaaSwap runs a GPU server at each worker node to manage all its local GPUs as a pool, such that functions can directly interact with the GPU server to access any GPU to facilitate resource sharing. In addition, the GPU server holds models of all local inference functions and can easily perform model swapping, without needing functions to realize.

Fig. 3 shows the architecture overview of FaaSwap. FaaSwap has two components, cluster manager and worker nodes. The cluster manager takes charge of cluster-level tasks, including request routing, node allocation, and resource scaling. Each worker node hosts a number of functions, runs a GPU server, and uses an intra-node router to control requests to local function instances. The GPU server uses its model repo to manage models in host memory, and runs an executor for each GPU in the pool, which handles CUDA calls, swaps required models, and manages memory accordingly . The server also has a controller that holds a global view of GPU memory and executor status and can determine how to schedule requests to GPUs (executors). Once a request arrives at the target function, it interacts with the scheduled executor using a GPU client and remotes CUDA API calls during inference execution. GPU server, router, and functions in the worker node run as containers.

Key to FaaSwap is to design and build an efficient GPU server that enables fast, resource-efficient inference and model swapping. This poses four challenges: (1) how FaaSwap enables efficient GPU remoting (§4.2); (2) how FaaSwap obtains the knowledge of model and achieves low-latency model swapping (§4.3); (3) how FaaSwap efficiently manages GPU memory (§4.4); and (4) how FaaSwap ensures the isolation and handles failures (§4.5). We next elaborate the designs of FaaSwap to address these challenges.

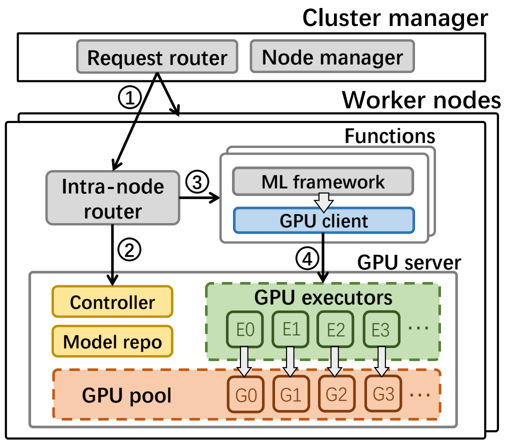


Figure 3: Architecture overview of FaaSwap. A request arriving in FaaSwap cluster is first routed to the worker node hosting its target function ①. The router in the node synchronizes with the GPU server to query the executor for this request ②, and then routes it to the function instance with target 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 a result to the user after request completion.

4.2 GPU Remoting

GPU remoting is fundamental to pooling, and we describe how FaaSwap enables GPU remoting and addresses the challenges therein.

CUDA API redirection. FaaSwap enables GPU remoting by redirecting CUDA API calls from function instances to GPU executors. Each function instance runs a GPU client that can intercept CUDA APIs from ML frameworks, e.g., PyTorch inference programs, and remotes them to GPU executors for CUDA execution. This allows a function instance to access various GPUs at a request granularity: after scheduling a request (② in Fig. 3), the GPU client can redirect all its CUDA calls to the target executor (④); following requests to this function can be scheduled to other executors, and the client varies the target accordingly. Hence we can effectively migrate load between GPUs with the support of model swapping (§4.3).

However, CUDA API redirection can incur significant synchronization overhead compared with native execution, which dramatically slows down model inference. According to our measurement, it can need thousands of CUDA API calls in an inference execution, e.g., over 4k calls for ResNet-152, and thus a large number of synchronizations between a function and a GPU executor cause a substantial delay, e.g., hundreds of milliseconds (Table 4), violating request-level

SLOs. FaaSwap avoids such synchronization overhead via asynchronous API redirection.

Asynchronous redirection. We observe that intermediate steps in an inference execution are typically performed asynchronously in GPU — the intermediate data get generated and consumed on GPU memory without requiring any data transfer to the host, until the execution is completed and the host receives an output result. 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 ensure the correctness of model inference.

Following this insight, we perform asynchronous redirection for CUDA APIs that can be executed asynchronously. In particular, we divide the set of CUDA APIs into two categories based on their semantics: synchronous, blocking APIs and asynchronous, non-blocking APIs. The former needs the host to wait for their completion and use the outputs in following steps, e.g., `cudaMalloc`, and thus we by default perform synchronizations. The latter do not change the runtime state in the host, e.g., `cudaLaunchKernel`, allowing asynchronous API redirection without blocking. FaaSwap supports common CUDA runtime APIs and CUDA libraries, e.g., cuDNN, and we show the category of each API in Appendix A.1.

With asynchronous API redirection, we can fuse multiple consecutive API calls into a single group and send them together. Such group-level API redirection can further reduce communications, but it requires an effective grouping strategy. Fusing too many calls into one group, e.g., all intermediate API calls, can greatly eliminate communications between the function and executor, which however needs functions to wait until all calls are issued. On the other hand, having too few calls in each group, e.g., one call per group, incurs no extra delay but can need a large number of communications, i.e., frequent API redirections. Therefore, we conduct extensive profiling and choose a "good" group size, which can balance the two factors for overall high redirection performance.

We show the performance advantages of asynchronous, group-level API redirection in Table 4 (§7). Compared with synchronous API redirection, FaaSwap can cut the inference latency of popular models by up to an order of magnitude due to significantly reduced communications. Surprisingly, FaaSwap can even outperform the native execution, i.e., using local GPU without GPU remoting, for evaluated CNN models. The performance gain is owed to parallel execution, as many asynchronous CUDA APIs in these models require only CPU, such that API redirection in fact distributes CPU-side workloads across both functions and executors. For Bert-qa that requires no CPU-side CUDA APIs, FaaSwap still leads to comparable performance with native execution, indicating a negligible overhead of asynchronous API redirection.

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 second. FaaSwap also preloads all CUDA kernels on each GPU to avoid loading overhead. We will discuss isolation of FaaSwap in §4.5.

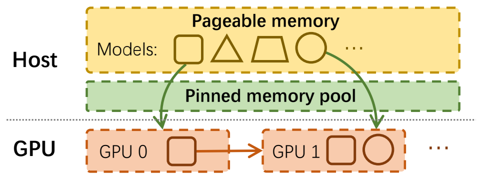


Figure 4: An example of model swapping. Models can be swapped from host to GPU through PCIe (green arrows), and across GPUs through NVLink (red arrow).

4.3 Model Swapping

We next describe how FaaSwap manages and efficiently swaps models in its GPU server.

Model management. FaaSwap can automatically track model knowledge during function cold starts. We note that the model access pattern typically remains the same across requests, e.g., access order of parameters, and thus can be easily obtained without needing user input. In particular, when a function instance starts loading the model, a GPU executor receives relevant CUDA API calls that contain its parameters, and tracks every GPU memory access for its first run. FaaSwap leverages access order of parameters to guide future model swapping, and also keeps a model copy in host memory. Both model copies and access patterns are maintained in the model repo (Fig. 3) until associated function instances are terminated.

Model swapping. With model knowledge, FaaSwap can perform model swapping at request level. The GPU server schedules a request to an executor (② in Fig. 3), which can trigger model swapping if the requested model is not loaded on the target GPU. Together with CUDA API redirection (§4.2), FaaSwap can easily execute a model on various GPUs across requests. Fig. 4 gives an example of FaaSwap's model swapping. Host-to-GPU model swapping is performed through PCIe, and requires a model to be pinned at first to enable DMA transfers, which triggers additional memory copy. FaaSwap therefore shares a pinned memory pool across models to reduce memory copy with only small pinned memory footprint. FaaSwap also supports GPU-to-GPU model swapping when a GPU is too busy to serve its models. It can swap a model to other idle GPUs via fast NVLinks, and schedule its requests to corresponding GPU executors, which ensures fast function switching across GPUs for load balancing, and achieves low request latency. FaaSwap performs model swapping when

GPU memory is insufficient. Like cache eviction, we simply invalidate GPU memory region of a model without needing to swap it back to host, which holds a copy for each model. This avoids the swapping overhead with only an insignificant cost. We defer detailed model swapping and eviction policies, e.g., when and where to swap models, to §5.

However, model swapping can pose two challenges. **First**, it requires FaaSwap to carefully manage and translate GPU memory addresses. As functions are agnostic to the underlying model swapping, model addresses of their CUDA calls remain the same across various requests, even when they run on different GPUs. Therefore, FaaSwap automatically tracks memory addresses for model swapping and updates relevant memory access in each CUDA API call accordingly, which is made transparent to functions. We defer the details of memory management to §4.4. **Second**, model swapping can incur extra delay to end-to-end inference latency. To improve overall performance, FaaSwap exploits model pipeline to hide the overhead of model swapping, which we describe below.

Optimize model swapping via pipeline. Note that model inference executes only a forward pass and is generally performed layer by layer. This allows us to overlap the transmission of next layers and the computation of previous layers in GPUs, thus enabling pipeline execution [14, 33]. We exploit two characteristics of FaaSwap to design its pipeline execution.

First, FaaSwap's GPU server executes a model and keeps track of its parameters at CUDA API level (§4.2), by which we perform CUDA-level model pipeline. In particular, it loads parameters of the requested model following their access order obtained at the first run; during inference execution, it checks if model parameters required by each CUDA API are loaded in GPU; otherwise it waits until they become ready. In this way, the executor concurrently swaps model parameters and executes CUDA APIs on those loaded to reduce overall latency. Such pipeline execution can apply to both host-to-GPU and GPU-to-GPU model swapping.

Second, CUDA-level model pipeline can require frequent synchronizations between host and GPUs to ensure each CUDA API call get issued only when its data are loaded. FaaSwap reduces such synchronization overhead by group-level model pipeline. It swaps multiple consecutive parameters as a group, and performs synchronization once an entire group is loaded into a GPU. Determining how parameters are grouped poses a tradeoff between the swapping performance and pipeline efficiency: grouping more parameters incurs less synchronizations with reduced swapping overhead, but leads to less overlap between model transmission and computation. FaaSwap's model pipeline needs a "good" group size that can balance the tradeoff and be generally applied to various models. We notice that grouping too many parameters can have little improvement on swapping performance, as synchronizations therein cause only negligible overhead. Therefore, we profile the performance of transmitting various-size data, and choose a knee point as a desired group size, which can achieve good swapping performance without impacting pipeline efficiency too much. Such group size can be directly applied to different models, and only depends on hardware configurations, e.g., PCIe bandwidth.

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

4.4 Memory Management

GPUs have a memory management system that provides similar functionalities with CPUs, e.g., memory allocation, and also supports unified memory to transparently handle data movement between host and GPU memory. However, native GPU memory management is designed for general-purpose workloads and cannot be directly applied to FaaSwap's model swapping. There are mainly two challenges. **First**, in FaaSwap model swapping not only requires data transmission, but also involves CUDA-level pipeline executions that can run on multiple GPUs across requests (§4.3). This requires FaaSwap to hide the memory details across various GPUs and performs fine-grained synchronizations, which is not natively supported by GPUs. **Second**, native memory allocation (e.g., `cudaMalloc`) incurs considerable overhead, while FaaSwap can need frequently model loading and eviction, significantly degrading overall swapping and inference performance. To address the above challenges, we design a GPU memory management system for FaaSwap's model swapping.

Memory address management. When model swapping occurs, either host-to-GPU or GPU-to-GPU, actual memory addresses of model parameters can differ from original ones, which requires FaaSwap to carefully manage memory layout and automatically translate each memory access in CUDA execution. FaaSwap exploits the memory layout of ML frameworks to facilitate address management. ML frameworks, such as PyTorch and TensorFlow, typically organize data into blocks for ease-of-management, where each GPU memory block can contain a number of parameters. This hence allows FaaSwap to perform memory mapping at block level. In particular, FaaSwap tracks memory blocks for each function and maintains a mapping to their actual physical addresses after model swapping. Internel data layout in each block, e.g., the offsets of its parameters, remains the same, such that FaaSwap can easily obtain the physical address of a particular parameter using its belonging block address and corresponding offset. Therefore, FaaSwap needs not to maintain much metadata for individual data pointers, effectively handling address translation without high management overhead.

Memory allocation and block management. FaaSwap needs to allocate and free memory blocks during model swapping and evictions. Using native GPU memory management for block allocation can cause high overhead, e.g., tens to hundreds of milliseconds for a single model (Fig. 9), which impairs overall swapping performance. FaaSwap hence preallocates all GPU memory and internally manage all blocks to avoid using native APIs in block allocation. However, this poses a challenge for block management: the size of blocks and their popularity can vary across models, requiring FaaSwap to efficiently manage blocks and avoid many memory fragments; otherwise it needs to frequently release existing blocks for reallocation and lead to high overhead. Buddy memory allocation [27] is a classic approach to reduce memory fragments, 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 example, PyTorch by default uses fixed-size blocks to host small- and moderate-size data, and thus these block sizes have high popularity across various models, e.g., 20 MB. FaaSwap therefore divides blocks into two categories based on their sizes, i.e., regular fixed-size blocks and others irregular, and manages them separately. In particular, FaaSwap divides all GPU memory into a number of *memory partitions* at bootstrap. Memory partitions are created via native CUDA allocation API and have the same size, each hosting either category of blocks. FaaSwap can perform Buddy-based management in each memory partition to allocate and reclaim blocks. For partitions hosting regular blocks, FaaSwap adopts a revised policy, e.g., directly dividing memory into fixed-size blocks rather than native Buddy allocation, which can further reduce fragments. Once all blocks of a partition are reclaimed, it becomes idle and can be later used by any block category. **Second**, in FaaSwap all blocks of a model are expected to be accessed entirely during model swapping and execution, and also reclaimed together after model eviction. Motivated by this observation, we can package blocks from the same model as tight as possible, e.g., collocating them on a single memory partition, such that model eviction can easily free entire memory partitions and make them available for future block allocation. FaaSwap can also periodically consolidates blocks to reduce fragments.

4.5 Isolation and Fault Handling

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

Resource isolation FaaSwap provides container-level isolation for CPU and memory resources [4], similar to existing serverless platforms. For GPUs, FaaSwap performs software-based isolation at its GPU server, which ensures GPU compute resources, e.g., SM, 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 virtualization (§4.4).

[4]FaaSwap makes no assumption on sandboxes and can also support microVMs [11].

Table 3: Latency (ms) of model pipeline execution with increasing ratio when concurrently swapping other models through PCIe. The diagonal numbers 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**

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

At 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

In this section, we present how FaaSwap achieves desired properties, i.e., SLO-aware and resource-efficient (Table 2), with its policy designs. We start with the design overview, followed by individual policies.

5.1 Design Overview

Objectives. 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 less than a user-specified deadline, and meter resource cost by the number of worker nodes. Key to achieving this goal is to *maximize the number of SLO-compliance 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. Maximizing the number of SLO-compliance functions at nodes can pose three challenges. **First**, model swapping allows FaaSwap to host much more functions on a worker node, the dynamic request patterns of which can cause short load burst, overload its GPUs, and result in request queueing. This requires FaaSwap to carefully determine how requests should be prioritized in the queue such as to meet latency SLOs for as many functions as possible. **Second**, FaaSwap schedules requests to GPUs when they arrive at GPU server (② in Figure 3), which accordingly determines where to swap target models. However, the performance of model swapping can be impaired and hard to predict due to bandwidth contention, which makes it challenging for request scheduling to ensure low-latency inference. For example, Table 3 shows the latency of model pipeline when concurrently swapping other models through PCIe, which leads to diminished performance especially for large models. **Third**, FaaSwap needs to determine how to evict models and which models should be kept in GPUs such as to reduce model swapping and improve overall inference performance. Achieving so in model eviction is non-trivial due to dynamic, hard-to-predict request arrival patterns.

We note that it is fundamentally unable to jointly find the optimal solutions to these challenges, as both future request arrival patterns and swapping performance are unpredictable in our settings. Even with perfect future knowledge and predictable performance, the problem is still NP-hard and do not apply to online inference [42]. We therefore address the challenges separately with heuristic solutions, which we describe below. We summarize how FaaSwap leverages these policies to achieve overall objective in §5.5.

5.2 Request Queueing

At each worker, intra-node router queues and dispatches requests (Figure 3), which aims to satisfy latency SLOs for as many functions as possible. Intuitively, FaaSwap can prioritize functions according to the possiblity to comply with SLOs, such that requests to functions with higher possibilities should get executed first for reduced queueing delay. Following this insight, we can divide all functions at a node into two sets based on whether they can potentially satisfy SLOs, and respectively maintains two queues for their requests, i.e., high- and low-priority. When there are available GPUs, FaaSwap first executes requests from the high-priority queue, and only dispatches low-priority requests if the former is empty. Functions can be moved between the two sets at runtime according to their possibilities to comply with SLOs. However, enabling this needs to address two key challenges: (1) how can we quantify the SLO-compliance possibility for a function, and (2) how shoud we partition functions into high- and low-priority sets.

Metric for function prioritization. A desired metric should capture "how much effort" required to satisfy SLOs, and thus functions with less effort can be easier and have more chance

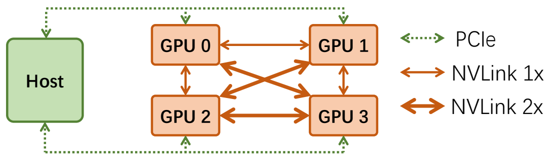


Figure 5: Topology of 4-GPU worker node in FC.

to achieve so. We therefore propose as the metric *required request count* (RRC), which measures the expected number of future requests served within deadlines in order to meet SLOs. Let n be the current number of requests for a function, and m be the number of requests served within deadlines out of total n requests. The RRC of the function can 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: $\frac{m+RRC}{n+RRC} = p$. RRCs of various functions can be normalized by average request latency. Functions with negative RRCs have already satisfied SLOs until now, while the larger a function's RRC is, the lower it has a possibility to make it. This allows us to prioritize requests to functions with smaller RRCs.

Divide functions into two priority sets. With RRCs, we can divide functions into high- and low-priority sets and determine request execution order in each function set. Intuitively, we can put functions with small RRCs (i.e., both small positive and negative) in the high-priority set, and the rest in the low-priority set, which can sustain existing SLO-compliance functions and increase the number whenever possible. Determining the RRC boundary between the two sets can be challenging — having too many (few) high-priority functions can be too aggressive (conservative) to enable more SLO-compliance functions. In FaaSwap we use a threshold $\alpha \in [0,1]$ to indicate the boundary and determine how many functions should be prioritized: we can prioritize more functions by increasing α, and aggressively put all functions in the high-priority set when α is 1. 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 set, where k is the largest integer such that $\sum_{j=1}^{k} \max(RRC_j, 0) \le \alpha \cdot \sum_{i=1}^{N} \max(RRC_i, 0)$. FaaSwap can automatically configure α at runtime based on overall load and function SLOs. When there is a short load surge and the number of SLO-compliance functions decreases, FaaSwap turns to be conservative with a small α; otherwise α can increase to prioritize more functions. We defer the detailed algorithm of α auto-configuration to the Appendix A.2. FaaSwap also periodically adjusts the two set of functions according to their up-to-date RRCs and α.

The request execution order in each priority queue can also be determined with function RRCs. In high-priority queue, requests are prioritized in a reverse order of RRCs, such that functions with small positive RRCs can be favored and easily made SLO-compliance. In contrast, requests in low-priority queue should follow the order of RRCs, as the smaller a low-priority function's RRC is, the more chance it can have to be prioritized and satisfy SLOs.

5.3 Scheduling and Model Swapping

We next describe how FaaSwap schedules requests to GPUs and swaps models at request execution. Once FaaSwap's GPU server receives a request (② in Figure 3), its controller determines which GPU (executor) should load the target model and process this request. Each GPU executes a request at one time to ensure the resource isolation (§4.5). The objective of request scheduling is to minimize per-request inference latency, which however can be challenging due to accompanying model swapping.

Bandwidth contention in model swapping. While the latency of model execution is often stable [22], FaaSwap's model swapping can incur unpredictable overhead due to PCIe bandwidth contention across GPUs [13, 26]. Fig. 5 shows topology of a worker node in FC, 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. We measure the performance of concurrent model pipeline execution on a pair of GPUs, as shown in Table 3. The performance slowdown caused by bandwidth contention can vary in models, and is generally more significant for large models that require more data transmission, e.g., Bert-qa. Having said that, we observe that swapping with light models can lead to reduced contention and lower latency compared with bandwidth-intensive models, e.g., ResNet-152 with DenseNet-169/Bert-qa. Therefore, we propose to leverage this characteristic to reduce interference in model swapping for improved 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 simple model profiling: if model pipeline significantly slows down inference execution, data transmission can be 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 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 swapping overhead (line 8). If the model hosted by busy GPUs, FaaSwap then schedules the request to perform GPU-to-GPU swapping, in that the source and target GPUs should have fastest NVLink connection (line 11). Otherwise, FaaSwap resorts to host-to-GPU swapping and prioritizes target GPUs whose neighbors are idle or running light models to reduce PCIe contention (line 18). In a nutshell, FaaSwap can minimize interference and overhead of model swapping for each request, and thus provides low inference latency.

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
```

5.4 Model Eviction

We next describe FaaSwap's model eviction policy. FaaSwap can cache models in GPUs for future requests, and determine how to evict inactivated models from a GPU when its memory is fully occupied and cannot load others activated. Unlike traditional cache eviction, model eviction in FaaSwap aims to improve overall inference performance by reducing swapping overhead. Swapping performance can directly impact end-to-end latency compared with cache hit rate, and thus model eviction should be made aware of swapping overhead of various models, i.e., heavy or light.

Similar to interference-aware scheduling (§5.3), we perform model eviction to minimize swapping overhead for future requests. We notice that swapping light models lead to negligible overhead for end-to-end performance, while heavy models can causes considerably long latency under pipeline execution due to model transmission and accompanying bandwidth contention (Table 3 and 4). Therefore, we tend to evict models that have almost no impact on performance when swapping, such as to cache more heavy models in GPUs for reduced host-to-GPU data transmission and interference. Following this insight, we divide models into two priority sets: heavy models hosted by only a single GPU should be prioritized, and the rest are low-priority and can be evicted earlier, including light models and heavy ones with multiple copies in various GPUs. As a result, FaaSwap only needs to frequently perform host-to-GPU and cross-GPU swapping for light and heavy models respectively, both of which lead to no PCIe bandwidth contention and thus achieve low swapping overhead. In each priority set, we adopt a common Least-Recently-Used (LRU) policy to determine the eviction order for its models.

5.5 Put All Together

Finally, we describe how FaaSwap achieves desired properties at cluster, i.e, SLO-aware and resource-efficient, with above policies. FaaSwap's cluster manager monitors node-level request load and function SLOs, and can eliminate potential SLO violations by load migration and node scaling. Since FaaSwap aims to maximize the number of SLO-compliance functions at node (§5.2), it can by design discard popular functions (i.e., with high request load) when a node is overloaded. Therefore, FaaSwap can migrate these functions onto other nodes with available resources and provision new nodes when needed, which effectively meets SLOs for all functions at low resource cost.

6 Implementation

We have implemented FaaSwap atop FC. 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 relevant components in FC. We also implement a demo router in 530 lines of Python code to provide 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, which is a common practice in FC. We provide a container image as a function template based on PyTorch, where original CUDA libraries are replaced by GPU clients to enable GPU remoting, e.g., `libcudart.so`. This requires no modification to the PyTorch framework.

7 Evaluation

In this section, we evaluate FaaSwap using production traces from FC. Our evaluation answewrs 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-compliance 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 FC 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.

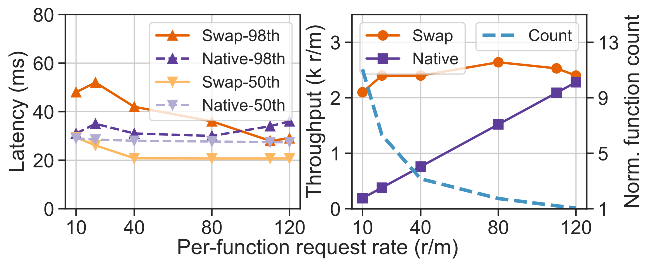


Figure 6: Performance of executing multiple ResNet-152 functions on a single GPU under FaaSwap's model swapping (Swap) and native execution (Native), which packs as many as possible functions on the GPU to saturate its memory. We show the median and 98^{th} tail latencies under various per-function request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

Metrics We focus on the ratio of functions meeting SLOs and GPU load in evaluation. For a function, it complies with SLOs only when its tail request latency is less than a deadline. By default, we use 98^{th} 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. FaaSwap performs host-to-GPU model swapping to reduce GPU memory footprint for high resource efficiency. Fig. 6 compares FaaSwap with native execution (Native), which simply keeps functions in GPUs without swapping and shares a GPU across requests. Native can only host a small number of functions due to limited GPU memory, and thus leads to poor aggregate throughput under low or median request rates. In contrast, FaaSwap can enable much more functions with sufficient host memory, e.g., over 10× under 10 r/m, which significantly improves throughput and leads to high GPU utilization. Only with a high request rate, e.g., 120 r/m per function, FaaSwap and Native have similar throughput. Note that 97% functions are requested less than once per second in FC production cluster (Fig. 2 left). Hence keeping models in host can dramatically improve GPU memory efficiency and effectively support more functions per GPU. In addition, FaaSwap achieves comparable performance with Native due to efficient model swapping. For example, the tail latency under 10 r/m only increases to about 50 ms

Table 4: ML models and the latency (ms) of GPU remoting and model swapping. Bert-qa [19] 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

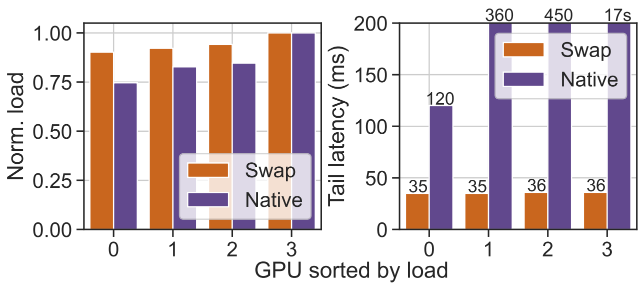


Figure 7: Per-GPU load normalized to the maximum (left) and 98th tail latency of requests on each GPU (right) under FaaSwap's model swapping (Swap) and Native.

with FaaSwap's model swapping, which can effectively meet latency SLOs. In FaaSwap, increasing the request rate can reduce model swapping with fewer models and thus lead to lower latency.

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.

7.2 FaaSwap at Node

We next evaluate the performance of FaaSwap at 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 we believe is representative for GPU inference functions as we discuss in §2.

FaaSwap with its policies. To understand the benefits of FaaSwap's policies, 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. 8 shows the ratio of SLO-compliance 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 in request queueing, which leads 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 significantly harms overall performance as we will describe in Fig. 9. FaaSwap-LRU can easily evict heavy models and cause PCIe bandwidth contention during host-to-GPU swapping, the high overhead of which also degrades the 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 exploits NVLink across GPUs and can cause more serious PCIe bandwidth contention than FaaSwap-LRU.

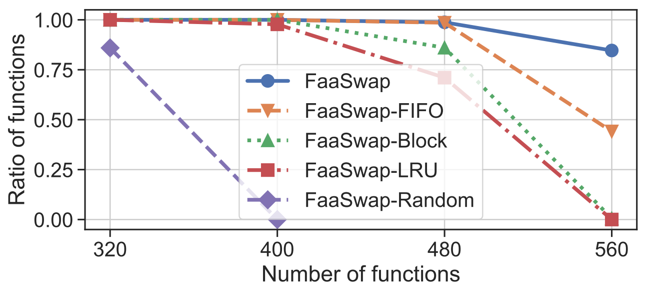


Figure 8: Ratio of SLO-compliance functions under FaaSwap and various policies.

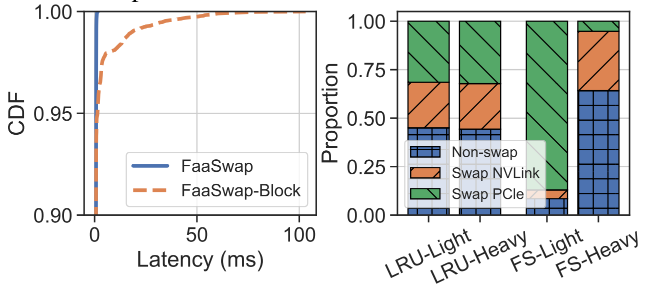


Figure 9: Behaviors of FaaSwap's block management and eviction policies: latency of block allocation under FaaSwap and FaaSwap-Block (left), and proportion of three swapping cases under LRU and FaaSwap's eviction policies (right).

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-compliance functions.

Policy behaviors. Fig. 9 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. 9 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. 9 (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. 10 shows the ratio of SLO-compliance 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 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.

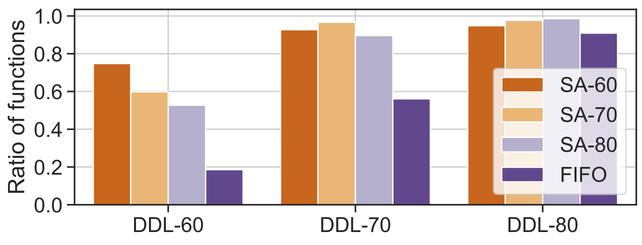


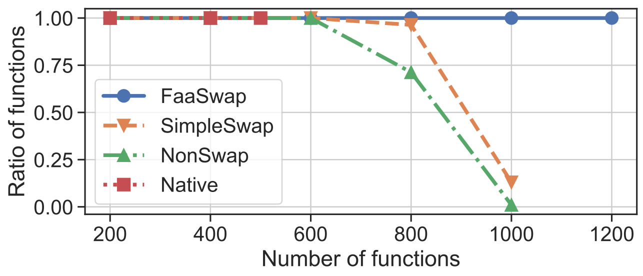
Figure 10: Ratio of SLO-compliance functions using FIFO and FaaSwap's SLO-aware (SA) policies. We set deadlines from 60 ms to 80 ms, and vary the targets in SA accordingly.

7.3 FaaSwap at Cluster

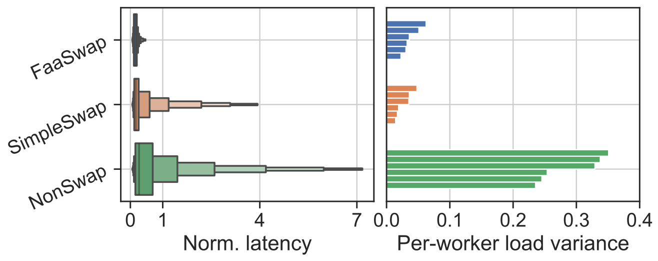
We next evaluate FaaSwap on a cluster deployment with 6 GPU workers. As running FaaSwap on a FC cluster incurs additional system overhead, we relax the SLOs and set deadlines for CV models and Bert-qa to 150 ms and 250 ms, respectively.

Baselines. We compare FaaSwap with three baselines. *(1) Native* uses native GPU containers running on specific GPUs, which is a common practice in FC. Each GPU worker can only host a fixed, small number of functions due to limited GPU memory (§7.1). We perform request-level GPU sharing in Native, where requests to a GPU are sequentially executed in a FIFO manner. *(2) NonSwap* allows GPU remoting similar to FaaSwap, but disables model swapping. Compared with Native, NonSwap can share GPU runtime across functions, and thus reduces memory footprint and enables more models per GPU. *(3) SimpleSwap* enables model swapping compared with NonSwap. Unlike FaaSwap, it only supports simple policies discussed in §7.2, such as FIFO request queueing, random scheduling, and LRU model eviction.

Cluster evaluation. Fig. 11 compares the performance of FaaSwap and three baselines. We first compare the ratio of SLO-compliance functions when increasing the function number from 200 to 1200. As shown in Fig. 11a, 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



(a) Ratio of SLO-compliance 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 11: Cluster evaluation of FaaSwap.

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-compliance 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. 11b 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. 11b (right) plots per-worker load variance under FaaSwap, SimpleSwap and NonSwap, 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 Discussion and Related Work

Large models. Swapping large models can incur considerable overhead with long end-to-end latency. Compared with keeping a full model in GPUs using a large amount of memory, caching its partial parameters can be more efficient in model pipeline. We can explore how a large model can be partly cached in order to navigate the tradeoff between inference performance and GPU memory cost, which we leave as a future work. In addition, increasingly popular large language models, can even exceed the memory capacity of a single GPU, which requires careful designs of model parallelism and pipeline [15, 33, 37]. Supporting these models in serverless cloud can be more challenging, which we will also leave as a future work.

GPU virtualization. Though in this work we mainly discuss sharing an entire physical GPU across functions, the design of model swapping and request-level sharing can be naturally extended to virtual GPUs. In fact, FC allows users to configure a function with a proportion of GPU, which is enabled by GPU virtualization [2, 8–10]. Such techniques can be integrated into FaaSwap's GPU server to partition a GPU into multiple virtual instances, each shared across multiple functions at request level. This enables much fine-grained GPU sharing, which is our future work.

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

GPU function snapshotting. Models can serve as snapshots of inference functions, by that FaaSwap's model swapping can also be exploited in function scaling. When a serverless platform requires launching new instances for a running function, it can quickly duplicate the model across GPUs through NVLink, leading to substantially reduced startup latency compared with function cold starts. We leave it as a future work.

Model pipeline. Recent works have also leveraged model pipeline to reduce inference latency, such as PipeSwitch [14] and DeepPlan [24]. These works focus on improving inference performance of individual models, which is orthogonal to FaaSwap and can be applied to further optimize model swapping.

Operator-level optimizations. By redirecting CUDA API calls to the server, FaaSwap obtains the operator-level knowledge when executing a ML model. Recent works have proposed to optimize the execution of operators and GPU kernels, such as operator fusion [16], which can be exploited by FaaSwap to further speed up model inference.

9 Conclusion

We present FaaSwap, a GPU-enabled serverless platform for SLO-aware, resource-efficient model inference. FaaSwap keeps GPU functions alive in host and supports efficient model swapping, which can easily enable pay-per-use billing and efficient GPU sharing across functions. FaaSwap can also meet latency SLOs for inference functions at low cost with its scheduling and model management policies. We have implemented FaaSwap atop FC, and evaluations show that FaaSwap can effectively comply with per-function latency SLOs and improve GPU efficiency.

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.

A.2 Auto-configuration in Request Queueing

FaaSwap can automatically configure α based on overall load such as to maximize the number of SLO-compliance 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-compliance 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.

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

CUDA library	**API**
Runtime (Async.)	cudaMemcpyAsync
	cudaMemsetAsync
	cudaLaunchKernel
	cudaFree
Runtime (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

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] AWS Lambda. https://aws.amazon.com/lambda/.

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

Algorithm 2 α Auto-configuration.

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *Proc. USENIX OSDI*, 2018.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

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

[29] 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.

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

[31] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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