F3: An FPGA-accelerated FaaS Framework

Charalampos Mainas
Technical University of Munich
Munich, Germany

Martin Lambeck
Technical University of Munich
Munich, Germany

Bruno Scheufler
Technical University of Munich
Munich, Germany

Laurent Bindschaedler
Max Planck Institute for Software Systems (MPI-SWS)
Saarbrücken, Germany

Atsushi Koshiba
Technical University of Munich
Munich, Germany

Pramod Bhatotia
Technical University of Munich
Munich, Germany

Abstract

FPGAs provide a programmable, energy-efficient, and compute-intensive acceleration substrate; thus, on the one hand, they offer a compelling solution for optimizing serverless workloads in cloud environments. On the other hand, FPGAs also introduce significant challenges that directly contradict the serverless model in the cloud, including their low-level and complex programming APIs, lack of virtualization and isolation mechanisms, high reconfiguration and communication overheads, and absence of orchestration mechanisms.

To this end, we present F3, the first system that enables efficient and secure use of FPGAs to accelerate serverless functions. F3 allows serverless functions to easily offload their tasks to FPGAs without worrying about low-level device management. The system automatically deploys and invokes FPGA-accelerated functions while providing isolation, high throughput, and low latency. To achieve these design goals, F3 simplifies the complexity of FPGA initialization by a high-level hardware-agnostic API, enables function isolation with unikernel-based FPGA virtualization, reduces cold starts and communication overheads with a per-node FPGA resource manager, and maximizes FPGA utilization with an FPGA-aware orchestrator.

We implement an open-source prototype of F3 based on a practical serverless framework, which consists of OpenFaaS, Kubernetes, and containerd, following the CRI/OCI industry standards for cloud orchestration engines. Our evaluation demonstrates significant latency reduction and throughput improvements for real-world application workloads, microbenchmarks, and Azure production traces.

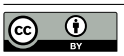
CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

FPGA, hardware acceleration, serverless computing

ACM Reference Format:
Charalampos Mainas, Martin Lambeck, Bruno Scheufler, Laurent Bindschaedler, Atsushi Koshiba, and Pramod Bhatotia. 2025. F3: An FPGA-accelerated FaaS Framework. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3731545.3731582



HPDC '25, July 20–23, 2025, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731582

1 Introduction

Motivation. Function-as-a-Service (FaaS) increasingly dominates cloud computing workloads due to its inherent advantages [10, 17, 50, 87, 92]. FaaS abstracts infrastructure management concerns away from developers, allowing them to focus solely on writing and deploying code, while cloud operators offer a *serverless computing model*. In the serverless computing model, the cloud provider automatically handles scaling, fault tolerance, and resource provisioning. Serverless computing also improves cost efficiency, as it only charges users for the resources consumed during function execution [74, 99].

Developers now use serverless computing for various workloads, including high-performance, compute-intensive tasks such as data analytics, scientific computing, and machine learning/AI [26, 47, 107, 119]. However, running such compute-intensive workloads only on CPUs may not be cost-effective or power-efficient, and integrating accelerators is increasingly becoming a norm to accelerate domain-specific computing.

As various types of accelerators are being adopted in the cloud: GPUs [31, 37], TPUs [51], and Field-Programmable Gate Arrays (FPGAs) [9, 96, 120], a natural question arises: *can we leverage these accelerators to also accelerate serverless workloads?* We are particularly interested in FPGAs, reconfigurable accelerators that can dynamically build application-specific custom logic for acceleration on demand. FPGAs are superior to other accelerators in terms of programmability, flexibility, and energy efficiency [56, 97]. Their ability to be reprogrammed to accommodate new workloads makes them a valuable component in on-demand, serverless environments where workloads shift frequently.

Limitation of state-of-the-art approaches (and challenges). This paper explores the design space of an FPGA-accelerated FaaS platform. Despite the clear benefits of FPGAs, hard-to-reconcile differences between FPGAs and serverless computing present a significant research gap in FPGA adoption for serverless functions. We identify four key challenges that are not fully addressed by the state-of-the-art studies [19, 40].

***First**, there is a mismatch between the serverless computing model and the FPGA execution model.* Offloading tasks to FPGAs through native FPGA libraries [5, 109] requires intricate low-level device management beyond accelerators [94] and complicates serverless execution patterns (e.g., function chain), which contrasts with the simple and easy-to-use serverless computing model. BlastFunction [19], dedicated to OpenCL applications, inherits these limitations. ***Second**, FPGA virtualization support is limited.* Multi-tenancy and isolation are vital to sharing resources and minimizing costs in serverless environments [8, 89]. However, since FPGA usage offloads code and data

outside the sandboxed CPU, sharing FPGAs in a multi-tenant environment requires support for FPGA virtualization, which is often limited. ***Third,*** *reconfiguration and communication overheads of FPGA accelerators negate their performance benefits.* The initialization ('cold' boot) of FPGA accelerators can significantly increase the invocation latency of serverless functions, as it involves reconfiguring the FPGA logic, typically measured in seconds [118]. Molecule [40] statically programs a fixed set of accelerators into FPGAs to avoid frequent initialization, at the expense of their flexibility and reconfigurability. Moreover, the lack of Inter-Process Communication (IPC) primitives for FPGA accelerators becomes a serious performance bottleneck for the serverless model, where typical functions are chained [78, 102] to build realistic applications. ***Fourth,*** *no orchestration support for FPGA accelerators.* FaaS platforms use efficient and battle-tested orchestrators to schedule tasks across distributed machines while maximizing resource utilization [15, 16]. However, reusing these orchestrators with FPGA resources is not ideal due to the latter's lack of preemption and migration capabilities without hardware support in FPGAs. As such, no cloud orchestrator exists for FPGAs.

Key insights and contributions. To overcome these four challenges, we present F3 (**F**PGA-accelerated **F**aaS **F**ramework), the first end-to-end serverless computing framework that supports FPGA acceleration in a convenient and isolated fashion with low latency and high throughput. ***First,*** *the F3 APIs* abstract the complexities of device management and allow CPU functions to interact with FPGA resources transparently. ***Second,*** *the F3 unikernels and hypervisors* let CPU functions run in an isolated, secure, lightweight sandbox to share virtualized FPGAs among multi-tenant workloads. ***Third,*** *the F3 Shell and vFPGA manager* achieve both low invocation/communication latencies and flexible accelerator deployment by leveraging partial reconfiguration and hardware-assisted data transfers. ***Fourth,*** *the F3 orchestrator* deploys functions across distributed nodes in an FPGA locality-aware fashion, minimizing invocation latencies while maximizing resource utilization. This cohesive architecture bridges the gap between FPGAs and serverless, enabling their isolated, efficient, and practical use in serverless computing.

Experimental methodology and artifact availability. We implement an open-source prototype of F3 following Container Runtime Interface (CRI) [34] and Open Container Initiative (OCI) [36] industry standards. The prototype consists of Kubernetes [16], OpenFaaS [79], containerd [46], and kata-urunc [77] upon four-node x86 server clusters with three AMD FPGA cards. We leverage IncludeOS [24] and Solo5 [121] to implement our F3 unikernel and hypervisor. We implement the vFPGA manager that extends the Coyote Shell [67] for low invocation latencies and function chaining. F3 is available as an open-source project [114].

We evaluate F3's effectiveness across three dimensions: performance for real-world applications, latency analysis with extensive micro benchmarking, and scalability analysis with Azure production traces. First, F3 achieves *average speedups of 28.6× (up to 150.3×)* over a CPU-only baseline across a broad range of real-world applications. Next, our micro-benchmarking analysis highlights that F3 has *comparable cold-start latency to standard containers (11.6% increases)* and achieves *1.4× to 1.7× speedups for function chaining* against the OpenFaaS baselines. Finally, using Azure production traces, we demonstrate that F3 can scale to a cluster of 200 machines while

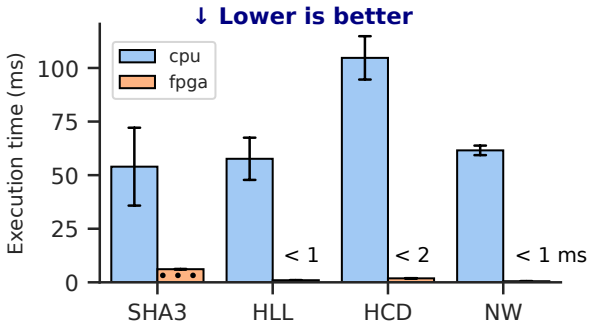


Figure 1: Bare-metal computation times for various workloads executed on FPGA and CPU. The result highlights FPGA's significant performance benefits for compute-intensive tasks. Experimental setups are detailed in § 5.

lowering the end-to-end average latency of function invocations by 3× on production traces.

Limitations of the proposed approach. While F3 aims to address key challenges in FPGA-accelerated serverless computing, its limitations revolve around the effectiveness of its abstractions for all possible serverless workloads. As the evaluation shows, FPGA-based acceleration might not be suitable for all workloads. We also note that our evaluation does not compare F3 with prior studies due to either their unavailable codebase [19] or dedicated hardware requirements [40], while this paper qualitatively differentiates our work from them.

2 Background and Motivation

2.1 Serverless Computing and Hardware Acceleration

Serverless computing enables the deployment and management of applications without the need for traditional server infrastructure [10, 17, 50, 87, 92]. In this model, developers write small and stateless units of code known as *functions* that are executed in an event-driven manner. More complex functionality is achieved by combining different functions, allowing for greater agility and faster time-to-market. As a result, serverless computing provides inherent scalability, high resource utilization, and elasticity by executing functions on demand and utilizing functions' stateless design to scale their execution to handle traffic peaks.

At the same time, the increasing demand for greater computational power in cloud data centers, along with the rise of machine learning workloads, exposes traditional CPU-centric architectures' limitations. With Dennard scaling reaching a plateau [126], general-purpose CPUs can no longer deliver the required performance. In response, cloud providers are deploying hardware accelerators such as GPUs, FPGAs, and ASICs to address the performance requirements [9, 31, 96].

Unfortunately, serverless platforms have yet to integrate hardware accelerators, as the two models present significant differences that are hard to reconcile. The event-driven, auto-scaling, and self-management characteristics of serverless computing, along with the diversity of serverless workloads, make the integration of hardware accelerators into serverless environments challenging [40, 93]. Hardware accelerators must offer fast provisioning and de-provisioning

capabilities to align with the transient nature of serverless applications. Furthermore, ease of management and seamless integration with serverless platforms are essential considerations to simplify development and deployment workflows.

While hardware accelerators come in various forms, from GPUs [7, 49, 54] to ASICs [82, 110, 128], we believe that FPGAs are well-suited to meet serverless computing requirements. FPGAs are reconfigurable, allowing the execution of various serverless applications and catering to their variability. They can be tailored to specific applications, making them well-suited for scenarios where adaptability is crucial [23, 27, 28, 115]. Finally, the potential performance benefits of FPGA over CPU are significant, as demonstrated in Figure 1.

2.2 Cloud FPGA Architecture

This paper targets standard FPGA-equipped cloud infrastructures, where FPGAs serve as standalone PCIe devices installed in CPU servers [122, 124]. A portion of their FPGA fabric statically configures a *Shell*, a set of essential hardware modules such as PCIe DMAs for CPU-FPGA communications [123] and MMUs for memory virtualization [67, 70]. The remaining part is called a *dynamic region*, where users can instantiate (*reconfigure*) their custom logic as *accelerators*.

We adopt standard FPGA programming models to design and develop custom logic on an FPGA. Users write the custom logic code in HDL [21, 112], HLS [64, 73, 109], or DSLs [18, 30, 65] and compile it to a *bitstream* using FPGA development tools such as Vivado [125]. Loading a bitstream to the FPGA reconfigures its dynamic region and instantiates accelerators.

CPU (host) applications are responsible for FPGA device management and task invocation through user-space libraries such as XRT [5] and OpenCL [109]. These libraries offer low-level *FPGA control APIs* that allow CPU applications to perform FPGA reconfiguration, execution, and data transfers.

2.3 Challenges and Key Ideas

Incorporating FPGA acceleration into the serverless computing model presents several challenges. To address this, we identify four core functionalities necessary for successful FPGA adoption. First, *a high-level API* is needed to simplify FPGA device management for developers, maintaining the ease of serverless development. Second, *FPGA isolation and virtualization* are crucial to ensure multi-tenancy within a single accelerator, as the fine granularity of the serverless model forces cloud providers to mix multi-tenant functions to achieve high utilization. Third, it is critical to *mitigate the latency issues* associated with FPGA accelerators to prevent potential increases in latency compared to CPUs. Finally, *an accelerator-aware orchestrator* is required to coordinate the efficient execution of functions on distributed FPGAs within a production-scale cluster.

Next, we break down individual challenges to achieve these functionalities and present our solutions.

#1: High-level API for FPGA abstraction. Serverless computing frees developers from dealing with hardware resources [55, 79]. In contrast, native FPGA libraries [5, 109] only offer low-level APIs that leave FPGA device management to developers. They force developers to carefully describe the execution lifecycle of the accelerators: reconfiguration, data transfers, and task offloading. In addition, these low-level APIs complicate popular serverless execution patterns, i.e., function chains [78, 102]. Therefore, *we need a high-level API for serverless functions to invoke FPGA accelerators, which liberates developers from the burden of managing FPGA devices.*

To address this challenge, we design high-level and easy-to-use FPGA control APIs that fit the serverless computing model. The APIs abstract the complex FPGA execution flow from cloud users and let the underlying system components transparently handle the low-level FPGA device management (§ 3.2).

#2: FPGA virtualization for multi-tenancy. Serverless platforms deploy multi-tenant functions on the same node while isolating them to maximize resource utilization and reduce running costs [75]. Hardware acceleration complicates the isolation as user function code and data are distributed across different devices. Moreover, FPGAs lack hardware support for multiplexing and isolation, presenting a severe challenge to their use in a multi-tenant cloud. Therefore, *we need a new isolation and virtualization mechanism that protects user code and data distributed across CPUs and FPGAs.*

We design a lightweight FPGA virtualization mechanism that isolates CPU functions and FPGA accelerators. A unikernel-based sandbox isolates CPU functions, while FPGA accelerators are maintained by a per-node resource manager that leverages the Shell (§ 3.3).

#3: Invocation latency and communication overheads of FPGA accelerators. Serverless functions are sensitive to cold start delays and communication latencies [41, 60, 117]. These challenges are more significant for FPGA accelerators. First, FPGA reconfiguration, which can reach seconds [118], exacerbates the start-up latency. Prewarming [20, 84, 101, 131] is a well-known approach to eliminate cold starts, but it is not easily adapted to FPGAs. Molecule [40] combines multiple accelerators into a single bitstream for pre-caching. However, this approach leads to poor flexibility (only a fixed group of accelerators can be used at a time) and significant cache miss latencies. Second, FPGAs can incur high communication overheads for function chains [39, 78, 102], building a complex workload by a composition of functions. While CPU functions leverage the IPC mechanism to mitigate the associated communication overheads [102], FPGA accelerators cannot communicate directly without modifying the hardware design, forcing them to do extra hops to transfer data from one to another (e.g., through CPU memory). Therefore, *we need a mechanism to mitigate function invocation latencies and communication overheads associated with FPGA hardware management.*

We design an overhead-less FPGA management mechanism for on-demand accelerator invocation, which leverages dynamic partial reconfiguration to guarantee accelerator isolation while preserving flexibility. It offers three features to reduce accelerator invocation and communication latencies (§ 3.4).

#4: Orchestration for distributed FPGAs. Orchestration is pivotal for serverless environments as it manages function deployment, coordination, and scaling [10, 16, 50, 87]. Orchestration for FPGA applications is far more complicated than CPU-only applications; FPGAs are prone to race conditions due to their lack of task preemption support, high reconfiguration overheads, and limited resources. Unfortunately, to our knowledge, no work exists that orchestrates and schedules FPGA functions or determines which FPGA-specific metrics are helpful for deployment decisions. Therefore, *we need a new orchestration mechanism that considers FPGA characteristics and ensures fairness while maximizing the overall FPGA utilization.*

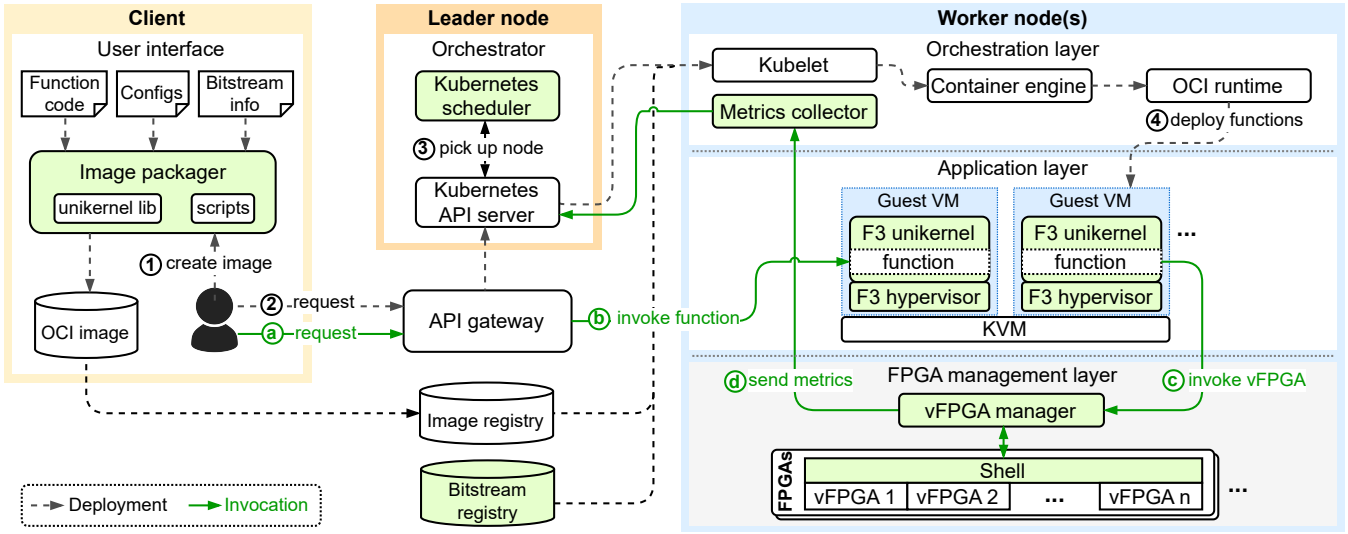


Figure 2: High-level overview of the F3 architecture, illustrating the interaction between client, application, orchestration, and FPGA management layers for function deployment and invocation. Key components are highlighted in green.

We design an FPGA-aware orchestrator to fairly distribute FPGA functions across worker nodes. Our extended Kubernetes scheduler measures and collects per-node FPGA usage metrics to enhance placement decisions (§ 3.5).

Summary. This paper tackles these challenges by building F3, the first end-to-end FaaS framework that offers all the core functionalities for FPGA adoption into serverless domains. While a few existing studies [19, 40, 100] partially address them, to our knowledge, F3 is the first system that brings comprehensive and practical solutions to all the challenges.

3 Design

3.1 System Overview

Figure 2 illustrates a high-level view of the F3 architecture. We target a Kubernetes-based cluster as an industry-standard serverless environment, where Kubernetes runs on the *leader node* as an orchestrator and manages multiple *worker nodes* equipped with PCIe-connected FPGA cards. *Clients* (users) send requests to deploy/invoke FPGA-accelerated functions to Kubernetes through the API gateway, e.g., OpenFaaS [79].

F3 system components. We highlight key system components shown in Figure 2. The *image packager* resides on the client node and offers user-space libraries for *F3 APIs* (§ 3.2) and scripts to let users build serverless functions. It packages the functions as Open Container Initiative (OCI) images [35] stored in the image registry. FPGA bitstreams are maintained in the *bitstream registry*. On the worker node, the functions are running within guest sandboxes offered by the *F3 unikernel* and *F3 hypervisor* (§ 3.3). The unikernel encapsulates CPU functions to ensure isolation in a multi-tenant cloud. The hypervisor launches the unikernel and communicates with the *vFPGA manager* to handle FPGA-related requests from the guest function. Each FPGA device configures the *F3 Shell* (§ 3.4) on their static region, which splits the FPGA fabric into multiple dynamic regions (vFPGAs) for concurrent execution and FPGA sharing. The vFPGA manager maintains vFPGAs and reconfigures/invokes the accelerators upon requests from CPU functions. The *orchestrator* (§ 3.5), i.e., Kubernetes, provisions and manages hardware resources in a cluster while cooperating with worker-node system components. The extended *Kubernetes scheduler plugin* performs FPGA-aware orchestration with FPGA usage metrics obtained from the *metrics collector* that tracks FPGA behaviors at the execution.

F3 system workflow. The serverless system workflow is comprised of two phases: *deployment* and *invocation*. In the *deployment* phase, users prepare CPU function code and serverless configuration file (step ①). The function code is built with the F3 unikernel library, making the code deployable on worker nodes. The user can then deploy the application, sending the request to the API gateway (step ②). Upon the requests, the Kubernetes scheduler is invoked and decides on which node the function is to be deployed (step ③). The decision is notified to system components on the respective node, and the OCI runtime spawns the function (step ④).

In the *invocation* phase, a user makes an invocation request (e.g., HTTP) to an endpoint created by the API gateway (step ⓐ). The gateway forwards the request to the watchdog running in the F3 unikernel, which invokes the function execution with the relevant arguments (step ⓑ). The function can also invoke FPGA accelerator execution by calling F3 APIs (step ⓒ). Upon completion of function execution, the watchdog returns the results to the API gateway, which forwards them to the user. Meanwhile, the Metrics collector periodically measures and calculates the FPGA's usage statistics and informs the Kubernetes scheduler (step ⓓ).

3.2 F3 API for FPGA Acceleration

We first introduce F3 APIs. The F3 APIs provide high-level and easy-to-use abstractions for serverless functions to execute on FPGA accelerators. The APIs abstract away the complexity of FPGA device management from users and delegate the responsibility to F3 system

F3 APIs	Description
void* alloc_fpga_buffer(size)	allocates input/output buffers.
int call_fpga_acc(acc_id, inputs, outputs, args)	invokes an FPGA accelerator.
int wait_fpga_acc(call_id)	waits for a job completion.

Table 1: The definition of F3 APIs.

```
void fpga-acc(uint64_t key_high, uint64_t key_low) {
  void* in  = alloc_fpga_buffer(INPUT_SIZE);
  void* tmp = alloc_fpga_buffer(INPUT_SIZE);
  void* out = alloc_fpga_buffer(OUTPUT_SIZE);

  int id = call_fpga_acc("gzip", in, out, null);
  wait_fpga_acc(id); /* single */

  call_fpga_acc("gzip", in, tmp, null);
  id = call_fpga_acc("aes", tmp, out, {key_high, key_low});
  wait_fpga_acc(id); /* chained */
}
```

Listing 1: A code example using F3 APIs.

components. They also support modern features of existing FaaS frameworks, e.g., function chain [105], and make it easy to integrate F3 functions into existing API gateways such as OpenFaaS [79].

API description. F3 APIs are accelerator-agnostic APIs that enable CPU functions to control any FPGA accelerators. Unlike low-level FPGA control APIs [5, 109], F3 APIs delegate jobs for FPGA initialization, e.g., acquire a free vFPGA, program a bitstream, and configure registers to the serverless backend. Moreover, F3 supports chaining multiple invocations of different FPGA accelerators, which is called an *accelerator chain*. The accelerator chain allows the output streams of an accelerator to be directly forwarded to another as input streams. It effectively works for services that consist of multiple FPGA-accelerated functions; it not only mitigates the complexity of programming inter-function communications but also reduces associated performance overheads.

Table 1 shows the F3 API definition. F3 offers only three primitive APIs that are essential for task offloading: memory allocation, accelerator invocation, and synchronization. alloc_fpga_buffer() creates FPGA-accessible data buffers in guest memory space. These buffers need to be initialized for input and output data of FPGA accelerators before the accelerator invocation. call_fpga_acc() invokes the execution of the selected FPGA accelerator. The accelerator, input buffers, and output buffers are specified as *acc_id*, *inputs*, and *outputs*, respectively. *args* represents optional parameters configured on the accelerator's control registers. Importantly, this API can *chain* FPGA accelerators; it builds a directed acyclic graph (DAG) based on data dependency of accelerator invocations, i.e., whether multiple API calls specify the same input/output buffer. wait_fpga_acc() lets CPU functions wait for FPGA accelerator execution, which is specified by a return value of call_fpga_acc(), i.e., *call_id*.

While F3 APIs offer a unified interface for managing various FPGA accelerators, some accelerators require unique invocation and synchronization sequences, e.g., specific orders of read/write accesses to control registers. To abstract these accelerator-specific workflows, F3 enables accelerator developers to register two low-level functions: *invoke(args)* and *wait()*, which encapsulate accelerator-specific

```
fpga-acc:
  fpga-resources:
    alveo-u50/f3-shell:2
  fpga-accelerators:
    gzip: "gzip_accelerator:v1.0"
    aes:  "aes_accelerator:v1.0"
```

Listing 2: An example of the F3 function configuration.

invocation and synchronization logic. They serve as dynamically loadable libraries for the vFPGA manager. Given these functions, call_fpga_acc() and wait_fpga_acc() APIs act as their wrappers. The vFPGA manager dispatches these high-level API calls from guest functions to the low-level functions for the selected accelerator.

Code example. Listing 1 shows an example function code, fpga-acc, that uses two accelerators, gzip and aes. It demonstrates two execution modes: single and chained accelerators. For both modes, the function needs to prepare input/output buffers before invocation (L2-4). When the function uses only a single accelerator, it simply invokes the accelerator (L6) and waits for completion (L7). When the function wants to chain two or more accelerators, it invokes the accelerators with shared buffers to pipe the output of one accelerator into the input of the other (L9-10); *tmp* is shared by two accelerator invocations in this example.

Function configuration. To deploy F3 serverless functions, the orchestrator needs to know their hardware requirements (e.g., the number of FPGAs) so that it properly manages the underlying FPGAs on behalf of users. F3 adopts the same approach as other serverless frameworks [10, 79], configuration files where users describe the hardware requirements. Listing 2 shows an example of the configuration file for the fpga-acc function (Listing 1). F3 offers two new fields for the FPGA management. The fpga-resources field (L2-L3) defines the type and number of vFPGAs. The fpga-accelerators field (L4-L6) defines accelerator names as labels and corresponding bitstreams. These labels (gzip, aes) are used as *IDs* when the function invokes these accelerators. The function code and configuration file are packaged as an OCI image [35] and uploaded to the image registry as well as other FaaS frameworks [10, 79, 92].

Bitstream registry. F3 offers the bitstream registry, where cloud users and developers store the bitstreams of their FPGA accelerators. The bitstream registry allows third-party IP vendors to register and publish their accelerators like Vitis Accelerated libraries [13] so that users do not have to design the hardware logic by themselves. Moreover, it facilitates multiple serverless functions to share the same accelerators, which prevents frequent FPGA reconfiguration. Users can select accelerators to be used from the bitstream registry and describe them in the configuration file. Appropriate bitstreams are loaded with the function's image in the deployment phase.

3.3 F3 FPGA Virtualization

We next introduce F3's FPGA virtualization designed for latency-sensitive serverless functions. F3 uses unikernels as a guest CPU sandbox. Unikernels are lightweight VMs that contain only essential OS components to execute the target application [83]. Their fast boot times and small memory footprint make them a good candidate for serverless environments, where scalability and responsiveness are critical. In order to facilitate an FPGA serverless system on top of

unikernels, we design the F3 unikernel, F3 hypervisor, and vFPGA manager. The hypervisor manages the life cycle of unikernels, and the vFPGA manager securely handles FPGA acceleration requests from multiple guests and manipulates the underlying FPGA device. The vFPGA manager also leverages Shells [61, 67, 70] to guarantee isolation for FPGA accelerators.

F3 unikernel and hypervisor. Figure 3 shows F3's virtualization system stack. The F3 unikernel follows an event-driven execution model for serverless functions, hosting an HTTP server called *watchdog* to receive and send serverless requests through the API gateway. Upon receiving a function invocation request, the watchdog triggers the *guest function* written by users, passing any necessary arguments and data for the function. The *acceleration library* exposes F3 APIs (§ 3.2) to the guest function for buffer allocation and accelerator invocation. For buffer allocation, the acceleration library creates *user buffers* that are mapped to hypervisor-side *host buffers*. The host buffers are accessible from vFPGAs by the DMA controller. For accelerator invocation, it communicates with the guest driver to issue hypercalls offered by the F3 hypervisor.

The F3 hypervisor is a unikernel monitor [121], a host process serving as a thin hypervisor layer for F3 unikernels. The hypervisor is assigned to each unikernel and launches the corresponding unikernel during function deployment. For security guarantees, the hypervisor does not allow guest unikernels to manage the FPGA device directly but offers hypercalls to forward their requests to the vFPGA manager.

vFPGA manager. We design the vFPGA manager, a per-node centralized FPGA resource manager that fairly and safely handles and executes FPGA acceleration requests from multi-tenant functions. The vFPGA manager schedules incoming requests on the node and executes them accordingly with the help of the underlying FPGA Shell. The centralized manager fits the FPGA serverless scenario, where deployed functions randomly send FPGA acceleration requests in an event-driven way to use the limited number of FPGA resources.

The *scheduler* of the vFPGA manager is responsible for initializing a vFPGA upon a new request. The *bitstream cache* is in-memory storage that keeps bitstreams of accelerators specified by the function description (§ 3.2). The scheduler can program the selected bitstream to vFPGAs using the *Dynamic Partial Reconfiguration (DPR) controller* offered by the Shell. Importantly, the scheduler does not reorder requests and follows an FCFS approach, while it aims to avoid frequently reconfiguring vFPGAs to reduce the start-up latency by reusing accelerators that are already configured on vFPGAs (described in § 3.4). After the accelerator is ready to execute, the *request handler* manipulates the dedicated DMA controller offered by the Shell to invoke the accelerator. The request handler specifies the addresses of host buffers as input/output of the accelerator, as they are accessible from the FPGA.

3.4 F3 Shell Management

We introduce the F3's Shell architecture and FPGA management that mitigates the invocation and communication latencies of FPGA accelerators. The Shell management enables three key features: accelerator reuse, parallelization, and accelerator chaining.

Architecture. Figure 4 illustrates the F3 Shell architecture. It follows the architecture of Coyote [67], but F3 can also adopt other

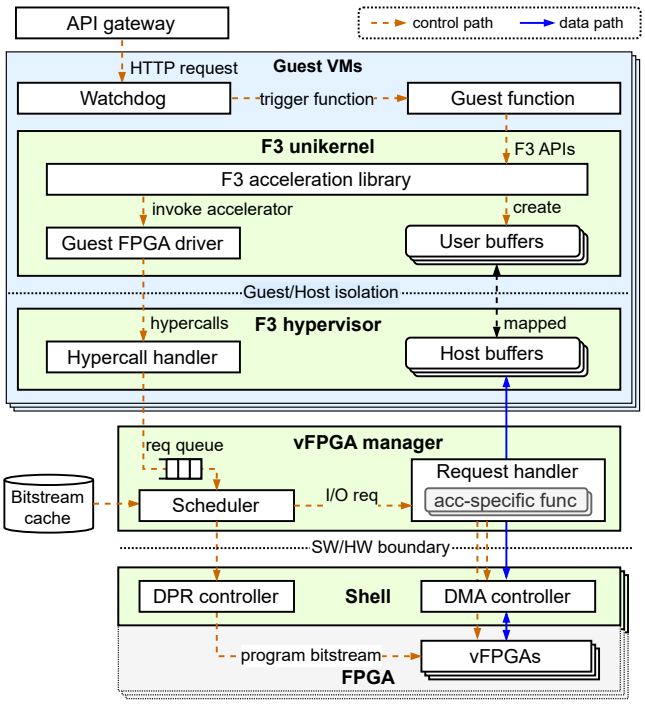


Figure 3: Detailed view of the F3 virtualization stack, showing the interaction between guest VMs, the vFPGA manager, and FPGAs. The data and control paths include hypercalls, DMA operations, and FPGA-specific request handling.

Shells offering multiple vFPGAs and memory isolation (e.g., AmorphOS [61], FSRF [70]). In F3, the vFPGA manager is responsible for manipulating the Shell. The PCIe DMA IP exposes memory-mapped I/O interfaces to the vFPGA manager for access to control and status registers (CSRs) and enables data transfers between host memory and accelerators on vFPGAs. The DPR controller allows the vFPGA manager to program a bitstream to any vFPGA. MMUs are assigned to every vFPGA and manage a virtual memory space for memory isolation for accelerators. The MMUs are also responsible for issuing read/write requests to the DMA controller, which is triggered by the vFPGA manager through CSRs.

The F3 Shell leverages partial reconfiguration, which causes a fabric fragmentation issue [61, 134]. It is supposed to be manageable for serverless functions that are small pieces of cloud services. Assuming they are already partitioned into small tasks, we expect that FPGA accelerators for serverless functions are small, independent computation logic rather than offloading the entire application. Moreover, FPGA partitioning reduces the invocation latency since the smaller FPGA area leads to faster reconfiguration [67].

Molecule [40] proposes a different approach to mitigate accelerator invocation latencies, caching multiple accelerators within a single bitstream. However, this approach compromises FPGA flexibility and can lead to underutilization. Cached accelerators cannot be released until all accelerators programmed on the FPGA are inactive, potentially causing significant cache miss latencies. Furthermore, it is impractical to pre-build a bitstream for any combination of users' custom accelerators, as even a single-bitstream generation

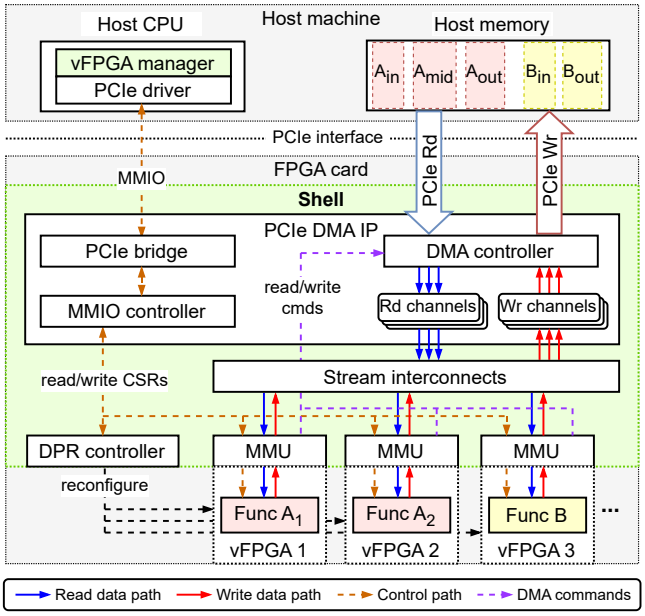


Figure 4: F3 Shell architecture, where two functions are deployed: A using two chained accelerators (*Func* A_1 and A_2) and B using one accelerator (*Func B*).

takes several hours [104]. Instead, F3 individually prepares single-accelerator bitstreams for dedicated DPR regions. Once generated, these bitstreams can be reused across the same type of FPGAs.

Accelerator reuse. Since F3 adopts an on-demand invocation of FPGA accelerators, FPGA reconfiguration significantly exacerbates the overall invocation latency. To eliminate the reconfiguration, the vFPGA manager enables temporal sharing of the already-configured accelerator on the vFPGA. In particular, the scheduler keeps track of bitstreams programmed to vFPGAs and, if an incoming request targets the existing accelerator, schedules the request to the corresponding vFPGA. Before assigning the new task, the vFPGA manager flushes the MMU and accelerator states to guarantee data isolation.

Parallelism. F3 assumes CPU functions store input/output data in the host memory. To allow multiple accelerators to be executed in parallel while sustaining the PCIe throughput, F3 adopts channel interleaving [123], which offers multiple read/write channels for each vFPGA and processes multiple DMA requests in parallel.

Accelerator chaining. In a typical serverless environment, chained functions are deployed to the same worker node and communicate with each other through IPC mechanisms to avoid expensive communications over the network [102, 133]. Similarly, for chained FPGA accelerators in our F3 framework, instead of going the long path of IPC between CPU function instances, the vFPGA manager lets the accelerators directly pass input/output data within the FPGA device. In particular, the vFPGA manager prepares one temporal buffer (A_{mid} in Fig. 4) and sets the buffer as the output of the leading accelerator (*Func* A_1) and as the input of the following accelerator (*Func* A_2). As soon as the first accelerator finishes, the vFPGA manager invokes the next accelerator in the chain.

Algorithm 1: The scoring algorithm of our scheduler.

```
1  schedule(pod, nodes, fpga_usages, reconf_times, bs_locations)
2  begin
       /* Find the most suitable node by scoring nodes */
3      picked_node ← null;
4      foreach node in nodes do
5          best_score, curr_score ← 0,0;
6          curr_score ← score(node,
             fpga_usages, reconf_times, bs_locations, pod);
7          if curr_score > best_score then
8              best_score ← curr_score;
9              picked_node ← node;
10     end
11     return picked_node;
12 end
13 score(node, fpga_usages, reconf_times, bs_locations, pod)
14 begin
       /* Score the node based on FPGA metrics */
15     score_u ← 1− (fpga_usages.index(node) / fpga_usages.size());
16     score_r ← 1− (reconf_times.index(node) / reconf_times.size());
17     score_l ← 0;
18     if bs_locations.find(node, pod.bitstream) == true then
19         score_l ← 1;
20     return (score_u × w_u) + (score_r × w_r) + (score_l × w_l);
21 end
```

3.5 F3 Orchestrator

We design an FPGA-aware orchestrator by extending the Kubernetes scheduler plugin [15] and adding a mechanism to monitor FPGA usage characteristics, i.e., the metric collector. Our orchestrator aims to achieve two goals: (1) minimizing the function invocation latency from the customer's viewpoint while (2) maximizing FPGA utilization from the cloud provider's viewpoint. Importantly, to minimize the invocation latency, our orchestrator minimizes the reconfiguration overheads by taking the *locality* of accelerators into account; if it receives multiple function invocation requests using the same accelerator, it attempts to deploy them on the same worker node so that the accelerator gets more chances to be reused without reconfiguration (§ 3.4).

Kubernetes cluster extension. Due to CRI/OCI compatibility, we have adapted Kubernetes for our FPGA-aware orchestration with two simple changes. First, our F3 cluster deploys the metrics collector, a worker-node daemon process that receives FPGA usage data from the vFPGA manager and reports it to the scheduler plugin. The scheduler uses the aggregated metrics to predict the frequency and duration of FPGA reconfigurations, cold starts, and resource utilization trends.

Second, the Kubernetes scheduler plugin [15] selects the most feasible node to place a set of functions (called *pod*) on by *filtering* and *scoring* all the worker nodes. We propose a new plugin extending the filtering and scoring steps to improve the invocation latency and FPGA utilization. In the filtering step, the plugin eliminates nodes that do not meet FPGA-related criteria required by the pod, in addition to CPU and memory requirements. In the scoring step, the plugin computes a score for each feasible worker node based on the reported metrics.

Scoring algorithm. Algorithm 1 details the scoring algorithm of our scheduler plugin. The scheduler picks up a pod to schedule next in an FCFS manner. It ranks each node in the node list, which contains all the feasible nodes after the filtering step, based on the score function

Spec.	Leader node	Server node
CPU	Intel(R) Xeon(R) Gold 5317 12 cores, 3.0GHz	Intel(R) Xeon(R) Gold 6238R 28 cores, 2.2GHz
OS	NixOS 23.11, Linux 6.8.10	Ubuntu 20.04, Linux 5.8.0
Memory	256GiB DDR4, 3200MHz	
Network	QSFP28 (100GbE)	
FPGA	-	1 × AMD Alveo U50 card

Table 2: Server cluster configuration.

(L.6). We design the score as a weighted average of multiple criteria periodically measured by the metrics collectors. F3 specifically uses three metrics: *FPGA usage time*, the time vFPGAs are occupied, *reconfiguration time*, the time spent for FPGA reconfiguration, and *bitstream locality*, a boolean value representing if the node has at least one vFPGA where a bitstream specified by the job is already programmed. The function first computes the scores for each metric (the best value receiving a score of 1), then weighs each metric's score by configurable weights (w_u,w_r,w_l) and computes the average (L.15-20). The system administrator can adjust each weight to reflect the desired trade-off between the different metrics. Note that lower vFPGA occupancy gives a higher score, so idle worker nodes can likely be chosen to improve FPGA utilization. Finally, the scheduler picks the node with the highest score to place the application (L.11).

4 Implementation

We implement our F3 prototype using industry-standard serverless system components: OpenFaaS [79] as the API gateway, Kubernetes [16] as the orchestrator, containerd [46] as the container engine, kata-urunc [77] (kata-containers [3] modified to spawn unikernels) as the OCI runtime. We use a default registry of containerd [46] for the image and bitstream registries. We target Alveo U50 FPGA cards [122] for acceleration.

F3 API and F3 unikernel. We implement the F3 API and F3 unikernel on top of IncludeOS [24]. There are two key components: the OpenFaaS watchdog and the acceleration library. The watchdog is an HTTP server that monitors the health and responsiveness of guest functions and reports them to OpenFaaS. The acceleration library implements the F3 APIs and performs the respective hypercalls to the F3 hypervisor. These components are directly linked against IncludeOS, which provides essential OS functions such as memory management and a TCP/IP stack.

F3 hypervisor. We implement the F3 hypervisor on top of Solo5 [121], a thin hypervisor layer on top of KVM. Solo5 creates a VM sandbox for each F3 unikernel and manages their execution lifecycle. Solo5 offers a custom PIO/MMIO-alike interface for network and file I/Os. We implement new hypercalls to Solo5 to provide FPGA device access to the F3 unikernel. The F3 hypervisor communicates with the vFPGA manager over a UNIX socket to handle FPGA-related hypercalls.

F3 Shell and vFPGA manager. Our prototype adopts Coyote [67] as the Shell for FPGA management, customized for Alveo U50 FPGA [122]. We disable all the external I/O interfaces (onboard memory, network) while enabling multiple vFPGAs and DPR. The number of vFPGAs is set to two due to the limited capacity of the U50 FPGA.

We implement the vFPGA manager as a host process interacting with the Coyote Shell through its kernel driver. Every F3 hypervisor establishes a connection with the vFPGA manager through the UNIX socket and maintains this connection till the end of the F3 unikernel execution. The vFPGA manager accepts three commands: a) *load_bitstream()* for informing the vFPGA manager of the bitstream of the guest function, b) *map_memory()* specifies the memory location of input/output buffers of the function, and c) *exec()* places the acceleration task in the selected vFPGA.

Application	Description	Bitstream
ADDMUL [52]	Integer arithmetic	~7.5 MB
AES [52]	AES-128-ECB encryption	~11.1 MB
SHA3 [91]	SHA3-512 hashing algorithm	~8 MB
GZIP [12]	GZIP compression	~12.4 MB
NW [44]	All pairs Needleman-Wunsch	~15.9 MB
HLS4ML [69]	Convolutional neural network	~16.5 MB
HLL [52]	Hyperloglog cardinality estimation	~14.9 MB
HCD [13]	Harris corner detection	~10.2 MB
MD5 [90]	MD5 brute force	~8.9 MB
FFT [11]	Auto correlation via FFT	~14.8 MB

Table 3: Benchmark and application summary.

F3 orchestrator. We use the Kubernetes Scheduling Framework [15] to implement a custom scheduler plugin that enables FPGA-aware scheduling. The custom plugins are implemented as Go modules that are compilable into the scheduler binary. The scheduler plugin attaches to the *Filter*, *PreScore*, *Score*, and *NormalizeScore* extension points that the scheduling framework exposes. The metrics collector is also implemented as a Go module. Metrics updates are synchronized with the Kubernetes scheduler by updating node annotations using the Kubernetes API.

5 Evaluation

We next comprehensively evaluate F3 through a multifaceted approach that includes end-to-end benchmarks (§5.1), microbenchmarks (§5.2), and real-world production traces (§5.3).

Testbed. We perform the experiments on our server cluster, which consists of one leader node and three worker nodes. Table 2 shows our server configurations. Every worker node is equipped with a single Alveo U50 card. We configure the Coyote Shell on each FPGA, which serves two vFPGAs.

Application benchmarks. Because there are no benchmark suites for FPGA-accelerated serverless functions, we use a representative subset of real workloads accelerated by state-of-the-art FPGA implementations (Table 3). Our workloads cover a broad range of application types: matrix operation (ADDMUL), data compression (GZIP), data analytics (NW, HLL), machine learning (HLS4ML), and image processing (HCD, FFT). These classifications are also shown in CPU-oriented serverless benchmark suites [62, 133].

Azure production traces. For large-scale design exploration, we leverage the real-world production traces from Microsoft Azure [138]. While the original traces focus on CPU functions only, we reasonably preprocess the traces with the measured invocation latency of our FPGA workloads listed in Table 3 on our testbed and simulate a large-scale FPGA-equipped cluster (detailed in § 5.3.1).

5.1 Overall Performance

We first broadly evaluate how F3 performs when executing applications representative of typical serverless environments.

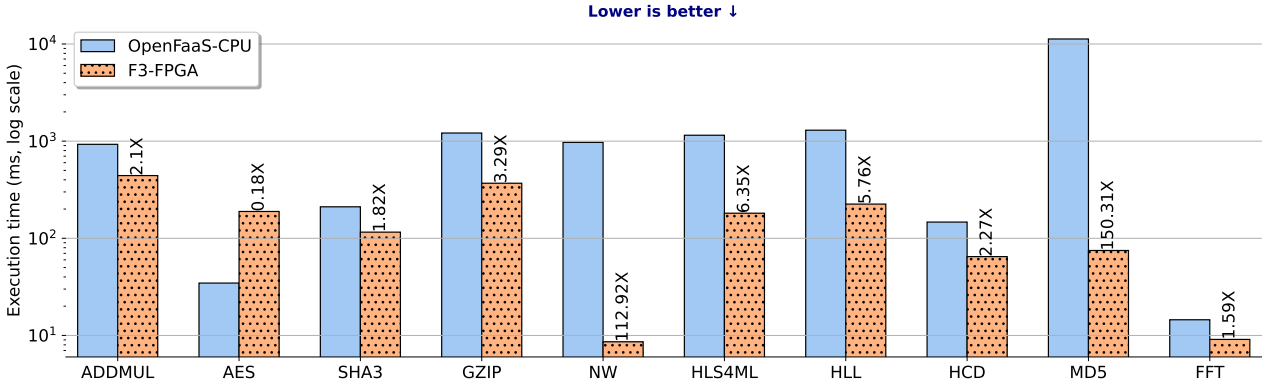


Figure 5: End-to-end function execution time on F3 and OpenFaaS (CPU). The numbers show the speedup by FPGA acceleration.

Single function performance. We begin with a microbenchmark comparing the performance of individually executing each function with FPGA on F3 (*F3-FPGA*) against a CPU-only baseline where the function executes inside a container (*OpenFaaS-CPU*). For the CPU baseline, we build a container based on the Alpine container image [57] and package each function and OpenFaaS's watchdog inside the container. We assume the case where both CPU and FPGA are warmed up and avoid cold starts. We repeat 20 executions for each application and calculate their mean values.

Figure 5 presents the results. F3 achieves 28.6 × average speedups and outperforms CPU-only execution in 9 out of the 10 applications (from 1.6 × to 150.3 ×). F3 significantly improves the performance of compute-intensive tasks (NW, HLS4ML, HLL, MD5) because they can benefit from FPGA acceleration. Relative speedups for SHA3, HLL, and HCD are smaller than the bare-metal computation (Figure 1) because the end-to-end execution time involves common jobs for the two settings, such as invocation request handling and data preprocessing on CPU (e.g., JPEG decoding for HCD). We observe that only AES is slower on F3 than the CPU baseline (0.18 ×) because the CPU baseline performance is highly optimized with AES-NI instructions of Intel processors [58]. *In summary,* F3 brings the significant speedup benefits of FPGA acceleration in a serverless situation.

Function invocation latency. Next, we evaluate the end-to-end latency of function invocations in cold and warm boot scenarios, comparing them as before to the CPU-only baseline. We use SHA3 to measure the invocation latencies. We minimize the input dataset size for SHA3 and eliminate its computation time to highlight the overheads induced by the system components. In the FPGA case, we additionally measure the latency for cold and warm boots combined with whether or not there is a need to reconfigure the FPGA.

Table 4 shows the invocation latencies. First, F3 offers comparable cold-boot latencies to the CPU baseline. In the cases of Cold Boot (CB) and CB with Reconfiguration (CBR), F3 increases 9.3% and 15.6% of the invocation latency compared to the CPU-only invocation, respectively. The overheads mostly come from F3's FPGA initialization steps and reconfiguration. These FPGA-specific overheads are reasonable and not dominant in the end-to-end tail latency. On the other hand, in the case of warm boots, we observe that F3's on-demand FPGA reconfiguration affects the tail latency. In a Warm Boot (WB) scenario, F3 induces even smaller invocation latencies than CPU-only execution. Meanwhile, in a WB with Reconfiguration (WBR) scenario, the reconfiguration overhead is dominant in the latency. We note that the WBR latency is still much less than CBR, i.e., only 7.3% in comparison. We address the reconfiguration overhead by applying a further optimization, *accelerator reuse*, to reduce the number of reconfigurations (§ 5.2). *In summary,* F3 induces comparable cold-boot and warm-boot latencies against the CPU baseline if we can suppress the FPGA reconfiguration.

5.2 Microbenchmarks

In this section, we empirically evaluate different aspects of the F3 system stack using a series of microbenchmarks.

Virtualization overhead. We first explore the overhead imposed by our virtualization mechanism from the application point of view. We use ADDMUL to measure the time required to complete each F3 API with and without hypervisor virtualization (*unikernel* and *native*). We omit the call of alloc_fpga_buffer() because it does not induce hypercalls.

Figure 6 compares the time taken by the API calls. Our F3 virtualization introduces μs-order overheads (10.8 μs, 4.6 μs) compared to the corresponding native execution. This overhead can be attributed to the virtualization stack's additional layers. The primary factor contributing to the overhead is the hypercall from the F3 unikernel to the hypervisor, which takes around ~ 5 μs to complete. Note that the overhead from these three hypercalls, even combined, is negligible in practice compared to the application runtime, which averages 167.9 ms (8.6-368.8 ms range) for the representative applications in Table 3. *In summary,* F3 induces negligible virtualization overheads at the function invocation.

Cold boot time. We next break down the overhead of cold boot in F3. We use ADDMUL and measure the time a client requests a deployment until the function becomes responsive. We then break down the overhead into each step: service image fetching, worker-node environment setups, and VMM/OS setups. We compare F3 with two different container settings: Docker runtime (*Docker*) and Kata Containers runtime (*Kata containers*). Note that Kata containers runtime deploys a container in a Firecracker microVM [6].

Figure 7 presents the cold boot analysis over the three settings. Overall, F3 induces reasonable cold boot overheads compared to Docker (11.6% increases), which lacks hypervisor isolation, and faster

Device	WB	WBR	CB	CBR
CPU	0.015 s	-	1.621 s	-
FPGA	0.007 s	0.141 s	1.787 s	1.921 s

Table 4: Invocation latency of SHA3 in different scenarios: Warm Boot (WB), WB with Reconfiguration (WBR), Cold Boot (CB), and CB with Reconfiguration (CBR).

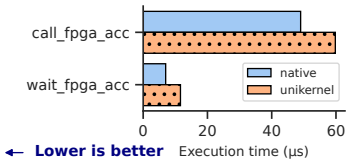


Figure 6: Unikernel overhead.

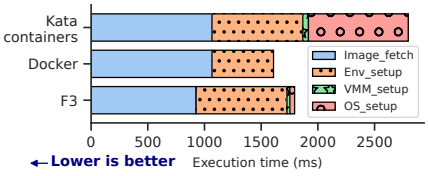


Figure 7: Cold boot analysis.

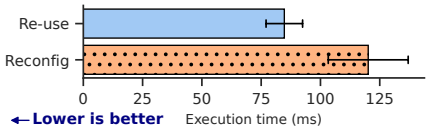


Figure 8: Reconfiguration overhead.

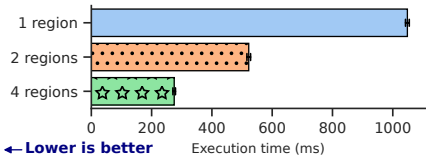


Figure 9: Parallelism of vFPGAs.

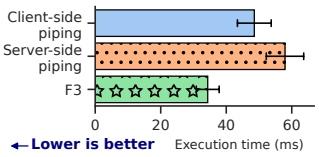


Figure 10: Function chaining.

boot times than the Kata container (35.8% reduction). In F3, the time for fetching the image (image_fetch) is reduced by 13.1% compared to the other runtimes due to the smaller storage footprint of the F3 unikernel. We also observe that the F3 unikernel significantly reduces the guest OS setup time (OS_setup) by 95.3% compared to Kata containers that use a standard Linux kernel image. Likewise, F3 reduces the VMM setup time by 47.8% thanks to its thin hypervisor layer. *In summary,* F3 offers strong hypervisor-level isolation with reasonable cold-boot latencies and is superior to the container-in-VM approach.

Accelerator reuse. We examine the effectiveness of the accelerator reuse function for mitigating the invocation latency in a warm-boot scenario (WBR in Table 4). To measure the reconfiguration overhead, we execute SHA256 [14] on an FPGA with 16 MB of random input data in two scenarios: Reconfig including a forced reconfiguration ahead of each invocation, and Re-use reusing the bitstream from the previous invocation.

Figure 8 shows the results, omitting the bitstream file I/O overheads. We observe that our accelerator reuse function reduces the invocation latency by 35.4 ms. We note that the reduction becomes more significant if we include the file I/O overheads, which take up to several hundred milliseconds depending on memory and cache activities. *In summary,* F3's accelerator reuse function effectively works for millisecond-order invocation latencies of serverless functions.

Parallelization. We investigate F3's performance of concurrent function invocations using multiple vFPGAs on a single FPGA device under a multi-tenant scenario. We target SHA256 and configure the accelerator with 1, 2, and 4 vFPGAs on a single U50 FPGA. In this experiment, 30 clients concurrently submit a task to execute SHA256 on 8 MB of random input data.

Figure 9 shows the results. We observe a linear correlation between speedup and the number of vFPGAs, i.e., approximately 2 × and 4 × speedup with two and four vFPGAs, respectively, compared to one vFPGA. *In summary,* F3 effectively handles concurrent invocation requests on a single FPGA device with the help of F3's Shell.

Function chaining. We now evaluate the effectiveness of the accelerator chaining, F3's I/O abstraction mechanism for function chaining in FPGAs (§ 3.4). Our experiment executes a chain of two functions, GZIP and AES, where a file is first compressed by GZIP and then encrypted by AES. The file size is 150 KB before and 70 KB after compression. We compare our accelerator chaining with two baselines: Client-side piping and Server-side piping of OpenFaaS [78].

Figure 10 compares the function-chaining approaches. We observe that F3's accelerator chaining achieves 1.4 × and 1.7 × speedups against Client-side and Server-side pipings, respectively. This is because our approach eliminates the network round-trip for the intermediate result compared to the other mechanisms. *In summary,* F3's communication-less function chains lead to higher throughputs than the OpenFaaS baselines.

Entity	LUT (%)		Register (%)		BRAM (%)		URAM (%)	
U50 FPGA	872,000	(100)	1,743,000	(100)	1,344	(100)	640	(100)
Shell (static)	114,590	(13.1)	179,663	(10.3)	158	(11.8)	0	(0.0)
PCIe DMA	59,034	(6.8)	58,153	(3.3)	80	(6.0)	0	(0.0)
MMU	5,995	(0.7)	9,821	(0.6)	28	(2.1)	0	(0.0)
vFPGA	253,112	(29.0)	507,840	(29.1)	456	(33.9)	176	(27.5)
ADDMUL	2,207	(0.3)	3,849	(0.2)	0	(0.0)	0	(0.0)
AES	43,448	(5.0)	9,777	(0.6)	0	(0.0)	0	(0.0)
SHA3	4,081	(0.5)	5,624	(0.3)	0	(0.0)	0	(0.0)
GZIP	44,134	(5.1)	35,630	(2.0)	42	(3.1)	6	(0.9)
NW	138,661	(15.9)	80,702	(4.6)	79	(5.9)	0	(0.0)
HLS4ML	72,770	(8.3)	31,987	(1.8)	20	(1.5)	0	(0.0)
HLL	16,979	(1.9)	23,781	(1.4)	48	(3.6)	0	(0.0)
HCD	23,820	(2.7)	23,192	(1.3)	33	(2.5)	0	(0.0)
MD5	8,242	(9.5)	8,412	(0.5)	0	(0.0)	0	(0.0)
FFT	15,866	(1.8)	29,273	(1.7)	354	(26.3)	8	(1.3)

Table 5: FPGA resource usage for Shell components and accelerators of applications (as per Table 3).

FPGA resource usage. Table 5 reports the resource usage of Shell components and applications in our setup. We configure the vFPGA size to approximately 29% of the entire FPGA fabric, which is sufficient to place the largest accelerator (NW) in the region. Because the reconfiguration time is linearly proportional to the area of the vFPGA region [67], further resource optimization can contribute to minimizing the reconfiguration overhead. *In summary,* F3's Shell and vFPGA regions are properly configured to fit real-world applications.

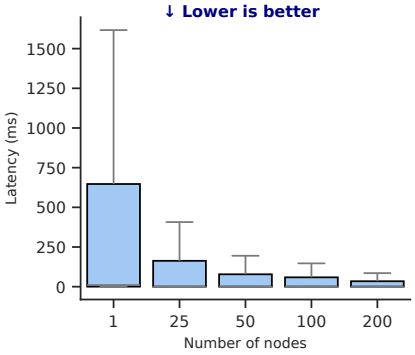


Figure 11: Cluster scalability.

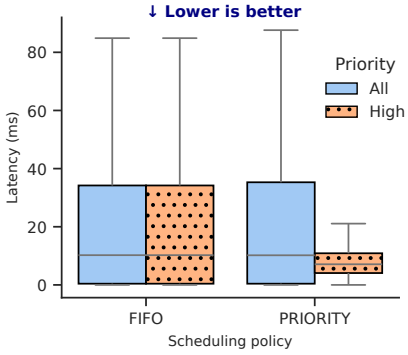


Figure 12: Scheduling analysis.

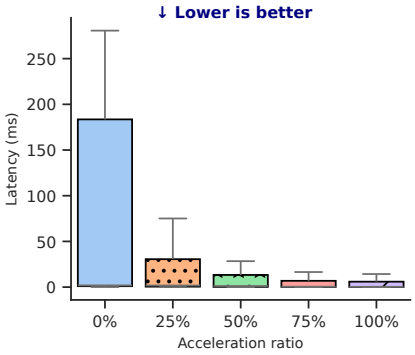


Figure 13: Selective acceleration.

5.3 Azure Production Traces

For large-scale design exploration, we evaluate the impact of FPGAs in a distributed production system running on Azure Functions [87].

5.3.1 *Experimental Methodology .*

Traces. We use production traces from Azure [138] that contain 1.9 million anonymized function invocations captured over two weeks. We further preprocess these traces to scale them up to realistic workloads for large-scale deployment, increasing the number of function invocations and their concurrency. We also enrich the traces by matching each function invocation to applications in Table 3, allowing us to work around the limitations an anonymized trace imposes. This approach is the closest to obtaining data on resource consumption and hardware acceleration from a real-world production serverless environment without having access to a more detailed trace or an actual production environment.

Preprocessing. We use kernel density estimation (KDE) to generate extended traces adhering to the original characteristics. This approach estimates invocation arrival time and duration distributions from the original trace and subsequently uses these distributions to create more extensive traces following the same distribution.

After extending the trace, we employ a two-stage mapping process to augment it and overcome the limitations of anonymization. In the first stage, we match functions from the anonymized trace to real-world counterparts in our testbed. This first stage involves a comprehensive analysis of all function identifiers and their invocation arrival and duration patterns, which we correlate to the most likely application types we have previously assessed. Despite the inherent ambiguities of this exercise, we strive for the closest approximation to reality. In the second stage, we augment the matched trace with parameters from our testbed to create a richer dataset. This enriched trace facilitates more nuanced simulations, enabling us to make informed assessments about potential performance improvements under varying system conditions.

Simulation. With system parameters for cluster size, FPGA configuration, function scheduling, and invocation serving in place, the simulator maintains the system state for each invocation, replaying requests as they appear in the traces. The simulator verifies if a deployed function is available for each invocation and picks the most suitable node otherwise. When the required bitstream is missing, we log a function cold start and reduce the trace duration by the expected acceleration inferred from the previous testbed evaluation (§ 5.2). We simulate the trace until no more invocations are left.

5.3.2 *Experimental Results .*

Next, we present the experimental results based on the simulation framework using the Azure production traces.

Scalability. We begin by evaluating how F3 scales with an increasing number of machines (scale-out). We increase the cluster size from 1 to 200 machines with one vFPGA per machine. We simulate the execution of the above trace using First Come, First Served (FCFS) scheduling and measure the latency for each function invocation.

Figure 11 shows latency boxplots for the scale-out experiment. The latency plots tighten as the cluster size increases because more FPGAs are available to process invocations, reducing queuing delays and cold starts. We observe improvements in the average latency of up to 3× when increasing the number of machines from 1 to 200. *In summary,* F3's performance effectively scales up to the number of machines in the cluster.

Scheduling strategies. Next, we evaluate the impact of the scheduling policy on latency. We execute the trace in our simulator using FCFS and priority-based scheduling for a cluster size of 100 with 4 vFPGAs per machine. Priority-based scheduling involves designating 25% of all invocations with a higher priority than other invocations, ensuring that the corresponding functions always have at least one vFPGA configured, i.e., removing cold starts but not queuing delays.

Figure 12 shows the result of this experiment. The priority-based policy achieves 0.69× lower invocation latency (7.1 ms) than the FCFS policy (10.2 ms) on average. Because prioritized invocations are more likely to reuse pre-configured accelerators, they avoid frequent FPGA reconfiguration and lead to low latencies. Priorities also significantly improve the 99th percentile of latency (P99), 0.01× lower than the FCFS policy, at the expense of worsening it for the other non-prioritized invocations (2.4× higher). Thus, priorities provide better latency and predictability. *In summary,* F3 supports different scheduling policies efficiently.

Selective acceleration. Finally, we consider the impact of accelerating only a subset of the functions in a workload while the rest run on CPUs. We consider five scenarios: 0% (no acceleration), 25%, 50%, 75%, and 100% (all functions accelerated). We select the subset of functions to accelerate uniformly at random. In all cases, we execute the trace with FCFS scheduling on a cluster size of 100 with 4 vFPGAs per machine. We measure the invocation latency of functions in each case.

Figure 13 shows the latency boxplots corresponding to each scenario. We observe that 25%, 50%, 75%, and 100% of the acceleration ratios achieve 2.0×, 3.5×, 4.6×, and 6.4× latency speedups relative to the 0% case, respectively. These results indicate that FPGA acceleration significantly benefits the workload by reducing the average latency substantially. *In summary,* F3 drastically improves the end-to-end latency even if the number of FPGA-accelerated functions is limited.

6 Related work

Serverless frameworks. Serverless computing has been the target of much research in the past few years [4, 8, 20, 22, 25, 41, 48, 53, 59, 60, 66, 76, 80, 84, 88, 95, 98, 101, 103, 106, 108, 116, 127, 129–132, 136–141]. Major cloud providers offer proprietary serverless function frameworks [10, 50, 87] that offer limited customizability to integrate hardware accelerators. Several open-source alternatives, such as OpenWhisk [45] and OpenLambda [55], provide more flexibility for on-premise deployments. We opt to integrate F3 with OpenFaaS [79] for simplicity, but our approach can also be implemented into other frameworks by wrapping function invocation inside lightweight VMs and unikernels.

Functions cold start. A significant challenge in serverless computing is the cold start problem. There is a wide variety of existing studies tackling the mitigation of cold start by function compression [20], data caching [101, 131], pre-warming [101], and checkpointing [66]. Due to our focus on FPGAs, F3 takes a different approach to reduce cold start delays by minimizing the necessary software stack to run a function and leveraging opportunistic task scheduling to reuse FPGA slots.

Serverless function virtualization. Lightweight VMs and hypervisors have been studied to improve isolation and performance in serverless environments. Firecracker [6] is a production-grade, lightweight hypervisor with a lean device model tailored for serverless workloads. LightVM [85] focuses on reducing boot time in Xen [2]. VMSH [111] provides a hypervisor-agnostic solution to attach services on the fly to a running lightweight VM. They achieve lightweightness to some extent by eliminating unnecessary functions from standard VMs and hypervisors without losing application compatibility.

On the other hand, our approach is based on unikernels, specialized single-image binaries that bundle application code with minimal OS services, significantly improving memory footprint, performance, and robustness. Despite manifold implementations in the last decade [1, 24, 63, 68, 83], there are not so many studies leveraging unikernels for serverless computing [43, 86]. Our focus on workload acceleration with FPGAs leads us to choose unikernels over other technologies because of their simplicity, modularity, and lightweightness.

Accelerators for serverless. With the surge in computation-heavy and machine-learning workloads, systems leveraging accelerators are in high demand in the cloud [33, 95, 113]. Systems like Molecule [40], Kernel-as-a-Service [93], BlastFunction [19], Mantle [100], and λ-NIC [29] strive to support various heterogeneous accelerators, where multiple OSes and a shim abstract the device heterogeneity from the serverless runtime. They align with our goal of achieving multi-tenancy for accelerators, while F3 offers further functionalities required to adopt FPGAs into today's serverless system stack, e.g., programmability and isolation.

FPGA multi-tenancy and scheduling. Multi-tenancy support for FPGA aims to maximize utilization while maintaining isolation between accelerators based on FPGA OSes and Shells, e.g., Coyote [67], AmorphOS [61], FSRF [71], Synergy [72], Optimus [81], ViTAL [134], and HeteroViTAL [135]. F3 could be built upon any of these Shells that support multiple isolated DPR regions (vFPGAs).

Efficient task scheduling for high FPGA resource utilization on a distributed cluster is an active area of research [28, 32, 38, 42]. These approaches offer priority-based task scheduling based on the comparative speedup of CPU versus FPGA, reallocating oversubscribed tasks to available CPUs. F3 diverges from these systems by leveraging and adapting Kubernetes and OpenFaaS to achieve similar functionality in a serverless computing ecosystem. Moreover, F3 provides a general mechanism to support various scheduling policies.

7 Conclusion

We have developed F3, the first comprehensive system that allows FPGAs to be seamlessly incorporated into serverless computing environments, enabling their efficient and accessible use. Overall, our paper makes the following contributions:

- **High-level FPGA API (§ 3.2):** We introduce a high-level, hardware-agnostic API that simplifies FPGA manipulation in serverless environments by abstracting low-level device management.
- **Unikernel-based FPGA virtualization (§ 3.3):** We propose a lightweight and secure isolation mechanism for serverless functions accelerated with FPGAs by means of a unikernel architecture.
- **FPGA Shell management (§ 3.4):** We run an FPGA resource manager that manipulates the Shell and improves invocation latencies through intelligent bitstream reuse and data transfer optimizations.
- **FPGA-aware orchestration (§ 3.5):** We design an FPGA-aware orchestrator that leverages accelerator locality to reduce reconfiguration overhead.

We build F3's end-to-end system stack based on an OpenFaaS framework and OCI/CRI standards. We comprehensively evaluate and demonstrate its utility using real-world applications, microbenchmarks, and Azure production traces.

Artifact availability

The F3 codebase is publicly available at https://github.com/TUM-DSE/F3.git.

Acknowledgements

This work was supported in parts by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY). The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

References

[1] [n. d.]. GitHub - rumpkernel/rumprun: The Rumprun unikernel and toolchain for various platforms — github.com. https://github.com/rumpkernel/rumprun/. [Accessed June 2, 2025].

[2] [n. d.]. Home - Xen Project — xenproject.org. https://xenproject.org/. [Accessed June 2, 2025].

[3] [n. d.]. Kata Containers. https://katacontainers.io/. Last accessed: June 2, 2025.

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

[5] Inc. Advanced Micro Devices. 2025. XRT Native APIs. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html. [Accessed June 2, 2025].

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[7] Ashwin M. Aji, Lokendra S. Panwar, Feng Ji, Milind Chabbi, Karthik Murthy, Pavan Balaji, Keith R. Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, Xiaosong Ma, and Rajeev Thakur. 2018. On the efficacy of GPU-integrated MPI for scientific applications. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing* (New York, New York, USA) *(HPDC '13)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/2462902.2462915

[8] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2023. Groundhog: Efficient request isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 398–415.

[9] Amazon. 2025. Amazon EC2 F2 Instances. https://aws.amazon.com/ec2/instance-types/f2. [Accessed June 2, 2025].

[10] Amazon. 2025. Serverless function, Faas serverless - Aws Lambda - AWS. https://aws.amazon.com/lambda/ [Accessed June 2, 2025].

[11] AMD. 2025. Fast Fourier Transform (FFT). https://www.xilinx.com/products/intellectual-property/fft.html. [Accessed June 2, 2025].

[12] AMD. 2025. Lossless Compression. https://www.xilinx.com/products/intellectual-property/ef-di-compress.html. [Accessed June 2, 2025].

[13] AMD. 2025. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries. [Accessed June 2, 2025].

[14] Assured. 2025. sha256. https://github.com/secworks/sha256. [Accessed June 2, 2025].

[15] The Kubernetes Authors. [n. d.]. Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ Section: docs.

[16] The Kubernetes Authors. 2024. Production-Grade Container Orchestration — kubernetes.io. https://kubernetes.io/. [Accessed June 2, 2025].

[17] The Knative Authors. 2025. Knative. https://knative.dev/docs/. [Accessed June 2, 2025].

[18] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[19] Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. 2020. BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 852–857. https://doi.org/10.23919/DATE48585.2020.9116333

[20] Rohan Basu Roy, Tirthak Patel, Rohan Garg, and Devesh Tiwari. 2024. CodeCrunch: Improving Serverless Performance via Function Compression and Cost-Aware Warmup Location Optimization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 85–101.

[21] Jayaram Bhasker. 1992. A VHDL primer. Prentice-Hall.

[22] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 381–397.

[23] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 34 (feb 2022), 42 pages. https://doi.org/10.1145/3506713

[24] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257. https://doi.org/10.1109/CloudCom.2015.89

[25] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[26] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, Vol. 2018. 2–8.

[27] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[28] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2597917.2597929

[29] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 67–77.

[30] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/2485922.2485945

[31] Google Cloud. 2025. GPU platforms. https://cloud.google.com/compute/docs/gpus. [Accessed June 2, 2025].

[32] Inc. Cloudera. 2025. Use FPGA scheduling. https://docs.cloudera.com/runtime/7.2.18/yarn-allocate-resources/topics/yarn-use-fpga-scheduling.html. [Accessed June 2, 2025].

[33] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. 2022. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 434–451.

[34] Kubernetes community. 2025. Container Runtime Interface (CRI). https://github.com/kubernetes/cri-api. Accessed June 2, 2025.

[35] Open Container Initiative community. 2025. OCI Image Format Specification. https://github.com/opencontainers/image-spec. [Accessed June 2, 2025].

[36] Open Container Initiative community. 2025. Open Container Initiative Runtime Specification. https://github.com/opencontainers/runtime-spec. Accessed June 2, 2025.

[37] NVIDIA Corporation. 2025. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. [Accessed June 2, 2025].

[38] Guohao Dai, Yi Shan, Fei Chen, Yu Wang, Kun Wang, and Huazhong Yang. 2014. Online scheduling for FPGA computation in the Cloud. In *2014 International Conference on Field-Programmable Technology (FPT)*. 330–333. https://doi.org/10.1109/FPT.2014.7082811

[39] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 356–370. https://doi.org/10.1145/3423211.3425690

[40] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 797–813. https://doi.org/10.1145/3503222.3507732

[41] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[42] Abid Farhan, Raafat Aburukba, Assim Sagahyroon, Mohammed Elnawawy, and Khaled El-Fakih. 2022. Virtualizing and Scheduling FPGA Resources in Cloud Computing Datacenters. *IEEE Access* 10 (2022), 96909–96929. https://doi.org/10.1109/ACCESS.2022.3204866

[43] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. USETL: Unikernels for Serverless Extract Transform and Load Why should you settle for less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3343737.3343750

[44] UTCS Systems for Concurrency and Evolving Architectures. 2025. FSRF codebase. https://github.com/utcs-scea/amorphos-fsrf. [Accessed June 2, 2025].

[45] The Apache Software Foundation. 2025. Apache OpenWhisk is a serverless, open source cloud platform — openwhisk.apache.org. https://openwhisk.apache.org/. [Accessed June 2, 2025].

[46] The Linux Foundation. 2025. containerd. https://containerd.io/. [Accessed June 2, 2025].

[47] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[48] Alexander Fuerst and Prateek Sharma. 2021. 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*. 386–400.

[49] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00021

[50] Google. 2025. Cloud Functions | Google Cloud — cloud.google.com. https://cloud.google.com/functions/. [Accessed June 2, 2025].

[51] Google. 2025. Cloud Tensor Processing Units (TPUs). https://cloud.google.com/tpu. [Accessed June 2, 2025].

[52] ETH Zurich Systems Group. 2025. Coyote codebase. https://github.com/fpgasystems/Coyote. [Accessed June 2, 2025].

[53] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[54] Robert Haase, Loic Royer, Peter Steinbach, Deborah Schmidt, Alexandr Dibrov, Uwe Schmidt, Martin Weigert, Nicola Maghelli, Pavel Tomancak, Florian Jug, and Eugene Myers. 2019. CLIJ: GPU-accelerated image processing for everyone. *Nature Methods* 17 (11 2019). https://doi.org/10.1038/s41592-019-0650-1

[55] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud'16)*. USENIX Association, USA, 33–39.

[56] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. 2022. A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. In *2022 14th International Conference on Computer Research and Development (ICCRD)*. 100–107. https://doi.org/10.1109/ICCRD54409.2022.9730377

[57] Docker Inc. 2025. alpine - Official Image. https://hub.docker.com/_/alpine. [Accessed June 2, 2025].

[58] Intel. 2025. Intel® Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html. [Accessed June 2, 2025].

[59] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 691–707.

[60] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[61] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 107–127. http://www.usenix.org/conference/osdi18/presentation/khawaja

[62] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

[63] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[64] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. *SIGPLAN Not.* 53, 4 (jun 2018), 296–311. https://doi.org/10.1145/3296979.3192379

[65] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127. https://doi.org/10.1109/ISCA.2016.20

[66] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 298–316.

[67] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[68] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[69] Fast Machine Learning Lab. 2025. Tutorial notebooks for hls4ml. https://github.com/fastmachinelearning/hls4ml-tutorial. [Accessed June 2, 2025].

[70] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. 2023. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3582016.3582048

[71] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J Rossbach. 2023. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 556–571.

[72] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. 2021. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 818–831.

[73] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge. 2005. Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing. *Proc. IEEE* 93, 2 (2005), 313–330. https://doi.org/10.1109/JPROC.2004.840303

[74] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 442–450.

[75] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* 54, 10s, Article 220 (sep 2022), 34 pages. https://doi.org/10.1145/3508360

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

[77] Nubificus Ltd. 2025. kata-urunc. https://github.com/nubificus/kata-containers/tree/feat_kata_urunc. [Accessed June 2, 2025].

[78] OpenFaaS Ltd. 2025. Chaining OpenFaaS functions. https://ericstoekl.github.io/faas/developer/chaining_functions/. [Accessed June 2, 2025].

[79] OpenFaaS Ltd. 2025. OpenFaaS. https://www.openfaas.com/. [Accessed June 2, 2025].

[80] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 132–147.

[81] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[82] Raju Machupalli, Masum Hossain, and Mrinal Mandal. 2022. Review of ASIC accelerators for deep neural network. *Microprocessors and Microsystems* 89 (2022), 104441. https://doi.org/10.1016/j.micpro.2022.104441

[83] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: the rise of the virtual library operating system. *Commun. ACM* 57, 1 (jan 2014), 61–69. https://doi.org/10.1145/2541883.2541895

[84] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[85] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[86] Ilias Mavridis and Helen D. Karatza. 2023. Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurr. Comput. Pract. Exp.* 35, 11 (2023). https://doi.org/10.1002/CPE.6365

[87] Microsoft. 2025. Azure Functions – Serverless Functions in Computing | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-

us/products/functions/. [Accessed June 2, 2025].
[88] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 228–244.
[89] Vivek Narasayya, Surajit Chaudhuri, et al. 2021. Cloud data services: Workloads, architectures and multi-tenancy. *Foundations and Trends® in Databases* 10, 1 (2021), 1–107.
[90] OpenCores. 2025. MD5 Pipelined. https://opencores.org/projects/md5_pipelined. [Accessed June 2, 2025].
[91] OpenCores. 2025. SHA3 (KECCAK). https://opencores.org/projects/sha3. [Accessed June 2, 2025].
[92] Oracle. 2025. Oracle Cloud Functions. https://www.oracle.com/cloud/cloud-native/functions/. [Accessed June 2, 2025].
[93] Tobias Pfandzelter, Aditya Dhakal, Eitan Frachtenberg, Sai Rahul Chalamalasetti, Darel Emmot, Ninad Hogade, Rolando Pablo Hong Enriquez, Gourav Rattihalli, David Bermbach, and Dejan Milojicic. 2023. Kernel-as-a-Service: A Serverless Programming Model for Heterogeneous Hardware Accelerators. In *Proceedings of the 24th International Middleware Conference* (<conf-loc>, <city>Bologna</city>, <country>Italy</country>, </conf-loc>) *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 192–206. https://doi.org/10.1145/3590140.3629115
[94] Khoa Dang Pham, Edson Horta, and Dirk Koch. 2017. BITMAN: A tool and API for FPGA bitstream manipulations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 894–897.
[95] Neha Prakriya, Yuze Chi, Suhail Basalama, Linghao Song, and Jason Cong. 2023. TAPA-CS: Enabling Scalable Accelerator Design on Distributed HBM-FPGAs. *arXiv preprint arXiv:2311.10189* (2023).
[96] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. https://doi.org/10.1109/MM.2015.42
[97] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. 2019. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. 1–8. https://doi.org/10.1109/ICESS.2019.8782524
[98] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 314–330.
[99] Anja Reuter, Timon Back, and Vasilios Andrikopoulos. 2020. Cost efficiency under mixed serverless and serverful deployments. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 242–245.
[100] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A case for function-as-a-service with disaggregated FPGAs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 333–344.
[101] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.
[102] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, and Antonio Corradi. 2021. A Shared Memory Approach for Function Chaining in Serverless Platforms. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. 1–6. https://doi.org/10.1109/ISCC53001.2021.9631385
[103] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 231–246.
[104] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 271–286. https://doi.org/10.1145/3297858.3304010
[105] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad
[106] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 663–677.
[107] Josef Spillner, Cristian Mateos, and David A Monge. 2018. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers 4*. Springer, 154–168.
[108] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. 2020. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.
[109] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69
[110] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. 2020. ASIC clouds: specializing the datacenter for planet-scale applications. *Commun. ACM* 63, 7 (jun 2020), 103–109. https://doi.org/10.1145/3399734
[111] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. 2022. VMSH: hypervisor-agnostic guest overlays for VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 678–696. https://doi.org/10.1145/3492321.3519589
[112] Donald Thomas and Philip Moorby. 1991. The Verilog Hardware Description Language. Springer Science & Business Media.
[113] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A framework for massively parallel streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 639–651.
[114] TUM-DSE. 2025. F3 codebase. https://github.com/TUM-DSE/F3. [Accessed June 2, 2025].
[115] Furkan Turan and Ingrid Verbauwhede. 2020. Trust in FPGA-accelerated Cloud Computing. *ACM Comput. Surv.* 53, 6, Article 128 (dec 2020), 28 pages. https://doi.org/10.1145/3419100
[116] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.
[117] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. 2020. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. 1–7. https://doi.org/10.1109/COINS49042.2020.9191377
[118] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Comput. Surv.* 51, 4, Article 72 (jul 2018), 39 pages. https://doi.org/10.1145/3193827
[119] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.
[120] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. 2020. When FPGA meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing* 10, 2 (2020), 1344–1357.
[121] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams
[122] AMD Xilinx. 2025. Alveo U50 Data Center Accelerator Card. https://www.amd.com/en/products/accelerators/alveo/u50/a-u50-p00g-pq-g.html. [Accessed June 2, 2025].
[123] AMD Xilinx. 2025. DMA/Bridge Subsystem for PCI Express. https://docs.amd.com/r/en-US/pg195-pcie-dma. [Accessed June 2, 2025].
[124] AMD Xilinx. 2025. VCK5000 Versal Development Card. https://www.xilinx.com/products/boards-and-kits/vck5000.html. [Accessed June 2, 2025].
[125] AMD Xilinx. 2025. Vivado ML Edition. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html. [Accessed June 2, 2025].
[126] Liming Xiu. 2019. Time Moore: Exploiting Moore's Law From The Perspective of Time. *IEEE Solid-State Circuits Magazine* 11, 1 (2019), 39–55. https://doi.org/10.1109/MSSC.2018.2882285
[127] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, et al. 2024. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 331–348.
[128] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. 2020. Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218676
[129] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2023. Flame: A Centralized Cache Controller for Serverless Computing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 153–168.

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

[131] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 335–350.

[132] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1489–1504.

[133] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 30–44.

[134] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 845–858. https://doi.org/10.1145/3373376.3378491

[135] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A virtualization stack for heterogeneous FPGA clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 470–483.

[136] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1187–1204.

[137] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus:{NIMBLE} task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 653–669.

[138] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580

[139] Zili Zhang, Chao Jin, and Xin Jin. 2024. Jolteon: Unleashing the Promise of Serverless for Serverless Workflows. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 167–183.

[140] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. BeeHive: Sub-second elasticity for web services with Semi-FaaS execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 74–87.

[141] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.