Opportunistic GPU Acceleration for Serverless Functions

Anonymous Author(s)
Submission Id: 144

Abstract

Hardware accelerators like GPUs are now ubiquitous in data centers and edge computing, but are not fully supported by common cloud abstractions such as serverless Functions as a Service (FaaS). Many popular and emerging FaaS applications such as machine learning and scientific computing can benefit from GPU acceleration. However, FaaS frameworks (such as OpenWhisk) are not capable of providing this acceleration because of the design mismatch between the GPU usage and FaaS programming model, which requires virtualization and sandboxing of each function, and must support highly dynamic and heterogeneous functions.

This paper presents the design and implementation of a FaaS system for heterogeneous hardware, which provides hybrid computing capabilities for general, black-box functions. We show how data and code locality determines GPU function performance, and translate principles from I/O scheduling such as fair queuing and anticipatory scheduling, to GPUs. On real-world FaaS workloads, our fair-queuing scheduling can reduce latency by more than 5x compared to FCFS and batch-oriented scheduling. Our scheduler-integrated memory movement optimizations significantly reduce GPU cold-starts, reducing function latency by more than two orders of magnitude, allowing FaaS operators to provide opportunistic acceleration and leverage antiquated GPUs.

1 Introduction

Function as a Service (also called FaaS) is an important and growing abstraction in cloud computing, with new and existing applications increasingly adopting serverless computing [69]. Users are enticed by its dynamic scaling, low cost, and ease of management, since the lifecycle of self-contained *functions* is orchestrated by the FaaS provider. FaaS has emerged as a common, narrow interface for a wide range of cloud applications such as web serving, machine learning (ML), internet of things (IoT), scientific computing, event-driven workflows and orchestration, etc.

Providing sandboxed and efficient execution for highly heterogeneous and dynamic FaaS workloads is one of the central challenges in serverless cloud computing [68]. FaaS workloads can be highly bursty, and have significant sandboxing overheads from function cold-starts when creating and starting virtualized environments (lightweight VMs or containers). The resource contention due to the above issues causes large performance degradation and poor function latency—and this cold-start problem will worsen as FaaS is used by a wider gamut of applications and their feature requirements increase.

In this paper, we seek to improve FaaS performance by developing *opportunistic acceleration* for serverless computing. Thanks to the boom in machine learning, accelerators like GPUs are now a common feature in data center servers and edge computing devices. Our goal is to provide GPU support to functions by leveraging these accelerators, and improve both function latency and server utilization. Functions are increasingly amenable to GPU acceleration, due to the adoption of the FaaS abstraction by applications across ML [18, 30, 57, 72], HPC [17, 36, 44, 62, 70], multimedia [15, 77], and other computationally intensive workloads.

Providing GPU acceleration to functions in black-box serverless environments poses three major challenges. **1.** The hardware and software stacks of GPUs are designed for highly parallel, throughput-intensive applications, and have limited support for multiplexing and virtualization. Typical functions run for milliseconds or seconds, which makes temporal multiplexing challenging due to the high context switch costs on GPUs. GPU virtualization technologies (such as MIG, MPS) are designed for a limited and static set of applications—however, function workloads are highly dynamic, which makes spatial multiplexing of compute-resources and memory challenging. **2.** From a serverless provider's perspective (which we take in this paper), functions are highly diverse and must run inside isolated and containerized environments which prevents the use of application-specific GPU multiplexing techniques (often used in ML training and inference [26, 31, 52, 55]). We thus seek general-purpose black-box solutions. **3.** Finally, we target heterogeneous GPU data center and edge machines that may lack state-of-the-art GPUs with enhanced virtualization support. Instead, we seek to run on out of data center GPUs which no longer provide adequate speedup for modern ML workloads and are stranded resources, and on edge GPUs with limited computing resources and multiplexing support.

We observe that cold-starts and (temporal) locality play a vital role in function performance on GPUs. Cold-starts associated with initializing GPU containers (such as nvidia-docker) can take *several seconds* and increase latency by up to 100×. While batching is a common technique with GPUs [14], doing so in a black-box and *fair* manner is challenging. We address these challenges through function scheduling, and use fair queuing principles to resolve the locality, fairness, and efficiency tradeoff. Specifically, we adopt scheduling techniques from I/O scheduling such as MQFQ (Multi-Queue

Fair Queuing [34]) and anticipatory scheduling [37] to develop a new class of GPU scheduling algorithms for FaaS workloads. We treat each function as a separate application, and dispatch invocations in order of their service requirements and priority weights, retaining the fairness properties of MQFQ while significantly reducing cold starts.

Developing an equivalence between I/O and GPU scheduling allows us to leverage classic and well studied ideas from disk scheduling, and provides a new and more principled approach to GPU resource management. Our work is unique in that it seeks to improve GPU utilization through scheduling, and does not depend on specialized application level approaches or hardware virtualization support. A major focus of prior work on GPU support for FaaS has been on application-specific optimizations (such as for ML inference) or disaggregation mechanisms. Our work is orthogonal, and investigates heterogeneous CPU and GPU function performance of realistic workloads at scale.

Since GPU memory is a precious resource, our scheduler is tightly integrated with memory management. We develop a minimalistic CUDA interposition technique for enabling virtual memory for GPU functions, and prefetch and swap out function memory based on their scheduler states. We implement and integrate our GPU scheduling policies and mechanisms in a state of the art high performance FaaS control plane, Ilúvatar [27], and support hybrid computing by dispatching invocations to both CPUs and GPUs based on their speedup. We make the following contributions:

1. We develop fair queuing based scheduling (called "MQFQ-Sticky") for black-box containerized functions, which preserves both locality and fairness by adapting ideas from I/O scheduling. This allows us fine-grained and intuitive control of both temporal and spatial multiplexing of GPU compute and memory resources. Our GPU memory management is integrated with the scheduler, and by virtually eliminating cold-starts, provides more than 300× reduction in latency compared to current GPU containers.
2. We extensively evaluate our scheduling policies and their tradeoffs using the Azure FaaS workload [61]. MQFQ-Sticky reduces the average latency by $1.2 - 10\times$, the variance by $3 - 8\times$, and improves fairness by more than 3×.
3. We have integrated our scheduling policies and mechanisms into the Ilúvatar [27] FaaS control plane, and provide the first open-source, practical, and high-performance GPU acceleration for black-box functions across data center and edge environments.

2 Background and Motivation

2.1 Providing Functions as a Service

Serverless computing entails executing snippets of user code, usually in an event-driven manner, inside protected and isolated sandboxes [60]. For users, serverless functions offer many features such as elastic scaling and scale-to-zero, making cloud-native applications easier to develop and deploy.

Table 1. Latencies (in seconds) for GPU and CPU **W**arm and **C**old functions.

Function	GPU [**W**]	CPU [**W**]	GPU [**C**]
Imagenet [ML]	2.253	5.477	8.581
Roberta [ML]	0.268	5.162	16.374
Ffmpeg [Video]	4.483	32.997	12.044
FFT [HPC]	0.897	11.584	2.648
Isoneural [HPC]	0.026	0.501	2.586
Lud [Rodinia]	2.050	70.915	2.125
Myocyte [Rodinia]	2.784	39.277	2.145
Needle [Rodinia]	1.979	144.639	2.292
Pathfinder [Rodinia]	1.472	134.358	1.997

The FaaS *provider* usually runs a *control plane* such as OpenWhisk and OpenFaaS (or proprietary ones in the case of popular services like Amazon Lambda [8]), for handling the scheduling, scaling, load-balancing, resource limiting, and accounting for each invocation.

From the FaaS control plane's perspective, the popularity and diversity of FaaS workloads makes these tasks particularly challenging, and can add several dozen milliseconds of latency in orchestrating function invocations [20, 21, 27]. Because of the richness of their usecases, function workloads are highly diverse with a range of several orders of magnitude in all dimensions. For example, both Azure [61] and Alibaba [47] workloads indicate that the inter-arrival-times can range from 0.1s to hours, the execution times can range from milliseconds to minutes, and the memory footprint from 100 MB to 5 GB. From the FaaS control plane point of view, an invocation is a "black-box" containerized task with known resource limits (such as the number of allocated CPUs and memory size). Function resource usage is typically similar across invocations, and control planes also use these estimates for implementing advanced policies for keep-alive [28, 58], placement, etc.

2.2 Why GPU Acceleration for Functions

Many applications that have adopted FaaS for its on-demand scaling also benefit from GPU acceleration, as shown by Table 1, where we compare the execution times of functions using an NVidia V100 GPU and an Intel Xeon Gold 3.2 GHz CPU. CPU functions are allocated one CPU core, and GPU functions can use the entire accelerator. Machine learning inference tasks such as Imagenet and Roberta see a 3x and 20x reduction in latency compared to a warm CPU container. Video encoding via *ffmpeg*, which is one of the most popular functions on AWS Lambda [3], can also leverage specialized hardware found in most GPUs for a 7x speedup. Scientific computing has started to be used in FaaS [38, 62, 63, 70], and also benefits from GPU acceleration for its common primitives such as FFT.

Thus a large class of FaaS workloads are potentially amenable to GPU acceleration. The serverless abstraction allows decoupling of computation from its location, and prior work

has investigated the use of remote disaggregated GPUs for FaaS [26, 51], and providing acceleration as a service [22, 67]. Serverless functions in public cloud with enabled GPUs have started to appear [1, 5], but are far from ideal. These use GPU-passthrough techniques [10], which statically allocate hardware and have low utilization, and sometimes one must even self-host the hardware [5].

2.3 GPU Programming Model

Applications cannot usually manipulate GPUs directly, and must use a manufacturer-provided driver for all operations. Host programs launch *kernels* which execute a code block with given memory inputs and a number of parallel threads to use. Multiple kernels can be launched concurrently, with the device handling scheduling of kernels and threads internally. Kernels run until completion, with execution time varying widely depending on the number of threads, input size, and complexity of the code being run. Traditionally, programs manually move data between the host and device (using `cuMemAlloc` for instance). Virtualizing GPU memory (by using CUDA's Unified Virtual Memory (UVM) [7]) allows memory overcommittment, which is important for supporting high degrees of multiplexing.

For multiplexing functions on GPUs, the naive approach entails fully assigning the GPU to a function. Because functions are short-lived (at most a few minutes), we can use the GPU in an FCFS run-to-completion model. However, this leads to poor GPU utilization since functions are often small and will not use all GPU resources, and also highly detrimental for function latency due to cold-starts, as we explain in the next section. A typical FaaS server runs 10–50 functions concurrently, and thus even if a small fraction of them can benefit from GPU acceleration, an immediate need arises for multiplexing the GPU to run multiple functions concurrently. However, GPUs have conventionally been designed for high throughput computation for a single long-running application, which is reflected in their hardware architecture and software stacks. The performance-first focus has also resulted in many cross-layer optimizations which make multiplexing and virtualization challenging.

3 Design Requirements and Key Challenges

Our work considers discrete and integrated GPUs, and our task is to provide *opportunistic* GPU acceleration to functions that benefit from it. We assume a classic, non-disaggregated hardware platform where the GPU functions can run locally. Due to the surge of ML workloads, discrete and integrated GPUs are a common feature in data center and even edge clusters. For example, the popular Nvidia Jetson Orin platform provides more than 60% of its compute capabilities (FLOPS) in the integrated GPU. Since FaaS is the common abstraction for supporting a wide range of applications, we seek a general-purpose *black-box* solution which supports arbitrary functions on heterogeneous clusters, and preserves existing resource isolation guarantees. This requires us to support different GPU usage patterns, and prevent the use of application-specific performance optimizations (such as for ML inference). For the FaaS hardware, we make minimal assumptions in terms of feature availability, and assume that the underlying cluster is a mix of servers with different GPU types, and support both data center and edge computing devices. Given all these requirements, the performance objective is to increase server utilization by running functions on GPUs, and reducing the overall function latency of FaaS workloads. The above constraints imposed by black-box FaaS workloads and GPU multiplexing also restrict the space of mechanisms and optimizations, which lead to the challenges described below.

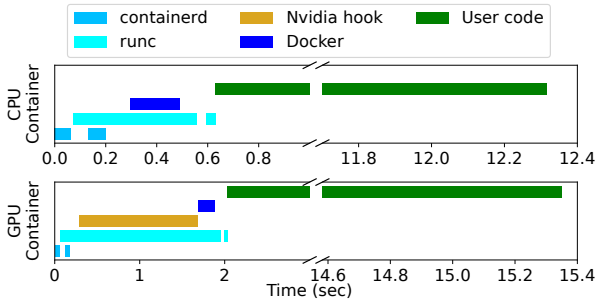


Figure 1. Timeline of cold-starts of CPU (top) and GPU (bottom) function containers running TensorFlow inference code. GPU initialization and code dependencies increase latency by three seconds.

3.1 Cold-starts for GPU Containers

Cold-starts due to sandbox creation and initialization are a well known performance problem for serverless functions [23, 46, 48, 50]. We find that such cold starts are severely exacerbated by GPU containers, increasing latency by $1.1 - 75\times$ (Table 1). A breakdown and comparison of these overheads for the ML inference function (using TensorFlow) is shown in Figure 1. For the GPU container (bottom figure), the Nvidia hook library adds more than 1.5 seconds of delay. User function code loads additional GPU libraries and dependencies, and its startup requires 1.5 additional seconds.

3.2 Tradeoffs in Locality, Throughput, and Fairness

A common approach to alleviate cold starts is to keep the container warm [28] in memory. This is also applicable for GPU containers, but with additional challenges. First, a warm container holds GPU memory which is much more limited (e.g., the V100 has 16 GB VRAM), which reduces keep-alive's effectiveness [28], and reduces the number of containers able to be kept warm. Additionally, the number of concurrently executing functions should also be kept low to mitigate performance interference, which can be excessive from both compute [56, 73] and data movement between host and device [35, 75]. This reduction in concurrency increases queue

waiting times. Finally, since function workloads are highly heterogeneous and dynamic, we must select the "active" functions carefully so as to balance fairness and throughput.

In other contexts, application switching costs have been reduced through batching. For example, an ML inference task can be given multiple inputs simultaneously as a batch—significantly improving throughput and utilization [13, 14, 74]. Such specialized and white-box solutions eschew isolation by making assumptions about the workload, such as the ability to modify the function code and input/output processing. For a general FaaS service, we need to provide locality improvements using a black-box approach.

Maximizing for locality entails large batches, which increases latency for both the batched function and also the other functions due to monopolization of GPU resources. For heterogeneous and dynamic FaaS workloads, batching policies are also significantly more challenging outside more specialized workloads like inference-as-a-service. For example, popular functions see 100× the average number of invocations, which can cause the rest of the long tail of functions to have exacerbated waiting times in the queue, leading to unfairness. Thus, while locality is useful for performance, it may lead to unfairness.

3.3 GPU Multiplexing Mechanisms

While many hardware- and software-level virtualization and multiplexing solutions exist, they are ill-suited for the highly dynamic and heterogeneous containerized function workloads. The conventional approach is to dispatch multiple GPU compute kernels (corresponding to different functions) concurrently, and let the GPU driver and hardware manage the sharing of GPU compute cores and memory. The GPU driver itself accepts individual compute *kernels* from applications and has a device-internal scheduler which maps them to available compute blocks as they arrive. This is one of the main mechanisms behind classic GPU virtualization work [24, 35, 75], but comes with significant performance overheads [76]. These GPU virtualization approaches duplicate application state in host memory, and allocate/deallocate all resources when switching between applications.

Application-specific optimizations for scheduling and multiplexing of GPU kernels can be performed at several levels of the GPU software-hardware stack. Kernel schedulers for domain-specific optimizations [19, 31, 43, 64] have been designed to coordinate kernels from several applications to improve on device scheduler performance. These operate only on compute kernels to prevent contention and assume active concurrent workloads are coordinated elsewhere to not exceed the device memory. Application-optimized and integrated scheduling solutions have recently been developed [52, 55, 64], which interpose on the application's kernel launches. For example, the structure of ML inference applications can be used for injecting kernel preemption code into the applications via TVM transformations [33] for decreasing head of line blocking and improving GPU utilization.

GPU compute cores and memory can also be *spatially partitioned* across clients/kernels using more modern GPU virtualization functionality. Nvidia MIG [54] (Multi-Instance GPU) pre-partitions device resources, and one or more of these virtualized GPU partitions can be assigned to a VM or container via direct device assignment. However, the slices are of pre-determined sizes, making them ill-suited to fine-grained and dynamic workloads [45]. Similarly, Multi-Process Service [53] (MPS) allows multiple processes to make share the device concurrently and has been proposed for FaaS [31]. An MPS server and the hardware device perform resource partitioning based on configuration at application start time. MPS is explicitly designed to let cooperative processes share GPU resources, and documentation specifies that it is intended to work with OpenMP/MPI applications [53]. If any process fails and crashes, all processes connected to the MPS server will crash, meaning one faulty serverless function will break all functions using that GPU, which we have frequently experienced in our testing.

Thus, neither MIG nor MPS are suited for serving FaaS workloads. Moreover, they are not uniformly available across GPUs—for example, Nvidia's Jetson IoT GPUs do not support MPS [2], and MIG support is only found in select GPUs after 2020 supporting Nvidia's Ampere architecture. Our requirement is to support acceleration for GPUs which may not be state of the art and thus not support hardware assisted virtualization such as MIG or vGPUs. Instead, we seek to interweave scheduling and multiplexing techniques to maximize fairness and throughput.

4 Design: Scheduling GPU Functions

The main new contribution and component of our hybrid serverless computing platform is the scheduler for GPU functions, which we describe in this section. The different hardware and workload model of GPUs requires a different approach to scheduling as compared to conventional CPU function scheduling. To tackle these challenges, we first show that fair queuing as used in I/O scheduling can be a useful framework for GPU scheduling (Section 4.1), then describe the GPU-specific scheduling (Section 4.2) and memory management optimizations (Section 4.3).

4.1 Key Insight: GPUs as Multi-Queue I/O Devices

We claim that the tradeoffs and challenges of guaranteeing fairness and high throughput for serverless GPU workloads are similar to modern I/O scheduling. Specifically, we can view GPUs as multi-queue I/O devices, and use fair scheduling algorithms like MQFQ [34] to provide a rigorous and well-tested conceptual framework. From a workload perspective, the I/O heterogeneity and fairness of different applications is similar to FaaS functions' heterogeneity. Similarly, modern disks have multiple internal dispatch queues, which

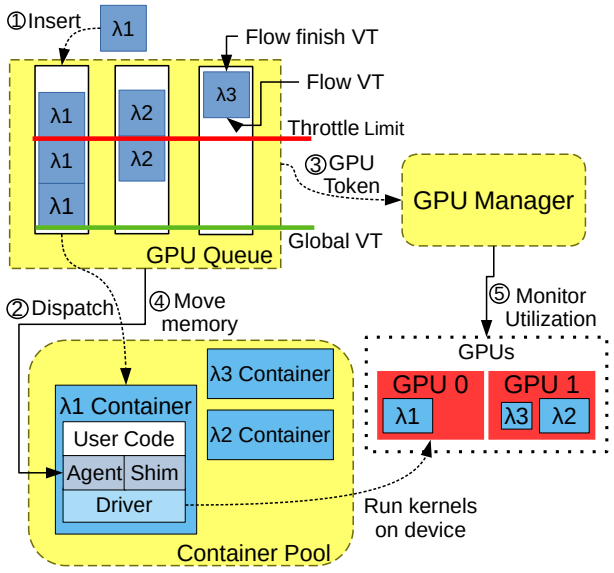


Figure 2. Scheduling GPU functions as flows with Multi-Queue Fair Queuing. Invocations are dispatched based on the virtual time. The container pool helps with warm starts.

also maps to the temporal and spatial multiplexing offered by GPUs. Successive invocations see lower latency from temporal and even spatial locality, a fact requiring significant attention when maximizing throughput.

Our scheduler maintains multiple dispatch queues, each queue corresponding to an individual function. These queues (also called *flows*) hold invocation requests which are analogous to disk read/write requests. Multiple invocations dispatched from a single flow preserve temporal locality and benefit from warm starts. The GPU, just like modern NVMe disks, also supports parallel dispatches, and we keep a subset of flows in the "active" state. Flows without en-queued invocations are "inactive". Fair queuing uses the notion of virtual time (*VT*) to capture the amount of service rendered to flows (normalized by priority weight). The flow's *VT* grows by a fixed increment after an item is removed and dispatched for execution. Flows are selected for dispatch based on their *VT*'s, and fairness arises from a bound on the maximum difference of flows *VT*'s.

Our insight is that the above classic fair queuing framework can be extended to meet the challenges of serverless GPU scheduling. Locality is maintained by dispatching successive requests from an active flow, by borrowing ideas from anticipatory scheduling and using MQFQ's concurrent dispatch, which increases throughput but still maintains fairness (albeit with a larger inter-flow *VT* bound compared to classic fair queuing with a single active flow). Each flow can be handled by a separate thread which can dispatch requests concurrently, taking advantage of device-level parallelism. The amount of device parallelism is configured and

Table 2. Key symbols and parameters for MQFQ-Sticky.

Symbol	Description
VT	Virtual Time, device wall clock service time accrued by a flow
Global_VT	Minimum *VT* across all active flows
T	Amount any flow's *VT* can exceed *Global_VT* before being throttled
D	Device concurrent invocations, can be fixed or dynamic with an upper-bound
TTL	Time-to-live for an empty flow to become inactive

controlled via tokens with the *D* parameter (Table 2). To prevent popular functions from monopolizing the GPU, flows are *throttled* if their *VT* exceeds the global *VT* (*Global_VT*) (which is the minimum of all flows' *VT*) by a certain threshold (*T*). *Fair queuing provides the necessary parameters for principled and tunable batching in an online manner, for highly dynamic and heterogeneous FaaS workloads.*

4.2 MQFQ-Sticky: Locality-enhanced Fair Queuing

The above MQFQ scheduling framework was designed for I/O. However, GPU functions have different compute and memory footprints, execution runtimes, and cold vs. warm execution times; all of which diverge from disk assumptions. The GPU device model is also different: the device parallelism is lower (SSDs support hundreds of active threads), and execution performance is highly sensitive to utilization and interference. We therefore modify the original MQFQ design to account for these differences, to get the maximum performance out of our accelerators.

Unlike disk requests with uniform block sizes, function execution times can vary significantly. We account for this by tracking the historical average execution time τ_k of each function k, and when an item is dispatched, its flow's *VT* is instead incremented by τ_k/w_k, where w_k is the priority weight of the function. Thus shorter functions are allowed more *invocations* than their long counterparts, but both get equivalent wall time on the GPU. After an invocation is dispatched, *Global_VT* is updated if necessary to a potentially new global minima across flow VTs.

Since GPU memory is limited, we use the flow states for **proactive memory management.** Flows that become active have their data moved onto the device in anticipation of use (if space is available). When a container is about to execute, we proactively move all its data to the GPU. Conversely, throttled and inactive flows have their data moved off-device, because we don't expect them to run in the near future. More details about the data monitoring and movement are described in Section 4.3. Further modifications of ours pertain to how functions are drained and added to the queues, and are described below.

Anticipatory Scheduling. Function performance is impacted by the availability of a warm container, and data

in GPU memory. We introduce anticipatory scheduling to MQFQ to maximize the use of both. Anticipatory scheduling for disks [37] boosts locality by keeping request streams "active" even if they are empty, in anticipation of future requests, which is especially beneficial for interactive applications. If a flow is empty (i.e., it has no pending invocations), then instead of immediately marking it inactive, we provide a grace period. Without this grace period, because of the proactive memory management described above, functions would see their warm containers immediately removed from GPU memory. Instead, we keep empty flows active for a configurable TTL (time to live), based on the function's inter-arrival-times. Specifically, we set the flow TTL to $\alpha \times$ IAT, where α is a tunable parameter. This policy is guided by the observation that reuse-distance is long-tailed [28], so a single global TTL is not ideal for both popular and rare functions.

Flow Over-run. Our second technique for improving warm starts is to allow the flows to dispatch invocations in small "mini-batches". An active flow's start time is allowed to be T units ahead of *Global_VT*. T is the second main configurable parameter: larger values will result in larger batches and more locality, but less fairness, since functions will have to wait longer before their batches are dispatched. If $flow.VT + T \geq Global\_VT$, then the flow is *throttled*. It may return to the *active* state only after other flows get to run and the *Global_VT* increases.

Device Concurrency and Feedback. Because each function uses different amounts of compute and memory during execution, a fixed level of device parallelism (D) like in disk scheduling may be sub-optimal. We therefore track memory usage of running containers and GPU utilization to adjust D dynamically, to minimize contention and execution overhead. This utilization-based feedback permits different scheduling rates based on the dynamic workload characteristics. We take two input parameters: the device utilization threshold (such as 90%), and the maximum parallelism level (irrespective of utilization). A coarse-grained controller loop runs every 200 ms to check the real-time utilization and changes the D level dynamically to ensure the utilization is under the threshold. Higher thresholds increase utilization and reduce queuing, but risk higher performance interference. More details of memory usage and GPU monitoring are described in Section 4.3 and Section 4.4 respectively.

Dispatch Concurrency. In classic MQFQ, flows can concurrently dispatch their requests (as long as they are within the T threshold). From a FaaS control plane perspective which needs to track invocation status carefully, we prefer a *single* dispatch thread which picks the eligible candidate flow queue (such that $flow.VT < Global\_VT + T$). Thus, the dispatch is not concurrent, which results in the most fair outcome via selecting flow with the lowest VT (i.e., earliest arrival). However, this reduces locality and batching opportunities, resulting in poor function performance.

To remedy this, we pick the next flow from the set of candidate flows by sorting on both recency and locality, as shown in line 6 of Algorithm 1. Our heuristic prefers longer queues which provide more batching opportunities and reduces their larger backlog. Ties are broken in favor of the flow with the least number of currently executing invocations (Line 9). This encourages multiple flows to progress and reduces the chance of a cold start caused by concurrent execution of the same function. Flow stickiness from this heuristic provides sufficient temporal locality between active flows to maximize throughput. This completes the description of the key attributes of our MQFQ-Sticky algorithm. We note that it maintains the fairness properties, since we are basically emulating dispatch concurrency, but prioritizing longer flows within the eligible flow window. That is, we still retain the MQFQ fairness property, which states that the service received (S) by two active flows during a span of wall-clock time (t_1, t_2) is bounded by:

$$\left| \frac{S_i}{w_i} - \frac{S_j}{w_j} \right| \leq (D-1)\left(2T + \frac{\tau_i}{w_i} - \frac{\tau_j}{w_j}\right) \quad (1)$$

Because of the controlled concurrency, a smaller bound may be possible for MQFQ-Sticky, which is part of our future work. The intuition is that classic MQFQ has $O(T!)$ possible permutations for dispatch, whereas our heuristic restricts the permutation space which is a strict subset.

4.3 Integrated Memory Management and Scheduling

Each container has a custom shim that intercepts calls to the GPU driver, specifically those for initialization and memory allocations. Requests by functions to allocate physical memory are captured in and converted into UVM (virtual device memory) allocations, allocation metadata is stored, and the result is returned to function. We use MQFQ flow states to guide memory movement. When some flow becomes active, all its CUDA-malloc'ed regions are *prefetched* into the GPU memory, in anticipation for continued use. We introduce and maintain a *container pool* of such created GPU containers, and executions of the function results in a *GPU-warm* start. The efficacy of this container pool is restricted by physical GPU memory, and thus throttled and inactive flows have their regions marked for eviction. This entails swapping and moving their GPU memory regions back to the much larger host CPU memory, with this eviction done asynchronously using LRU (least recently used) order. In rare cases, a throttled and swapped out flow may get invoked again, which leads to a *"GPU-cold but CPU-warm"* start since the container is already fully initialized, but data dependencies are not located on device. In the above case, prefetching may need to evict some other container's GPU regions, increasing latency.

4.4 Multi-GPU Load Management and Feedback

The GPU monitor is responsible for two key mechanisms: tracking GPU assignment and monitoring GPU utilization.

Algorithm 1 MQFQ-Sticky algorithm.

```
1:  procedure DISPATCH
2:      Global_VT ← min_{f∈flows}(f.VT)
3:      chosen ← None
4:      for flow ∈ flows do
5:          update_state(flow, Global_VT)
6:      cand ← filter(flow.active ∧ flow.len > 0, flows)
7:      sort(cand, on : length)
8:      if D ≠ 1 then
9:          sort(cand, on : in_flight)
10:     chosen ← top(cand)
11:     token ← get_D_token(chosen)
12:     if token == None then
13:         return None
14:     return chosen.pop()
15:
16: ▷ Update state of flow, given the global VT
17: procedure UPDATE_STATE(flow, Global_VT)
18:     if flow.is_empty and flow.in_flight == 0 then
19:         if Date.Now() − flow.last_exec ≥ TTL then
20:             ▷ Flow has expired
21:             flow.state ← Inactive
22:     else if flow.VT − Global_VT < T then
23:         ▷ Flow has exceeded threshold
24:         flow.state ← Throttled
25:     else
26:         flow.state ← Active
```

Creating a GPU context uses physical memory we can't control, so the monitor only allows a fixed number of containers to exist at one time. Note that we support multi-GPU systems, and still maintain a single set of MQFQ flow queues per system. New functions are launched on the GPU with the most available resources and future invocations of the function run on the same GPU to take advantage of locality.

Dynamically setting D is dependent on the two computing and memory resource limitations of GPUs. Because we intercept memory allocations, we can closely track device memory usage of containers thanks to our driver shim, and only allow a new dispatch when the needed container won't overload the device's physical memory. Ensuring there is available compute is not as straightforward to manage as memory due to unpredictable function characteristics. An ML inference task for example could have known compute usage, since input and weight tensors have fixed uses based on the execution graph. Other types of GPU functions can launch compute kernels unpredictably, based on the application's internal control flow. The actual size and number of kernel launches often vary with function arguments, making an a priori utilization intractable. Accepting this, we choose to externally monitor device utilization and launch new invocations when headroom is deemed sufficient to support another dispatch. To avoid a thundering herd of launches, we increment the tracked usage by a factor of $1/D$, and let the next monitoring update capture actual usage. When both resources are deemed available for a dispatch, we "increase" D by allowing the item to start executing. For this additive increase control-loop, we require $D_{\max}$ as another parameter for providers to set upper-bounds based on the workload and service requirements.

5 Implementation and Microbenchmarks

This section provides implementation details of our opportunistic GPU functions and microbenchmarks of the optimizations. We have implemented the scheduling and multiplexing policies as part of a control plane for hybrid serverless computing. We use Ilúvatar [27] as the base, and our implementation has has three main components: i) GPU scheduling policies, ii) CUDA multiplexing, and iii) policies for dispatching functions to both CPUs and GPUs based on speedup. Ilúvatar is a low latency and highly scalable FaaS control plane whose overheads are more than 100x lower than OpenWhisk for CPU workloads, which it achieves due to per-worker queues, and asynchronous and batched execution of non-critical resource management operations. Our GPU function implementation adheres to the above design principles, and is implemented in around 3,000 lines of Rust.

Invocations are dispatched by a dedicated thread which monitors available GPU resources, and if GPU tokens are available, selects a function to execute based on Algorithm 1 described in the previous section. The dispatch thread is also notified upon invocation completion events, which helps maintain high device parallelism (D). For servers with multiple GPUs, we maintain a single dispatcher which allows late binding of functions to individual GPUs. Locality is still maintained by the dispatcher implementing "sticky" load balancing among GPUs: line 9 helps us avoid moving functions across GPUs and miss the high impact of cold-starts. Function flows and VT tracking are kept in one data structure, and protected behind Read/Write locks.

GPU functions have a special "Device" tag as part of their registration metadata, which also informs scheduling decisions. GPU functions may also register a CPU counterpart, which is invoked when GPU acceleration is not necessary. The CPU vs. GPU dispatch decision is made before functions are inserted into the respective device queues. Our current dispatch policy creates and uses the GPU speedup as the key decision metric, and runs the top-p percentile of functions on GPUs (i.e., using a relative utility framework). We have also implemented more advanced dispatch policies which consider multiple attributes like popularity and speedup, but we focus on the above simple static policy to keep the focus on the GPU scheduling.

Utilization monitoring. We track both GPU compute utilization and per-container and total device physical memory usage. Using NVML [4] bindings in Rust, we query compute utilization and record its instantaneous and moving average,

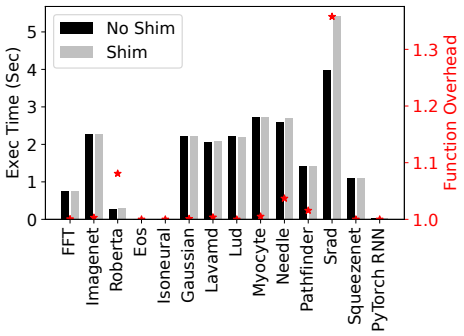


Figure 3. Functions see little to no impact from our interception and substitution of allocation calls. This matches performance promised by Nvidia for UVM applications.

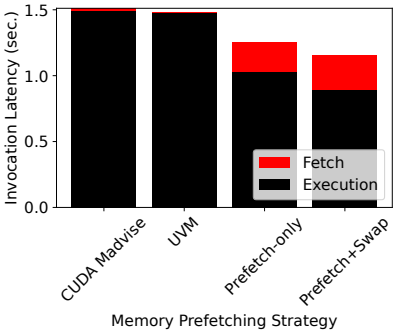


Figure 4. Active memory management (`Prefetch+Swap`) improves execution latency.

for use in determining D. We query this information every 200 ms to balance having up-to-date information while avoiding excessive CPU utilization on the host. To track memory usage, our shim includes a report of memory allocations still held by the application to the worker alongside other invocation results. This data updates the container's record in the worker, which then tracks memory pressure on the device. When a container needs to run, or a flow is made active, we can evict containers belonging to inactive flows.

5.1 CUDA Interposition Shim

We run functions inside Docker containers [6] using the Nvidia Container Toolkit [9] to attach specific GPUs to them. For dedicated GPUs with limited memory, we use a CUDA interposition "shim" which intercepts CUDA calls made by the function. Our shim implementation is similar to NVShare [12], but simpler: we only use it for intercepting memory allocation calls and forcing the function to use virtual memory. This requires about 500 lines of code (written in C) to be injected using `LD_PRELOAD`. We use CUDA's Unified Virtual Memory (UVM) to oversubscribe device memory. When using UVM, the application sees a unified host-device memory space, with memory pointers being valid in both spaces. The CUDA driver moves and ensures coherency of UVM memory between the host and device as use and pressure demands, mimicking disk-based swap space found in operating systems. Our shim intercepts all calls to the driver for allocations for physical memory made via `cuMemAlloc`, and makes a UVM allocation of the same size using `cuMemAllocManaged`. We record the size and memory pointer position, then return the pointer to the application, thus maintaining execution transparency. It can use this memory as if it were physically allocated, reading, writing, or copying it to the host using traditional driver calls. If the function already uses UVM, we intercept and forward its allocations, recording the metadata for our memory management tasks.

The performance overhead of our interception is primarily influenced by the memory access patterns of the function and the extra layer of virtual memory (UVM), and is shown for different functions in Figure 3. All results are averaged over 10 trials, and we see a negligible latency impact on most functions. The rest see single-digit percentage increases, with `Srad` standing out with a 30% overhead in execution time due to the UVM shim. These results are in line with Nvidia's own reporting on the performance change when migrating applications to UVM [7]. This low overhead shim is thus ideal for virtualizing GPU memory.

5.2 Memory Management

We implement the warm pool, memory prefetching, and swapping optimizations inside the control plane, integrated with the scheduler. Recall that we prefetch the GPU memory of active functions, which is not provided out-of-the-box by CUDA UVM. Default UVM only moves memory on-demand, and also exposes `madvise` hints (via `cuMemAdvise`) for memory ranges, but neither allow for deterministic control of memory placement. Our default policy (`Prefetch+Swap`) asynchronously copies memory to the device before invocation, and swaps it back to host memory after the flow becomes idle (or evicted on-demand using a least recently used policy). After we choose a flow for dispatch, we call `cuMemPrefetchAsync` from the shim to prefetch the container's GPU memory. Doing this in a non-blocking manner allows us to overlap prefetching with the control plane marshalling invocation arguments to send to the container. Not having to block while waiting for memory to be moved saves significant time on the critical path. When a flow is throttled or memory is needed to run other functions, we direct the shim to again use `cuMemPrefetchAsync` move memory off the device to CPU memory.

We compare different memory management policies in Figure 4. We run 16 copies of the FFT function from Table 1, each using 1.5 GB of device memory which oversubscribes the GPU's memory by 50%. Each copy is sequentially invoked 20 times. The impact of these different memory policies are displayed in Figure 4, with average time spent in-shim shown in red and function code execution in black. With such high overcommttment and the stock UVM driver controlling data placement, the execution time is 40% worse

than the optimal seen in Table 1. Execution time is higher because memory must be paged in on-demand from the host as kernels access it, and old memory paged out. Surprisingly, using `CUDA Madvise` to control memory placement performs slightly worse. The madvise calls to the driver don't actually move any memory, and we just waste time sending the memory directives with no benefit to execution time. In contrast, our `Prefetch+Swap` policy reduces latency by over 33% compared to stock UVM. We also compare against a `Prefetch-only` policy which does not proactively swap function memory but instead relies on UVM for reclaiming pages. This shows that adding the swapping optimization (our default), provides a latency improvement 6%, and matches the ideal non-UVM execution time listed in Table 1.

Our system can also run on heterogeneous and edge hardware, such as the Nvidia Jetson Orin AGX. Because its integrated GPU has no dedicated memory, our memory prefetching optimizations are not applicable, but all the other locality, fairness, and dispatching enhacements are relevant.

6 Experimental Evaluation

Our experimental evaluation is examines the effectiveness of our scheduling based approach for opportunistic GPU acceleration for FaaS workloads for three main metrics: i) on-GPU execution latency (which captures how efficiently we are using the hardware), ii) per-function end-to-end latency which includes the queue wait time, and iii) device utilization and throughput for hybrid CPU-GPU systems.

Setup and Workloads. All experiments were run on servers running Ubuntu 20.04 on kernel version 5.4, with a 48 physical core Intel Xeon Platinum 8160 CPU with hyperthreading disabled, 250 GB of RAM, and an Nvidia V100 GPU running driver version 470.239.06. This isn't the latest GPU hardware, and emphasizes that our design can work with a variety of hardware, doesn't require advanced features, and is easily scalable and adaptable to other systems.

Function were sampled from Azure trace [61] in the same manner as previous works [27, 29], and function frequencies are scaled using the empirical CDF of the inter arrival times, to yield workload traces of different intensities. Based on the execution times, we select the closest matching GPU function from Table 1 that doesn't exceed that time. Each experiment is run with the same trace composed of 24 functions, run for 10 minutes, and presented results are the average of 5 repeated runs. We evaluate on multiple traces with different function mix and invocation frequency distribution, providing a wide spectrum of realistic workloads with different heterogeneity and device loads (Table 3). Although MQFQ-Sticky is capable of enforcing per-function QoS, we set the weight of all functions to 1, for ease of exposition. The evaluation is presented in a top-down manner: we first analyze the end-to-end latency, then show scaling behavior, and finally investigate the effect of various MQFQ-Sticky parameters described earlier in Table 2.

Scheduling Policies. To examine the locality, fairness, and utilization tradeoff, we implement and evaluate three *additional* scheduling policies in addition to our default MQFQ-Sticky. The second policy is `FCFS`, which uses the warm pool and memory management (we move function memory on-device before executing, and move it back off after each invocation has completed). This represents a scheduling policy with most of the important GPU optimizations, but one which is not fully locality or fairness aware. Our third and final scheduling policy variant is `Batch`, which also uses all the GPU optimizations, and maximizes locality by batching invocations of each function individually. Unlike MQFQ-Sticky, `Batch` greedily maximizes batch sizes (and hence locality) irrespective of the build-up of other functions.

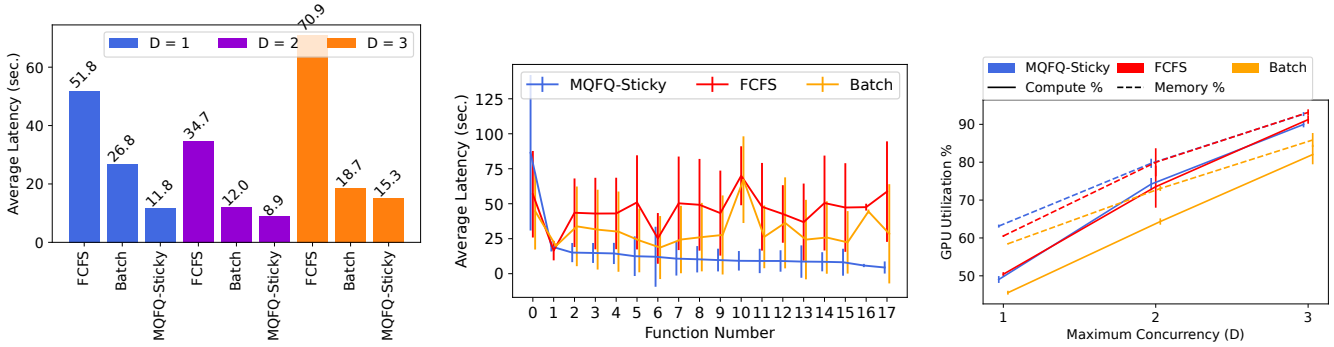
Using the MQFQ design allows for easy parametrization and implementation of these and other policies for exploring the locality vs. performance tradeoff. Both `FCFS` and `Batch` use the integrated memory management and CUDA interposition shim, and thus allow us to separate out the impact of MQFQ-Sticky's core ideas with minimal code changes and differences. For `FCFS`, all functions are inserted into a single queue. For `Batch`, we insert invocations into per-function flows, and dispatch the entire *flow* containing the oldest item.

6.1 GPU Scheduling Performance

To characterize the differences in the above scheduling policies, we first show the empirical evaluation with a *medium-intensity* workload, which comprises of 24 functions with an average arrival rate of 2 invocations per second. This workload results in average GPU utilization of around 70% (Figure 5c), and represents the average case.

Average Latency. The latency across all invocations is shown in Figure 5a. Not shown in the figure is the current baseline `FCFS Naïve` scheduling with Nvidia-docker, which does not have a container pool and suffers from excessive cold-starts. **The `FCFS Naïve` average latency is close to 3,000 seconds—a 300× overhead.** The high latency is because of every invocation results in a cold-start, causing a large queue buildup. Note that our workload trace is open-loop, and thus invocations are generated at fixed intervals.

MQFQ-Sticky outperforms `FCFS` by 5× with a 11.8 vs 51.8-second average respective latency thanks to its locality and fairness oriented design. `Batch` has middle of the road performance, lacking fairness and advanced locality policies. For this workload, at higher concurrency levels, MQFQ-Sticky improves latency by an additional 25% to an 8.9-second average per invocation. Both competing policies also benefit from concurrency, but neither outperform MQFQ-Sticky. When D is set too high (D=3), the device cannot handle the higher concurrency, and all policies suffer varying degrees of degradation due to resource contention and interference. For this workload, the queuing delays account for more than 99% of the end to end function latency, and thus scheduling policies have significant impact.



(a) Average latency is significantly lower with MQFQ-Sticky for different device-parallelism (D) levels.

(b) The average and variance of **per-function latency** is much lower with MQFQ-Sticky.

(c) Device utilization for the medium-load trace.

Figure 5. Latency, fairness, and utilization for a medium-intensity FaaS workload.

Table 3. The latency benefit of MQFQ-Sticky improves with increasing GPU utilization.

GPU Util (%)	Req/s	MQFQ-Sticky(s)	FCFS(s)	Batch(s)
28.025	1.122	3.932	7.843	4.962
35.747	1.797	2.432	4.511	2.838
38.889	1.943	3.434	8.031	3.441
42.740	1.690	1.046	1.371	1.054
43.347	2.572	2.929	7.309	3.454
52.054	1.125	2.956	3.512	2.415
57.467	4.263	5.630	37.743	9.585
68.141	2.693	10.543	44.450	13.570
74.216	2.553	8.899	34.719	11.996

Fairness. In Figure 5b, we show the per-function latency (averaged across all its invocations). We use inter-function latency variance as our fairness metric. `FCFS` has the worst global inter-function latency variance (752), and the highest average latency. MQFQ-Sticky reduces latency in the range of $2-10\times$, and has only one-third the inter-function latency variance of `FCFS`. Also, the invocation latency variance for each function (the error bars) is $3-4\times$ lower compared with `FCFS` and `Batch`.

Result: *MQFQ-Sticky policy provides a* $5\times$ *reduction in average latency across all functions, and also reduces their jitter and tail latency by* $3-4\times$.

6.2 Scaling

We now look at load, GPU, and memory scaling properties of MQFQ-Sticky.

Latency vs. load. Table 3 shows the latency for different workload traces, each with a different mix of functions and IATs (inter arrival times), resulting in different average GPU utilization. In general MQFQ-Sticky performs better at higher utilization, reducing average latency by $4-6\times$ vs. `FCFS`. When considering the latency weighted by the number of invocations, and normalized to the no-interference case, the performance gap is even higher, since smaller functions are more affected by cold-starts and unfairness in relative terms. MQFQ-Sticky's weighted normalized latency is more than $10\times$ lower (vs `FCFS`) at higher loads, and $2.5\times$ lower vs. `Batch`.

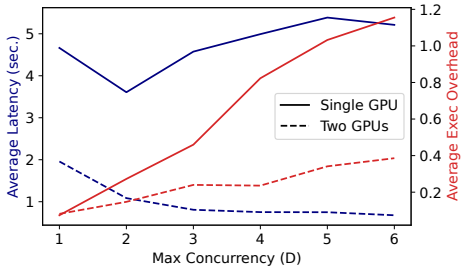


Figure 6. MQFQ-Sticky also uses locality-aware scheduling for multiple GPUs, significantly reducing queuing.

Table 4. Hybrid CPU+GPU reduces latency by more than 2x compared to CPU-only and GPU-only execution.

Case	Avg. latency (s)
CPU-only	2.03
1 GPU	3.43
2 GPUs	1.08
CPU+GPU	1.00

Multiple GPUs. Our system easily scales to orchestrating and dispatching across multiple GPUs. We run a high-load trace and show the comparison in Figure 6 after we add a second, identical, GPU to the server. Two GPUs not only allows us to run $D \times 2$ invocations, but also do on-the-fly load balancing between them to avoid compute contention with higher D. As a baseline, the multi-GPU blue dashed line has $2.3\times$ lower latency at D=1. At higher device parallelism, the multi-GPU case sees a latency reduction of $4\times$ vs. the single GPU setting. Device parallelism also slightly increases the execution overhead due to interference, but is offset by the smaller queues.

Hybrid CPU+GPU execution. For evaluating hybrid execution, we use a GPU-speedup based dispatch policy. We

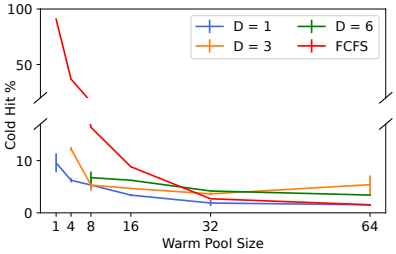


Figure 7. Container-pool reduces cold-starts. MQFQ-Sticky provides higher locality.

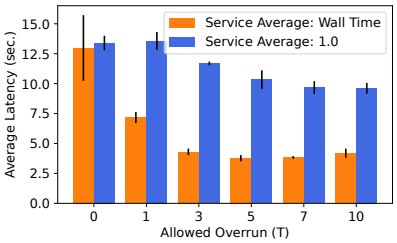


Figure 8. Larger T yields more batching and lower latencies because it allows popular flows to run ahead. Using historical function execution latencies helps significantly compared to uniform flow costs in classical fair queuing.

use offline profiling to obtain the GPU speedup for functions, and only run the top 50 percentile of functions on GPU, and rest use the CPU. This corresponds to functions having GPU speedup of $> 3\times$ being eligible for GPU acceleration. Other functions avoid queuing for the GPU and run immediately on the system's plentiful 48 CPU cores. The average function latencies are shown in Table 4 for a high-intensity workload. Due to excessive queuing and contention, a single GPU degrades latency by 38%. Adding a second GPU alleviates this load and reduces latency by half. Interestingly, hybrid execution with the CPU and single GPU reduces latency even further, thus showing the benefits of heterogeneous hardware for FaaS workloads.

Container Pool Size. For temporal locality, the invocation patterns and batching plays a key role in reducing cold starts. The performance difference between MQFQ-Sticky and FCFS can be largely attributed to the cold-hit ratio of the invocations. Figure 7 shows the "miss-rate curves" for the medium-intensity trace as we increase the number of containers in our container pool. We focus on the number of containers in the pool, rather than MB of pool memory for simplicity. Idle containers do take up CPU memory, and work managing the memory used by caching containers is orthogonal to our design [28]. Since MQFQ-Sticky prefers smaller batches of functions and does anticipatory keep-alive, it has a high temporal locality and its cold-hit % is in the range of 2-8% across a range of pool sizes and device concurrency. In contrast, FCFS has 50% cold-starts with a pool size of 4, and achieves parity with MQFQ-Sticky only at largest pool sizes when the popular functions can fit in the container cache.

6.3 Impact of Scheduling Parameters

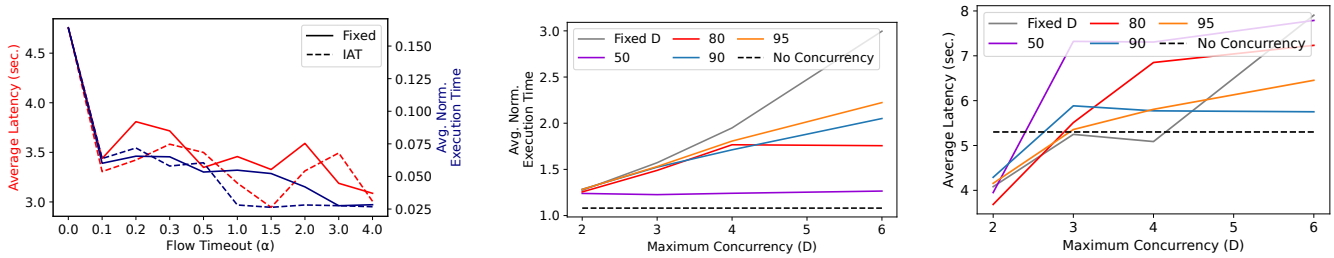
In this subsection, we explore the effects of configuration knobs to see their effect on performance, which also sheds a light on the empirical relationships between fundamental parameters of locality and throughput.

Flow over-run (T). Recall that flow virtual times are within T of each other. Larger T results in more locality and batching opportunities, but decrease fairness, since flows may get to monopolize resources for longer before being throttled. Figure 8 shows the average latency decreasing, but with diminishing returns, as the over-run is increased. The figure also shows the value of using function wall clock execution times. When all flow usages are assumed to be constant (1.0 in the figure), long functions may dominate, which increases average latency by more than 3x. *Thus, a small amount of over-run guided by function characteristics helps significantly.*

Flow keep-alive TTL. Empty flows remain active until a TTL expires, after which they're made inactive and have their resources evicted. Figure 9a shows the improvement to both execution time as compared to ideal warm performance and latency as the TTL grows. The execution overhead is reduced because of warm start locality, and is the main factor behind latency reduction. Setting any global TTL (the solid line), at even a small 0.1 seconds, improves latency and overhead by 25% and 50% respectively. Increasing the TTL to up to 4 seconds sees significant, but diminishing, returns.

By default, we base TTL on each function's inter-arrival-time (IAT), rather than have a global fixed time. This method sets the TTL for each flow to the IAT multiplied by the value (α) on the X axis, plotted as the dashed line. For flow timeout (α), our default range is between 1 and 2, which contains the minima, as shown for this particular trace (at 1.5). Higher TTLs ($\alpha > 3$) may result in too many active functions and pressure on the container pool. However, our design is robust to very large TTLs: the pool uses LRU eviction and the resulting impact on performance even at high TTLs is low. *Thus, anticipatory scheduling improves latency by 50%, and MQFQ-Sticky performance is not sensitive to the TTL.*

Device Concurrency. We now explore the tradeoff due to device concurrency, which can improve utilization, but also risks performance interference and is dependent on the co-located applications, which are constantly changing in FaaS workloads. We run our medium workload and adjust device concurrency using D and examine the effect on execution time (without considering queuing delays) in Figure 9b. All numbers are normalized to the no concurrency (i.e. $D = 1$) case. When we increase D and use a fixed value, shown in the gray line, we always have D number of invocations trying to execute on the GPU concurrently. Normalized execution time is unsurprisingly correlated with D, as GPU contention between invocations causes up to 100% overhead at the maximum $D = 6$.



(a) Anticipatory flow keep-alive (non-zero flow TTL) can reduce latency by up to 50%. We use function IAT for scaling the TTL.

(b) Concurrent function invocations (D) increase execution time due to contention. GPU utilization thresholds reduce overload.

(c) Concurrent execution can reduce latency by reducing queuing. However, this is negated by execution interference at higher levels.

Figure 9. Flow TTL and device parallelism (D) are set based on workload and utilization.

By default, we use the GPU utilization upper bound (shown by different lines in Figure9b). Setting the upper bound to 50% utilization prevents over-saturation of the GPU, and reduces overheads significantly. However, this increases the queuing and total latency, as shown in Figure 9c. For this workload, the total latency increases slightly due to contention and interference, and the limited memory of our GPU (16 GB). The "sweet spot" is $D == 2$, and latency increases slightly by 30% at higher levels. Our present design leaves the maximum D to the operator, since the utilization based capping dampens its effect. *Thus, allowing concurrent dispatch to the GPU, controlled to minimize contention, significantly improves global latency and utilization of the device.*

Result: *Our introduced features such as flow over-run, anticipatory scheduling, utilization-driven concurrency, all contribute to latency reduction by* $1.5 - 3\times$. *A wide range of these parameters yield similar performance, making our system robust, yet still providing operators enough flexibility for fine-tuning based on workloads and operational requirements.*

7 Related Work

Locality is an important design and optimization principle in FaaS—and is a fundamental result of code and data initialization required for each function. Keep-alive policies for warm-starts apply temporal locality [25, 58, 61, 66] and caching [28, 65] principles for the CPU memory pool; load balancing also benefits from stickiness [11, 16, 29]. Our work extends these principles to GPU functions via locality enhanced fair queuing and proactive memory management.

GPUs in serverless computing is already a rich and fast-growing area of research. A big portion of prior work [26, 42, 51] focuses on disaggregated accelerators, with GPUs accessed over the network using techniques such as rCUDA [24]. In contrast, we look at local GPUs without remote execution. Using FaaS-inspired abstractions to provide GPU acceleration as a service is also common: applications are broken down into kernels which can be run "anywhere". Kernel-as-a-Service [55] and Molecule [22] are two examples of this approach, where the main challenges are designing and providing efficient and usable API-remoting mechanisms. [39] also uses remote memory pooling to address the exacerbated cold-start problems for GPUs, and also proposes parallel data-dependency and compute context prefetching through code-level optimizations. Paella [52] similarly breaks apart model inference tasks into CUDA kernel launches to minimize scheduling "bubbles". These and other recent [78] specialized code-modifying techniques are orthogonal to our work, since we require general black-box functions.

The popularity of **ML inference** has resulted in a large number of specialized solutions to efficient GPU scheduling, which have similar challenges, but a different optimization spaces: inference resource requirements are much more deterministic [32] and thus amenable to data-driven optimization [14], and the lack of isolation among requests provides many locality-enhancing and batching opportunities [59, 74]. For instance, both FaST-GShare [31] and TGS [71] leverage profiles of ML workloads to monitor GPU utilization and use 2d bin-packing (with time and memory dimensions) to schedule inference workloads.

Finally, **scheduling** is crucial for FaaS performance, with key tradeoffs in late vs. early binding [40, 41]. Efficiency and fairness tradeoffs in GPU scheduling have been recently resolved [49], but only in the offline context with a limited number of batch jobs with known utilities.

8 Conclusion

We showed that harnessing GPU resources for functions is practical and can achieve good performance through the use of locality-aware scheduling and memory management. Black-box containerized functions and heterogeneous and dynamic function workloads present many challenges to efficient GPU utilization. MQFQ-Sticky, our scheduling algorithm, is inspired by I/O fair scheduling. Empirical analysis of its performance indicates it reduces function latency by more than 50× compared to current GPU containers.

References

[1] Best practices for GPU-accelerated instances. https://www.alibabacloud.com/help/en/fc/use-cases/best-practices-for-gpu-accelerated-instances/.
[2] MPS support of Jetson Xavier .

https://forums.developer.nvidia.com/t/mps-support-of-jetson-xavier/62397.

[3] Netflix AWS Lambda Case Study. https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/.

[4] Nvidia management library. https://developer.nvidia.com/nvidia-management-library-nvml.

[5] Project: GPU-Enabled docker image to host a Python PyTorch Azure Function. https://github.com/puthurr/python-azure-function-gpu.

[6] Docker. https://www.docker.com/, June 2015.

[7] Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.

[8] AWS Lambda. https://aws.amazon.com/lambda/, 2020.

[9] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html, 2020.

[10] Cold starts. https://www.alibabacloud.com/help/en/fc/use-cases/quasi-real-time-inference-scenariossection-rzz-zcb-w4e, 2024.

[11] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 365–380, 2023.

[12] Georgios Alexopoulos and Dimitris Mitropoulos. nvshare: Practical gpu sharing without memory size constraints. 2023.

[13] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, Atlanta, GA, USA, November 2020. IEEE.

[14] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.

[15] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[16] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. Beyond load balancing: Package-aware scheduling for serverless platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 282–291, 2019.

[17] Arda Aytekin and Mikael Johansson. Harnessing the power of serverless runtimes for large-scale optimization. *arXiv preprint arXiv:1901.03161*, 2019.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[19] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16, 2017.

[20] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration, 2024.

[21] Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. Understanding the neglected cost of serverless cluster management. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, page 22–28, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM international conference on architectural support for programming languages and operating systems*, pages 797–813, 2022.

[23] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[24] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

[25] Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Journal of Systems Architecture*, page 103115, 2024.

[26] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2022.

[27] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. Ilúvatar: A fast control plane for serverless computing. 2023.

[28] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Alexander Fuerst and Prateek Sharma. Locality-aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 2022, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. Mlless: Achieving cost efficiency in serverless machine learning training. *arXiv e-prints*, pages arXiv–2206, 2022.

[31] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 635–644, 2023.

[32] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[33] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[34] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. {Multi-Queue} fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019.

[35] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[36] Ling-Hong Hung, Dimitar Kumanov, Xingzhi Niu, Wes Lloyd, and Ka Yee Yeung. Rapid rna sequencing data analysis using serverless computing. *bioRxiv*, page 576199, 2019.

[37] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 117–130, 2001.

[38] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion - UCC '19 Companion*, pages 43–50, Auckland, New Zealand, 2019. ACM Press.

[39] Justin San Juan and Bernard Wong. Reducing the Cost of GPU Cold Starts in Serverless Deep Learning Inference Serving. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 225–230, Atlanta, GA, USA, March 2023. IEEE.

[40] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Practical scheduling for real-world serverless computing. *arXiv preprint arXiv:2111.07226*, 2021.

[41] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 289–305, San Francisco California, November 2022. ACM.

[42] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. GPU Enabled Serverless Computing Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 533–540, Cambridge, March 2018. IEEE.

[43] Jiho Kim, John Kim, and Yongjun Park. Navigator: Dynamic multi-kernel scheduling to improve gpu performance. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[44] Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Serverless computing provides on-demand high performance computing for biomedical research. *arXiv preprint arXiv:1807.11659*, 2018.

[45] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[46] Ping-Min Lin and Alex Glikson. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv:1903.12221 [cs]*, March 2019. arXiv: 1903.12221.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[48] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. Cold Start Influencing Factors in Function as a Service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, Zurich, December 2018. IEEE.

[49] Zizhao Mo, Huanle Xu, and Wing Cheong Lau. Optimal Resource Efficiency with Fairness in Heterogeneous GPU Clusters, March 2024. arXiv:2403.18545 [cs].

[50] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Vadim Sukhomlinov, and Naren Nayak. Agile Cold Starts for Scalable Serverless. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, page 6, 2019.

[51] Diana M Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020.

[52] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 595–610, 2023.

[53] Nvidia. Nvidia MPS. https://docs.nvidia.com/deploy/mps/index.html, 2023.

[54] Nvidia. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2023.

[55] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. Kernel-as-a-service: A serverless interface to gpus. *arXiv preprint arXiv:2212.08146*, 2022.

[56] Rajat Phull, Cheng-Hong Li, Kunal Rao, Hari Cadambi, and Srimat Chakradhar. Interference-driven resource management for gpu-based heterogeneous clusters. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 109–120, 2012.

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

[58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[59] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. Efficient gpu sharing for serverless workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, pages 17–24, 2020.

[60] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, April 2021.

[61] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[62] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

[63] Josef Spillner, Cristian Mateos, and David A. Monge. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In Esteban Mocskos and Sergio Nesmachnow, editors, *High Performance Computing*, volume 796, pages 154–168. Springer International Publishing, Cham, 2018.

[64] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. 2024.

[65] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[66] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet of Things Journal*, 10(5):3917–3927, 2022.

[67] Blesson Varghese, Javier Prades, Carlos Reano, and Federico Silla. Acceleration-as-a-service: Exploiting virtualised gpus for a financial application. In *2015 IEEE 11th International Conference on e-Science*, pages 47–56. IEEE, 2015.

[68] 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457, 2021.

[69] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–61, 2023.

[70] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365. IEEE, 2018.

[71] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[72] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71(2):450–463, 2021.

[73] Shinichi Yamagiwa and Koichi Wada. Performance study of interference on gpu and cpu resources with multiple applications. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[74] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[75] Hangchen Yu, Arthur M Peters, Amogh Akshintala, and Christopher J Rossbach. Automatic virtualization of accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 58–65, 2019.

[76] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2017.

[77] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*, pages 61–66, 2019.

[78] Han Zhao, Weihao Cui, Quan Chen, Shulai Zhang, Zijun Li, Jingwen Leng, Chao Li, Deze Zeng, and Minyi Guo. Towards Fast Setup and High Throughput of GPU Serverless Computing, April 2024. arXiv:2404.14691 [cs].