Proteus: Heterogeneous FPGA Virtualization

Felix Gust
Technical University of Munich
Munich, Germany
felix.gust@tum.de

Shu Anzai*
University of California
Los Angeles, USA
shuanzai@ucla.edu

Charalampos Mainas
Technical University of Munich
Munich, Germany
charalampos.mainas@tum.de

Atsushi Koshiba†
Tokyo University of Science
Tokyo, Japan
atsushi.koshiba@tum.de

Pramod Bhatotia
Technical University of Munich
Munich, Germany
pramod.bhatotia@tum.de

Abstract

Cloud providers have widely adopted FPGAs to meet the high-performance, energy-efficient demands of cloud workloads. While they offer homogeneous FPGAs per service, recent FPGA products exhibit increasing heterogeneity in terms of vendors, capacity, off-chip memory, and performance. These diverse properties not only render applications incompatible across different FPGAs but also lead to performance disparities and resource inefficiency.

To fill these gaps, we propose Proteus, a heterogeneous FPGA virtualization framework. Proteus allows applications to manage diverse FPGAs transparently by abstracting their properties with four key contributions: *an FPGA virtualization stack* to abstract different FPGA stacks from applications, *a platform-agnostic API* to manage FPGAs regardless of their vendors/architectures, *memory virtualization* to optimize memory allocation depending on the off-chip memory type, and *a performance-aware scheduler* to deploy applications on the best-suited FPGAs based on their predicted performance.

We implement Proteus for cross-vendor FPGAs: AMD (U50, U280) and Intel (Stratix 10). Our evaluation highlights that Proteus makes applications deployable on any FPGA with 4.9-6.8% overheads for AMD FPGAs, and the performance-aware scheduler yields 13.8% throughput gain on average.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Cloud computing**.

Keywords: FPGA, operating systems, virtualization, orchestration, shell, schedulers, memory virtualization, heterogeneous systems.

*Work done while the author was an intern at TU Munich.
†Work done while the author was at TU Munich.

This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803577

ACM Reference Format:
Felix Gust, Shu Anzai, Charalampos Mainas, Atsushi Koshiba, and Pramod Bhatotia. 2026. Proteus: Heterogeneous FPGA Virtualization. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3767295.3803577

1 Introduction

Field Programmable Gate Arrays (FPGAs) are widely adopted in data centers [20, 96] and commercial cloud instances [5, 28, 56, 89, 101, 110] to meet rising high-performance demands on modern cloud workloads [19, 33, 36, 43, 103, 109, 117]. While a wide variety of accelerators have evolved in the last decade, such as GPUs [30, 32], TPUs [45], ASICs [29, 46], FPGAs bring unique advantages thanks to their reconfigurability. FPGAs enable users to design and deploy custom logic tailored to their target computations, delivering not only high performance and energy efficiency but also flexibility and customizability for ever-changing cloud workloads.

In contrast, the increasing heterogeneity of cloud FPGAs poses a significant hurdle for users seeking to quickly develop and deploy FPGA-accelerated workloads. Table 1 summarizes the FPGA products offered by major cloud providers and vendors, clearly demonstrating this heterogeneity across several dimensions, including vendors, logic capacity, off-chip memory, and backend runtime. These diverse characteristics not only complicate application porting across different FPGAs but also lead to unforeseen performance discrepancies due to varying hardware specifications.

Despite the increasing diversity of cloud FPGA resources, each cloud vendor relies on *homogeneous* FPGA resources due to FPGA system incompatibility and maintenance difficulties. Therefore, users are forced to use a single FPGA type per cloud service, regardless of their application requirements. This poor flexibility leads to high development efforts, performance mismatches, and low resource utilization. These facts motivate us to answer the following question: *Can we design a new cloud FPGA framework that virtualizes and orchestrates heterogeneous FPGAs to improve FPGA utilization, flexibility, and performance?*

Product	Vendor	Board	Logic	Memory	Runtime
Cloud FPGA instances					
AWS F2[5]	AMD	Custom	2,852k	64GiB DDR 16GiB HBM	Vitis/XRT
Alibaba F3[28]	AMD	Custom	2,586k	64GiB DDR	Vitis/XRT
AWS VT1[101]	AMD	U30	460k	8GiB DDR	Video SDK
MS Azure[89]	AMD	U250	3,780k	64GiB DDR	Vitis/XRT
VMAccel[110]	Intel	IA-840F	2,692k	32GiB DDR	oneAPI
Other commercial FPGA products					
Entry[121]	AMD	U50	1,907k	8GiB HBM	Vitis/XRT
Mid-range[122]	AMD	U55C	2,852k	16GiB HBM	Vitis/XRT
High-end[120]	AMD	V80	5,631k	32GiB HBM	Vitis/XRT

Table 1. The heterogeneity of commercial FPGA products offered by cloud instances and FPGA vendors.

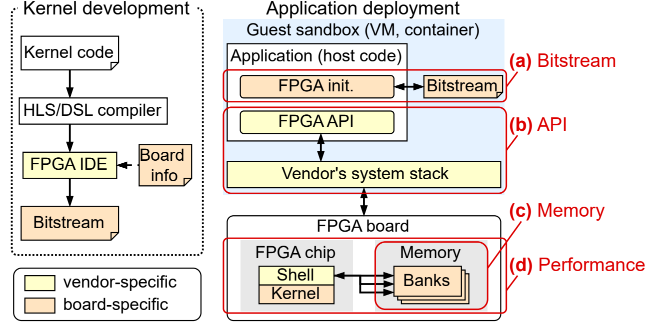


Figure 1. The cloud FPGA architecture. (a)-(d) are properties of heterogeneous FPGAs we address in this paper.

While there are existing studies targeting heterogeneous FPGA support for application and hardware development, such as kernel code compatibility [74, 81] and hardware functionalities [82], there is *no* end-to-end solution that addresses heterogeneous cloud FPGA management for application deployment. More specifically, we highlight the following four challenges that our work addresses. ***First, guest application sandboxes for heterogeneous FPGAs are missing.*** Existing FPGA virtualization systems [84, 130, 136] assume that FPGA-specific software stacks (runtimes, drivers) are installed in guest boxes, which forces applications running there to explicitly use a specific type of FPGA. ***Second, vendor-provided runtimes support different APIs and execution models.*** Vendor-provided FPGA system stacks and runtimes [93, 132] offer various APIs for FPGA management [2, 38, 49], even though they offer equivalent functionalities. This fact makes guest applications *incompatible* with different FPGAs of different vendors. ***Third, off-chip memory types affect application portability and performance.*** As shown in Table 1, off-chip memory types expose not only memory capacity gaps, which affect the execution capability of applications, but also different memory types, i.e., DDR and High Bandwidth Memory (HBM). Their achievable throughputs heavily depend on the applications' data placement and access patterns. ***Fourth, the performance gaps across FPGAs complicate task scheduling.*** The performance of the same custom FPGA logic varies depending on the target FPGA properties, i.e., their varying maximum clock speeds (Fmax) [50, 75, 141] and off-chip memory specifications [25, 55, 102, 111]. Task scheduling that ignores these performance gaps can lead to unexpected slowdowns.

To address these challenges, we propose Proteus, a heterogeneous FPGA virtualization mechanism in cloud environments. Proteus comprehensively addresses the aforementioned challenges by achieving four contributions:

(1) **Heterogeneous FPGA virtualization stack** (§3.2). Proteus OS and hypervisor decouple guest applications from heterogeneous FPGA system stacks and allow them to be deployed with any underlying FPGAs.
(2) **FPGA-agnostic API** (§3.3). Proteus API is a unified API that enables guest applications to transparently manage any FPGAs, ensuring cross-FPGA compatibility.
(3) **Heterogeneous memory virtualization** (§3.4). Proteus's memory management mechanism abstracts the complex off-chip memory architectures and transparently optimizes buffer allocation and data transfers.
(4) **Performance-aware scheduler** (§3.5). Proteus's task scheduler predicts the application performance on different FPGAs and chooses the best-suited FPGA based on the predicted performance and run-time FPGA usage.

We implement our prototype upon two vendor-specific FPGA stacks, AMD's Vitis/XRT [132] and Intel's FPGA SDK for OpenCL [39], supporting four FPGA setups: AMD's HBM FPGAs (Alveo U50 and U280), AMD's DDR FPGA (U280), and Intel's DDR FPGA (Stratix 10). Our evaluation shows that Proteus can deploy applications across the three AMD FPGA setups, with average overheads of 6.8%, 6.1%, and 4.9% per setup. In addition, Proteus's memory management mechanism achieves up to 1.95 $\times$ speedups by applying data placement and access optimizations without code changes. Lastly, Proteus's performance-aware scheduler improves application throughput by 13.8% on average compared to the worst-case scenario, only 0.3% below the best-case scenario (14.1%).

2 Background and Motivation

2.1 Cloud FPGA Architecture

We first introduce the architecture of commodity FPGA-equipped cloud environments, as illustrated in Figure 1.

FPGA application. FPGA applications consist of *kernel code* and *host code*. The kernel code, which represents computation logic (*kernel*) on FPGAs, is written in Hardware Description Languages (HDL) [15, 107]. High-Level Synthesis (HLS) [11, 49, 67, 78, 100] and Domain Specific Language (DSL) compilers [12, 26, 68, 79] convert software code (e.g.,

C/C++, Python) into HDL. Vendor-specific FPGA IDEs, such as AMD Vivado [129] and Intel Quartus [40], then compile the HDL code and generate a *bitstream*, hardware configuration data used to reconfigure the FPGA. The host code is a CPU application that manages the FPGA device through control APIs such as OpenCL [49]. These APIs communicate with vendor-specific system stacks (runtime and OS drivers) to invoke reconfiguration, kernel execution, and data transfers.

FPGA device. In commodity cloud environments, FPGAs serve as PCIe-connected devices [5, 28, 89, 110] directly connected to CPU servers. These devices are equipped with the FPGA chip and other peripherals, including off-chip memory modules and network ports. To establish communications between a host server, FPGA kernels, and other peripherals, a portion of the FPGA fabric is typically configured as a *Shell* [37, 69, 132], a static region containing glue logic, such as a PCIe DMA and memory controllers. The remaining partitions serve as *reconfigurable slots* for deploying kernels.

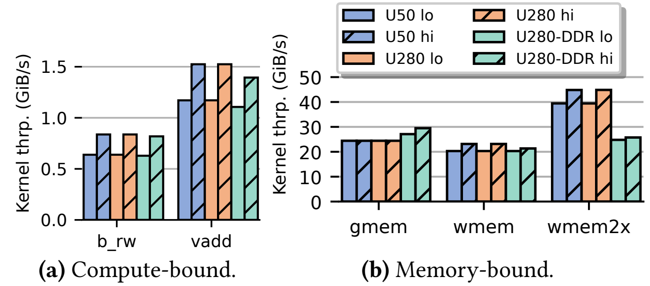
2.2 FPGA Heterogeneity Properties in the Cloud

The heterogeneous properties of a typical cloud deployment, as illustrated in Figure 1, pose several key challenges for adopting various FPGA devices, as described below.

(a) Bitstream incompatibility. First, bitstreams are not compatible with different FPGAs. Although some HLS/DSL compilers address cross-device or cross-vendor compatibility for kernel code [80], the bitstreams generated by FPGA IDEs are *fundamentally incompatible*. Unlike CPU binaries, whose instructions can be translated across heterogeneous CPUs via cross-ISA translation [47, 98], bitstream configurations, such as wiring, physical placement of logic, and external I/O connectivity, are strictly tied to each target FPGA. Moreover, existing FPGA APIs and runtimes [2, 38, 49, 132] directly expose FPGA devices to host code, which are typically identified by the Shells installed on each board [7, 8, 59]. The bitstream incompatibility and lack of FPGA device virtualization force host code to explicitly manage corresponding bitstreams, losing application portability and compatibility.

(b) API heterogeneity. Second, FPGA control APIs are specific to vendor-provided FPGA runtimes. While these runtimes guarantee API compatibility for the same vendor's FPGA devices [93, 132], they expose different APIs for host code [2, 38, 39]. Although OpenCL [39] and oneAPI [38] are designed as unified APIs for different accelerators, FPGA vendors even extend the original API specifications to offer vendor-specific features for device-specific memory configurations [61, 62, 131]. This API heterogeneity makes the host code incompatible across vendor-specific FPGA platforms.

(c) Memory heterogeneity. Third, cloud FPGA products are equipped with different off-chip memories as shown in Table 1. In particular, there are two major memory types: traditional DDR, which offers high capacity and per-bank



(a) Compute-bound. **(b)** Memory-bound.

Figure 2. Kernel throughputs on heterogeneous FPGA setups (Table 4) for selected applications from Table 5. lo and hi mean lower/higher clock speeds (300/400 MHz).

throughput, and emerging HBM, which offers more memory banks but smaller capacity per bank. Existing FPGA systems expose this memory heterogeneity directly to applications as either static hardware configuration [69, 128] or low-level APIs that require users to explicitly describe data placement [131]. The memory heterogeneity significantly increases users' engineering efforts to achieve their desired performance, and even makes their kernels unexecutable due to memory capacity limits.

(d) Performance heterogeneity. Fourth, heterogeneous FPGAs create significant performance gaps due to two key factors: the kernel's maximum clock frequency (*Fmax*) and off-chip memory types. Fmax is a well-known performance metric limited by the worst negative slack (WNS) from IDE's synthesis/placement results for the kernel's hardware design. We observed Fmax variances of up to 168 MHz across different FPGA products (§5.1). The off-chip memory types are also crucial for memory-bound kernels. Their throughputs can be significantly affected by the memory type and allocation strategies, depending on memory access patterns [118].

Figure 2 shows how these factors affect the actual throughputs of (a) compute-bound and (b) memory-bound kernels on various FPGAs; the clock speed is a key factor for compute-bound kernels, while the off-chip memory type affects memory-bound kernels more significantly. The unawareness of these performance gaps is problematic for orchestrators [58] responsible for task scheduling and deployment in the modern cloud, leading to unexpected performance drops.

Our goal. To overcome these gaps, we propose Proteus, a new mechanism for *heterogeneous FPGA virtualization* that abstracts heterogeneous properties and provides diverse cloud FPGAs to users as unified, *virtual FPGAs*. Unlike prior FPGA virtualization studies that focus on time- and space-sharing of a single FPGA [4, 17, 18, 65, 69, 76, 84] or only address the API compatibility [94, 136], Proteus offers an end-to-end solution that allows multi-tenant applications to efficiently and transparently share a pool of heterogeneous FPGAs across distributed nodes, while ensuring the API and bitstream compatibility, isolation, and performance.

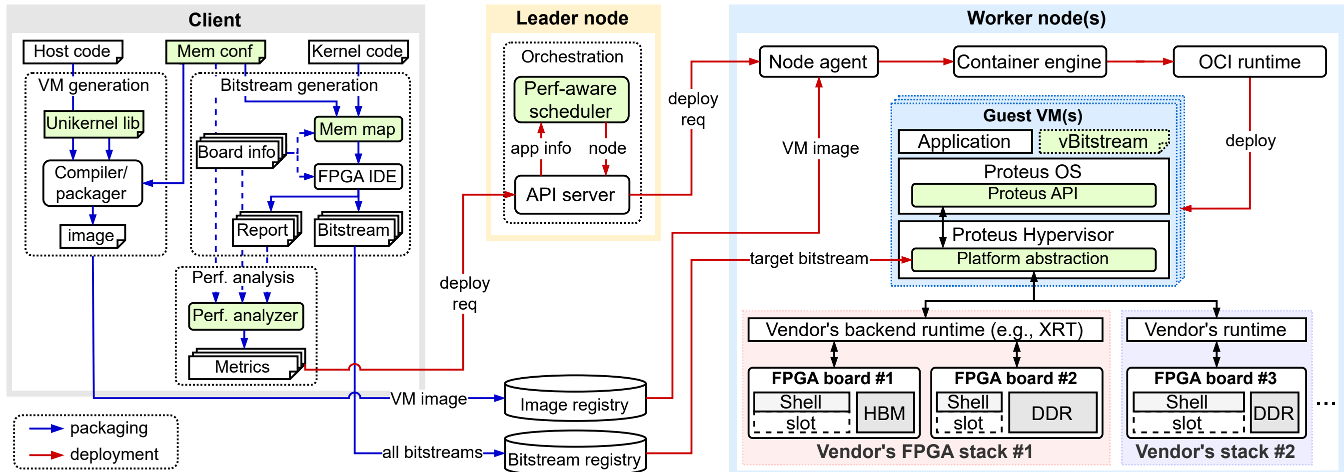


Figure 3. The system overview of Proteus. The key features are highlighted as green boxes.

2.3 Design Challenges and Key Ideas

We highlight the design challenges of Proteus to abstract these heterogeneous properties along with our corresponding solutions.

#1: Guest sandbox virtualizing heterogeneous FPGAs. To address (a), we design a new virtualization stack in a multi-tenant cloud, which provides guest applications with a unified view for heterogeneous FPGAs and bitstreams. Existing vendor solutions [130] and academic studies [84, 136] provide guest sandboxes (containers, VMs) for homogeneous FPGAs that install software stacks tailored to those FPGAs. However, they make guest applications incompatible with heterogeneous FPGAs and prevent orchestrators from flexible task deployment on different FPGAs, lowering FPGA utilization.
Key idea. We propose an FPGA-agnostic sandbox and hypervisor, which decouples applications from heterogeneous FPGA stacks and enables flexible task deployment (§3.2).

#2: Platform-agnostic FPGA API. To address (b), we design a unified API that enables applications to manage heterogeneous FPGAs regardless of vendor or platform. Existing APIs offered by vendor-provided FPGA platforms [2, 38, 49] lack functionalities to abstract off-chip memory architectures for buffer allocation, which necessitates vendor-specific extensions for explicit data placement on the target FPGA memory [61, 62, 131]. Such low-level management for heterogeneous FPGAs must not be exposed to guest applications to guarantee application compatibility.
Key idea. We propose a platform-agnostic API that abstracts FPGA heterogeneity, including off-chip memory types, and transparently manages various FPGAs (§3.3).

#3: Data placement for heterogeneous FPGA memory. To address (c), we design an FPGA memory virtualization mechanism that abstracts off-chip memory architectures and liberates users from complex data allocation optimizations for individual FPGA boards. Existing studies leverage static HLS code analysis for data placement optimizations [118]. However, these approaches are not only specific to their target languages but also cannot handle dynamic memory requests from applications, such as memory oversubscription.
Key idea. We propose a heterogeneous memory virtualization mechanism that transparently performs buffer placement optimization and oversubscription based on application demands and off-chip memory on the target FPGA (§3.4).

#4: Task scheduling for heterogeneous FPGAs. To address (d), we design a task-scheduling algorithm for heterogeneous FPGAs that enables cloud orchestrators to make task-deployment decisions based on kernel performance differences across FPGAs. A key challenge is analyzing kernel performance across different FPGAs. Existing studies [83] propose a static code analysis for performance prediction of OpenCL kernels, while they do not consider the end-to-end execution flows, including PCIe data transfers and their data access patterns between host and FPGA memory.
Key idea. We propose a performance-aware task scheduler that estimates the end-to-end execution latency of kernels and detects performance gaps on FPGAs (§3.5).

3 Design

3.1 System Overview

Figure 3 illustrates the Proteus system overview, where an orchestrator on a *leader node* deploys applications into *worker nodes* equipped with one or more *FPGA boards*. There are two steps to execute applications: packaging and deployment.

Packaging. For application packaging, users prepare a VM image that runs the host code, FPGA-specific bitstreams generated from the kernel code, and a memory configuration

memconf (§ 3.3) common for all FPGAs. For host code development, Proteus provides a unikernel library, i.e., *Proteus OS*, that abstracts underlying FPGAs and provides a platform-agnostic *Proteus API* for guest-FPGA communication. For kernel code development, users can use commodity FPGA IDEs [40, 129], while Proteus extends this process by offering a *memory mapper* and *performance analyzer*. The memory mapper ensures that each kernel has access to all memory banks on each FPGA board, and the performance analyzer estimates the kernel's performance for each FPGA. Users upload these images to the *image registry* and *bitstream registry*.

Deployment. For application deployment, users submit a deployment request to the orchestrator, where the *performance-aware scheduler* selects the most appropriate worker node based on the application's estimated performance. The deployment request is then forwarded to the worker-node components (node agent, container engine, OCI runtime), which fetch the VM image and deploy it on the selected node. The applications are individually encapsulated by the Proteus OS to ensure isolation among guest applications. Proteus API calls from guest applications are trapped by the *Proteus hypervisor*, which communicates with vendor-specific FPGA stacks via the *platform abstraction layer*. For FPGA initialization, i.e., loading a bitstream, Proteus OS exposes a *virtual bitstream (vBitstream)*, an FPGA-agnostic configuration file, to guest applications. On behalf of applications, the hypervisor manages an actual bitstream and initializes the FPGA.

To support a new FPGA, Proteus only needs two extensions: a platform abstraction layer and a memory mapper, both provided by vendors or cloud providers. Users only provide device-agnostic host/kernel code and memconf.

3.2 Proteus OS and Hypervisor

We design Proteus OS and hypervisor, a heterogeneous FPGA virtualization stack for a multi-tenant cloud. As we target cloud-native environments, where lightweight sandboxes [3, 87, 99, 106, 135] are commonly used, we adopt a unikernel [72, 73, 85, 90] as a guest sandbox. Unikernels' hypervisor-level isolation is well-suited for FPGA virtualization in a multi-tenant cloud, as they virtualize I/O devices, unlike containers [105, 135], which share the host OS [31, 34].

Proteus OS. Figure 4 illustrates Proteus OS and hypervisor architectures. Proteus OS encapsulates each application, offering two key FPGA abstractions: *virtual bitstream* (vBitstream) and *Proteus API*. The vBitstream is an abstraction of bitstream files, which contains metadata to identify actual bitstreams in the registry, and *memconf* (detailed in § 3.3). It is forwarded to the Proteus hypervisor for FPGA initialization.

The Proteus API is a unified, FPGA-agnostic API that allows a guest application to transparently obtain, manipulate,

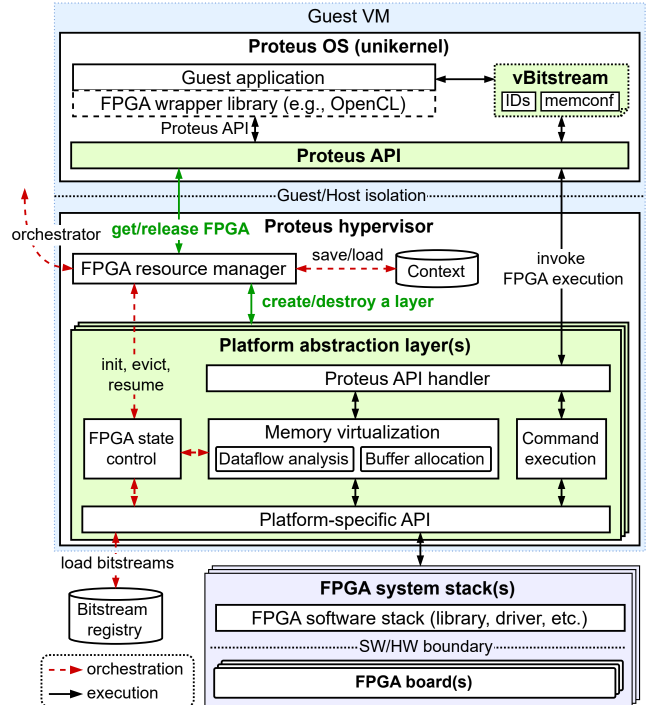


Figure 4. Proteus OS and hypervisor architecture.

and release FPGAs regardless of their types, detailed in §3.3. Proteus can offer a wrapper library for compatibility with other APIs, e.g., OpenCL. The wrapper library transparently converts these API calls into the corresponding Proteus API.

Proteus hypervisor. Proteus hypervisor is spawned per application and manages FPGAs upon Proteus API calls, offering two key components: the *FPGA resource manager* and *platform abstraction layer*. The resource manager initializes and allocates FPGA resources. When an application requests an FPGA, the resource manager creates a corresponding platform abstraction layer and triggers FPGA initialization, which involves searching the bitstream registry.

The platform abstraction layer is dedicated to its corresponding FPGA system stack and performs platform-specific FPGA operations. It converts Proteus APIs into platform-specific APIs such as OpenCL [39], XRT API [2], and oneAPI [38]. It also performs state operations for cloud orchestration services, such as task preemption and migration.

FPGA sharing and isolation. The Proteus hypervisor supports FPGA time-sharing with isolation between applications. Before switching applications, it completes all FPGA operations and flushes off-chip memory buffers to prevent data leakage. The FPGA's internal state (e.g., BRAM) is reset when a new bitstream is loaded. Additionally, Proteus ensures bitstream authenticity by using a dedicated registry to sign and verify uploaded bitstreams, similar to Docker Hub [57].

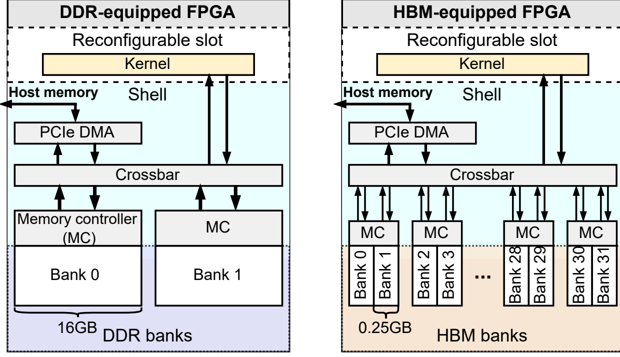


Figure 5. Modern FPGA device architectures [119, 121].

```
1 void vadd(int *in1, int *in2, int *out, size_t size) {
2   for(int i; i<size; i++)
3     out[i] = in1[i] + in2[i];
4 }
```

```
1 [connectivity]
2 nk=vadd:1:vadd_1
3 sp=vadd_1.in1:HBM[0]
4 sp=vadd_1.in2:HBM[1]
5 sp=vadd_1.out:HBM[2:3]
```

Listing 1. Example kernel code and its memory configuration in Vitis [128].

3.3 Proteus Programming Model and API

We introduce Proteus API and programming model, designed to simplify FPGA development by abstracting device-specific bitstream management and complex off-chip memory architectures. While Proteus API inherits the existing programming model [38, 49], it offers a new execution model that abstracts off-chip memory differences on FPGA devices.

FPGA hardware architecture. We first introduce the hardware architecture of modern cloud FPGA devices [5, 110, 119, 121], illustrated in Figure 5. It consists of three components: reconfigurable slots, the Shell, and off-chip memory banks. While the Shell architecture depends on FPGA platforms [37, 65, 69, 132], they have common components: a PCIe DMA IP for host-FPGA data transfers, memory controllers for off-chip memory reads/writes, and a crossbar for interconnects. Off-chip memory is the primary memory for kernel execution. There are two popular memory types: DDR and HBM. HBM is a 3D-stacked memory that achieves higher throughputs than conventional DDR memories thanks to multiple banks, e.g., HBM2 on Alveo U50 and U280 has 32 banks with 460 GB/s theoretical bandwidth, whereas U280's DDR4 has two banks with 38.5 GB/s bandwidth [119, 121]. However, HBM capacity and per-bank throughput are typically limited; for example, U280's HBM bank has 256 MiB and achieves 14.375 GB/s, whereas its DDR4 bank has 16 GiB and

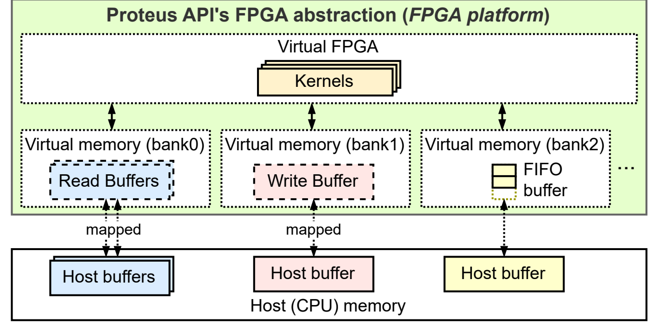


Figure 6. Proteus API's programming model.

```
1 # vmem[{bank}],{access_type},{min_datasize}
2 sp=vadd_1.in1:vmem[0], mem:in,  4096
3 sp=vadd_1.in2:vmem[1], strm:in, 4096
4 sp=vadd_1.out:vmem[2], mem:out, 4096
```

Listing 2. Memory configuration (memconf) in Proteus.

achieves 19.25 GB/s. As demonstrated in § 2.3, off-chip memory types can significantly affect the kernel's throughput, necessitating proper data placement and access pattern optimizations for each FPGA to achieve optimal performance.

Unfortunately, existing FPGA APIs [2, 38, 49] do not abstract off-chip memory types and force developers to explicitly define memory bank connections for their kernels [62, 128]. Listing 1 shows an example of kernel code (vector add) and its memory configuration in Vitis [128], where two input ports are tied to HBM banks 0 and 1, and one output port to two continuous banks 2 and 3. Such a hard-coded configuration is unrealistic for heterogeneous FPGAs equipped with different off-chip memories. Furthermore, developers are also forced to change their host code to explicitly allocate buffers to appropriate banks [131, 133]. Proteus API mitigates these difficulties by abstracting the complex memory architecture and data management from developer views.

Programming model. Figure 6 illustrates Proteus API's execution and memory model. Similar to OpenCL [49], Proteus abstracts on-FPGA computation logic as *kernels* and data as *buffers*. To manage kernels and buffers uniformly across different FPGAs, the API exposes *FPGA platforms*, which are an abstracted view of FPGA devices. The platform consists of *virtual FPGAs* where kernels are created, and *virtual memory (vmem) banks* where buffers are created. The vmem banks have no capacity or number limits; the platform abstraction layer dynamically manages and optimizes buffer allocation and data transfers to actual memory banks based on the kernel's data access pattern (detailed in §3.4). Proteus supports two types of buffers: *memory* and *stream*. The memory buffers function as normal memory blocks, whereas the stream buffers act as FIFOs, enabling asynchronous reads/writes that can overlap with kernel execution.

Proteus API	Description
Create buffers and kernels	
proteus::buffer(*size, host_ptr*)	create a buffer (return *buffer*).
proteus::kernel(*name, args*)	create a kernel (return *kernel*).
Initialize an FPGA platform	
proteus::platform(*virt bs*)	get FPGA (return *platform*).
platform::add(*kernels* or *buffers*)	add objects to the platform.
platform::rm(*kernels* or *buffers*)	delete objects in the platform.
Kernel execution and synchronization	
platform::execute(*kernels*)	invoke kernel execution.
platform::sync(*kernels*)	wait for completion of *kernels*.

Table 2. Proteus primitive APIs.

```
1  void vadd_host() {
2    /* Create kernels and buffers */
3    in1  = proteus::buffer(size, in1_host);
4    in2  = proteus::buffer(size, in2_host);
5    out  = proteus::buffer(size, out_host);
6    vadd = proteus::kernel("vadd", {in1, in2, out, size});
7
8    /* Allocate buffers and kernels to the FPGA platform */
9    platform = proteus::platform(virt_bs);
10   platform.add(vadd, in1, in2, out);
11
12   /* Execute the kernel */
13   platform.execute(vadd);
14   platform.sync(vadd);
15 }
```

Listing 3. Example host code written for Proteus API.

To enable this programming model, Proteus asks users to write a memory configuration file *(memconf)*, shown in Listing 2. Its syntax is similar to Vitis (Listing 1), but there are two key differences. First, the kernel's input and output ports are connected to user-defined vmem banks, and the physical memory topology is abstracted. Second, for each bank connection, users define two more arguments: *memory access type* and *minimal data size*. The former defines buffer types (mem/strm) and I/O directions (in/out), which are used for actual buffer allocations and data transfers during execution. The latter defines the smallest unit of data sizes consumed/produced by the kernel, which is used for memory oversubscription (§3.4) and performance analysis for scheduling (§3.5).

Proteus API. Table 2 shows primitive Proteus APIs, and Listing 3 shows the example host code for the vadd kernel (Listing 1). First, the host code instantiates buffers and kernels (L3-6). Proteus allows the host code to bind FPGA buffers to larger host-side buffers. In such a case, at the kernel execution, Proteus automatically splits the host-side buffer into multiple chunks that fit the buffer size, and repeats the execution and data transfers until all data is processed. Next, the host code obtains an available FPGA platform (L9) and adds the created kernels and buffers to it (L10). Upon the request, the Proteus hypervisor creates a platform abstraction layer for the selected FPGA, and the newly created layer accordingly initializes the target FPGA, i.e., reconfigures the FPGA slot with the dedicated bitstream, and creates buffers on off-chip memory banks. Lastly, the host code invokes the kernel execution (L13-14). Unlike other FPGA APIs [2, 38, 49], the Proteus API is agnostic to data accesses [13]; the host code does not have to explicitly describe host-FPGA data transfers because the platform abstraction layer transparently optimizes and performs data transfers based on the kernel's data flow and memory topologies (detailed in §3.4).

3.4 Heterogeneous Memory Virtualization

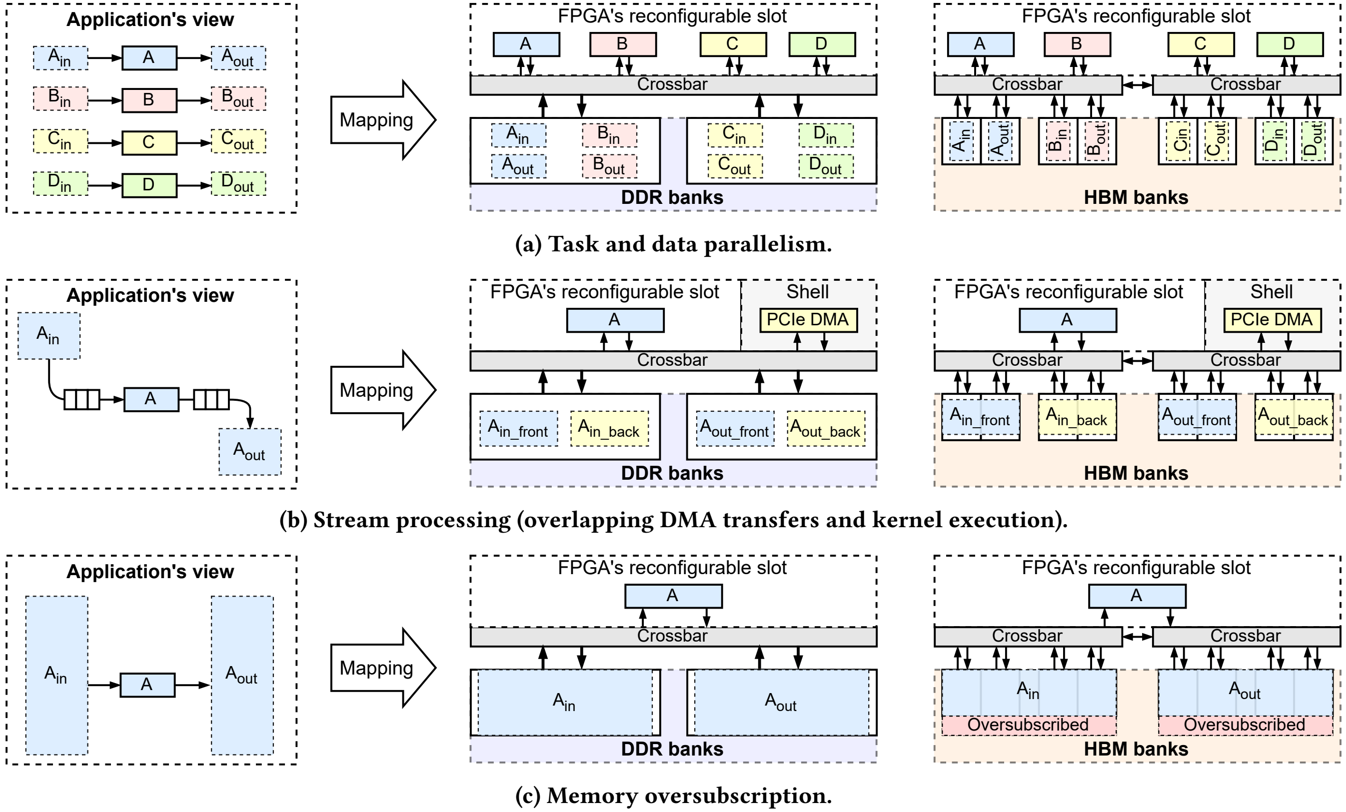
We introduce Proteus's memory virtualization mechanism for heterogeneous off-chip memory architectures. Proteus transparently maps buffers to physical memory banks and manages data transfers based on the kernel's data access patterns, described by memconf, and dynamic memory requests from applications. Like other FPGA execution models [38, 49, 69], Proteus adopts static memory allocation, requiring all memory buffers to be allocated before kernel execution.

Buffer allocation strategy. Proteus allocates buffers to maximize the kernel's throughput by distributing them across different memory banks to prevent I/O channel conflicts. There are two exceptions: (1) when there are more virtual memory banks than physical ones, and (2) when buffer sizes exceed the physical memory capacity. In case (1), Proteus tolerates allocating multiple buffers to the same bank while still attempting to distribute them evenly. In case (2), it attempts memory oversubscription by splitting the buffers into smaller chunks if possible and repeatedly processing all chunks. If these strategies fail, Proteus returns a runtime error.

Figure 7 demonstrates typical memory access patterns of the kernels, which can be described by the Proteus API, and a simplified view of their actual buffer placements on DDR or HBM-equipped FPGAs. We introduce how the above strategy works for each access pattern as follows.

Task and data parallelism. Figure 7(a) shows the task and data parallelism, where four kernels (A-D) are executed in parallel. For HBM, Proteus can effectively distribute kernels' input/output buffers into different banks. On the other hand, for DDR, Proteus tolerates sharing a single memory bank between two kernels, resulting in lower throughput.

Stream processing. Figure 7(b) shows the stream processing, where the kernel continuously processes input data streams from a FIFO and produces data to another FIFO. In this case, Proteus uses double buffering [6] to optimize the kernel's throughput, overlapping kernel computation with host-FPGA data transfers via the Shell's PCIe DMA. Because



(a) Task and data parallelism.

(b) Stream processing (overlapping DMA transfers and kernel execution).

(c) Memory oversubscription.

Figure 7. Proteus's FPGA application models and data placement strategies for different memory architectures.

both DDR and HBM FPGAs can avoid channel conflicts, DDR is expected to achieve higher throughput than HBM due to its higher per-channel bandwidth.

Memory over-subscription. Figure 7(c) shows a memory oversubscription example, where the buffer size is larger than the HBM memory capacity. In this case, Proteus checks if memory oversubscription is possible. Specifically, it uses the kernel's port's minimum data size (*min_datasize* defined in Proteus's memconf) as a lower bound for the chunk size. If the capacity of the memory banks is bigger than the chunk size and the original buffer can be aligned with it, Proteus splits the buffer into chunks. It then repeatedly transfers them and executes the kernel until all data is processed.

Saving/loading FPGA memory states. Proteus's memory virtualization mechanisms enable saving and loading FPGA memory states and provide state-based orchestration services, such as task migration and checkpoint-and-restore, across heterogeneous FPGAs. When saving FPGA states for task eviction or checkpointing, the platform abstraction layer performs the synchronization operation to flush all the on-the-fly kernel execution and data transfers. Then it saves all FPGA contexts (buffer contents and kernel metadata). Afterwards, the Proteus hypervisor also suspends a guest CPU application and saves its contexts if necessary. Proteus targets stateless kernels and does not save internal states (registers, BRAMs) due to the FPGA's non-preemptive nature [66]. Similar to existing systems [94, 136], Proteus handles migration by waiting for all ongoing FPGA executions to finish.

When loading FPGA states for task resuming, the Proteus hypervisor picks up the FPGA and creates the platform abstraction layer. Then the layer recreates all saved buffers and kernels on the target FPGA. The saved FPGA context can be loaded to any FPGA device as long as the kernels and buffers can be mapped, as demonstrated in Figure 7.

3.5 Performance-aware Scheduler

We design a performance-aware scheduler to address the performance gaps across heterogeneous FPGAs. Proteus enables this awareness by performing an offline performance analysis that estimates a kernel's performance on target FPGAs. The analysis results are attached to task deployment requests, allowing the scheduler to select the most appropriate FPGA (node) for each task.

While our scheduler primarily aims to hide FPGA performance heterogeneity, it is designed as a pluggable scheduler for orchestrators such as Kubernetes [58], leveraging the orchestrator's dynamic capabilities to mitigate resource contention, e.g., load balancing and task migration. For example, when faster FPGAs are busy, it can initially deploy tasks

Parameter	Description	Given by
$\mathcal{F}_i$	an FPGA card	board info
$bw^{h2f}_{\mathcal{F}_i}, bw^{f2h}_{\mathcal{F}_i}$	$\mathcal{F}_i$'s PCIe read/write bandwidths	board info
$\mathcal{B}$	a bank of off-chip memory	board info
$bw^{rd}_{\mathcal{B}}, bw^{wr}_{\mathcal{B}}$	$\mathcal{B}$'s memory read/write bandwidths	board info
$\mathcal{K}$	a kernel of user logic	IDE report
$C_{\mathcal{K}}$	an execution latency of $\mathcal{K}$ (cycles)	IDE report
$f_{\mathcal{F}_i}$	$\mathcal{K}$'s clock frequency on $\mathcal{F}_i$	IDE report
$\mathcal{P}^{rd}_j, \mathcal{P}^{wr}_k$	input/output ports of $\mathcal{K}$	memconf
$\mathcal{D}_{\mathcal{P}^{rd}_j}, \mathcal{D}_{\mathcal{P}^{wr}_j}$	input/output data sizes of $\mathcal{P}^{rd}_j, \mathcal{P}^{wr}_k$	memconf
$\mathcal{W}_{\mathcal{P}^{rd}_j}, \mathcal{W}_{\mathcal{P}^{wr}_j}$	port widths of $\mathcal{P}^{rd}_j, \mathcal{P}^{wr}_k$	IDE report

Table 3. Input parameters for the FPGA scoring algorithm.

to slower FPGAs to the extent of user demand and migrate them to faster ones once they become available.

Scoring algorithm. The offline performance analyzer calculates performance scores for heterogeneous FPGAs. Our performance model clarifies relative performance gaps across FPGAs, which arise from factors such as Fmax, off-chip memory bandwidth, and PCIe bandwidth. Table 3 identifies input parameters used for the scoring algorithm. These parameters are given by either specifications of FPGA boards [60, 119, 121], synthesis/implementation reports of FPGA IDEs [123, 126], or user-defined parameters specific to Proteus (memconf).

Assuming that a kernel $\mathcal{K}$ can be deployed on N types of FPGAs, $\mathcal{F}_i$ $(i \in 1, 2, \ldots, N)$, Proteus defines a score, $score_{\mathcal{K},\mathcal{F}_i}$, which represents the end-to-end latency of the kernel execution on a specific FPGA. The latency consists of three steps: (1) host-to-FPGA input data writes, (2) kernel execution, and (3) FPGA-to-host output data reads. We also account for different data access patterns, i.e., non-pipelined memory access and pipelined stream access. For stream-access buffers, data transfers are assumed to overlap with the kernel execution (§3.4). Therefore, we calculate the score as follows:

$$score_{\mathcal{K},\mathcal{F}_i} = \mathcal{T}^{wr}_{\mathcal{F}_i} + \max\left(\mathcal{T}^{sin}_{\mathcal{F}_i}, \mathcal{T}_{\mathcal{K},\mathcal{F}_i}, \mathcal{T}^{sout}_{\mathcal{F}_i}\right) + \mathcal{T}^{rd}_{\mathcal{F}_i} \quad (1)$$

where $\mathcal{T}^{wr}_{\mathcal{F}_i}$ and $\mathcal{T}^{rd}_{\mathcal{F}_i}$ are the latencies of PCIe data transfers for input and output memory-access buffers, $\mathcal{T}^{sin}_{\mathcal{F}_i}$ and $\mathcal{T}^{sout}_{\mathcal{F}_i}$ are the latencies for stream-access buffers, and $\mathcal{T}_{\mathcal{K},\mathcal{F}_i}$ is the latency of the kernel execution. Because stream input/output transfers overlap with kernel execution, their latencies are constrained by the slowest operation. Since each kernel has an arbitrary number of input/output ports, we define M and Q as the total numbers of input and output ports, respectively. To distinguish memory accesses and stream accesses, we define the numbers of input and output ports connected to memory-access buffers as M' $(M' < M)$ and Q' $(Q' < Q)$. Under these conditions, the data transfer latencies can be calculated as follows:

$$\mathcal{T}^{wr}_{\mathcal{F}_i} = \frac{\sum_{j=1}^{M'} \mathcal{D}_{\mathcal{P}^{rd}_j}}{bw^{h2f}_{\mathcal{F}_i}}, \quad \mathcal{T}^{rd}_{\mathcal{F}_i} = \frac{\sum_{k=1}^{Q'} \mathcal{D}_{\mathcal{P}^{wr}_k}}{bw^{f2h}_{\mathcal{F}_i}} \quad (2)$$

$$\mathcal{T}^{sin}_{\mathcal{F}_i} = \frac{\sum_{j=M'}^{M} \mathcal{D}_{\mathcal{P}^{rd}_j}}{bw^{h2f}_{\mathcal{F}_i}}, \quad \mathcal{T}^{sout}_{\mathcal{F}_i} = \frac{\sum_{k=Q'}^{Q} \mathcal{D}_{\mathcal{P}^{wr}_k}}{bw^{f2h}_{\mathcal{F}_i}} \quad (3)$$

Next, we calculate the kernel latency $\mathcal{T}_{\mathcal{K},\mathcal{F}_i}$. Our model assumes that the kernel uses a standard pipelined optimization that enables parallel data reads and writes. Therefore, the kernel latency, which is constrained by the slowest operation of multiple reads and writes, is calculated as follows:

$$\mathcal{T}_{\mathcal{K},\mathcal{F}_i} = \max\left(\max_{1\le j\le M}\left\{\frac{\mathcal{D}_{\mathcal{P}^{rd}_j}}{th_{\mathcal{P}^{rd}_j}}\right\}, \max_{1\le k\le Q}\left\{\frac{\mathcal{D}_{\mathcal{P}^{wr}_k}}{th_{\mathcal{P}^{wr}_k}}\right\}\right) \quad (4)$$

where $th_{\mathcal{P}^{rd}_j}$ and $th_{\mathcal{P}^{wr}_{kw}}$ represent the read and write throughputs of an input port $\mathcal{P}^{rd}_j$ $(j \in 1, 2, \ldots, M)$ and output port $\mathcal{P}^{wr}_k$ $(k \in 1, 2, \ldots, Q)$. The read and write throughputs for each port are affected by two factors. First, it is constrained by the kernel's actual execution latency. For example, if an input port can theoretically achieve 10 GiB/s but the kernel actually consumes data with 8 GiB/s read throughput due to a higher execution latency, the actual throughput is 8 GiB/s. Second, it is constrained by an off-chip memory congestion, which happens when one or more ports are connected to the same memory bank. To consider these factors, $th_{\mathcal{P}^{rd}_j}$ (and $th_{\mathcal{P}^{wr}_k}$) is calculated as follows:

$$th_{\mathcal{P}^{rd}_j} = \sigma^{rd}_{\mathcal{B}} \cdot th'_{\mathcal{P}^{rd}_j} \quad (5)$$

$$th'_{\mathcal{P}^{rd}_j} = \min\left(bw_{\mathcal{P}^{rd}_j}, th^{\mathcal{P}^{rd}_j}_{\mathcal{K}}\right) \quad (6)$$

where $bw_{\mathcal{P}^{rd}}$ is the theoretical read bandwidth of $\mathcal{P}^{rd}_j$, $th^{\mathcal{P}^{rd}_j}_{\mathcal{K}}$ is the read throughput achieved by the kernel computation, and $\sigma^{rd}_{\mathcal{B}}$ is *a congestion factor* of the memory bank $\mathcal{B}$ to which $\mathcal{P}^{rd}$ connects. First, $bw_{\mathcal{P}^{rd}_j}$ is calculated as follows:

$$bw_{\mathcal{P}^{rd}_j} = \mathcal{W}_{\mathcal{P}^{rd}_j} \cdot f_{\mathcal{F}_i} \quad (7)$$

where $\mathcal{W}_{\mathcal{P}^{rd}_j}$ is the bus width of $\mathcal{P}^{rd}_j$. Next, $th^{\mathcal{P}^{rd}_j}_{\mathcal{K}}$ is calculated as follows:

$$th^{\mathcal{P}^{rd}_j}_{\mathcal{K}} = \mathcal{D}_{\mathcal{P}^{rd}_j} \cdot \frac{f_{\mathcal{F}_i}}{C_k} \quad (8)$$

where $C_{\mathcal{K}}$ is $\mathcal{K}$'s execution latency (in cycles) spent to produce one unit of output data, which is given by FPGA IDE's synthesis and implementation reports [123, 126]. Lastly, $\sigma^{rd}_{\mathcal{B}}$ is calculated as follows:

$$\sigma^{rd}_{\mathcal{B}} = \min\left(1, \frac{bw^{rd}_{\mathcal{B}}}{\sum_{j=1}^{M^{rd}_{\mathcal{B}}} th'_{\mathcal{P}^{rd}_j}}\right) \quad (9)$$

Algorithm 1: The performance-aware scheduler.

```
1  select_node(task, bitstreams, fpgas, nodes)
2  begin
       /* Scoring all FPGAs available for the task */
3      foreach fpga in fpgas do
4          if bitstreams.find(fpga) == null then
5              continue;
6          fpga.score ← score_fpga(fpga, task);
7          task.fpgas.insert_sorted_by_score(fpga);
8      end
       /* Return a node equipped with the picked FPGA */
9      picked_node, best_score ← null, fpgas[0].score;
10     foreach fpga in task.fpgas do
11         node ← nodes.find(fpga);
12         if node == null then
13             continue;
14         if fpga.score > α × best_score then
15             break;
16         picked_node ← node;
17     end
18     return picked_node;
19 end
```

where $bw^{rd}_{\mathcal{B}}$ is the maximum read bandwidth of $\mathcal{B}$, and $M^{rd}_{\mathcal{B}}$ is the number of input ports connected to $\mathcal{B}$. Similarly, we can calculate the throughputs for output ports ($th_{\mathcal{P}^{wr}_k}$) from write-specific parameters and obtain $th_{\mathcal{K},\mathcal{F}_i}$.

Scheduling algorithm. Algorithm 1 shows the proposed scheduling algorithm, triggered by new task deployment requests or other events, such as task completion. The scheduler picks a task from a wait queue and assigns it to the best worker node. First, the scheduler sorts all available FPGAs for a given task based on their pre-calculated performance scores in descending order (*task.fpgas*) (L3-8). It then attempts to assign the task to the best-performing FPGA (L9-16). If worker nodes equipped with the highest-scoring FPGA are occupied, the scheduler can pick a slower one. If the selected FPGA's score falls below a user-defined threshold, i.e., $\alpha \times best_score$ ($0 < \alpha < 1$), the task is not assigned to any worker node and returned to the wait queue.

4 Implementation

We implement a prototype of Proteus for various FPGA devices. Our prototype supports two FPGA system stacks of two major FPGA vendors: AMD (Xilinx) and Intel (Altera).

Proteus OS and API. We build the Proteus OS based on the IncludeOS unikernel [16]. We implement a device driver that allows a guest application to request/release FPGAs via hypercalls. We also implement the Proteus API and an OpenCL wrapper library as a part of the guest unikernel library; we port OpenCL header files [48] and implement their function bodies to invoke the corresponding Proteus API.

Proteus hypervisor. We build the Proteus hypervisor on Solo5 [115], extended with an FPGA resource manager and a platform abstraction layer. We add two hypervisor calls that create and destroy platform abstraction layers. Upon the creation, the hypervisor initializes lock-free message buffers to establish direct communications between the Proteus API and the layer. We support two vendor-specific platforms: AMD Vitis/XRT [132] and Intel FPGA SDK for OpenCL [39]. Because these platforms offer only one reconfigurable slot, our prototype currently lacks space-sharing with multiple slots. To enable this, Proteus needs to integrate with multi-slot platforms like Coyote [69], which requires a revision to memory management to dynamically monitor kernel memory usage and allocate resources to prevent contention.

Specs	U50	U280	U280-DDR	S10-emu
PCIe	PCIe Gen3 x16			
Vendor	AMD (Xilinx)			Intel (Altera)
Runtime	Vitis/XRT 2023.2 [132]			OpenCL SDK 20.2 [39]
FPGA IDE	Vivado 2023.2 [129]			Quartus Pro 20.2 [40]
Board	Alveo U50 [121]	Alveo U280 [119]		Stratix 10 GX [60]
Shell	XDMA base_5 [8]	XDMA base_1 [7]		OpenCL BSP [59]
Logic	1,907k	2,852k		2,753k
Memory	HBM2 8 GiB	HBM2 8 GiB	DDR4 32 GiB	DDR4 1 GiB

Table 4. Heterogeneous FPGAs on our testbed.

FPGA memory management. We implement the memory management mechanisms only for AMD FPGAs by using Xilinx OpenCL extensions [131], which allow us to dynamically configure buffer allocation to memory banks. To enable this, when compiling kernel code, Proteus connects all memory banks and the kernel's input/output ports via data paths. This extension is also possible for Intel FPGAs [39, 93], which support memory-mapped I/Os to access individual banks.

Performance-aware scheduler. We implement the performance analyzer for AMD FPGAs using metrics obtained from Vitis HLS synthesis reports [123] and xclbinutil [127]. We also implement the leader-node scheduler and worker-node daemons that handle task deployment and migration. The scheduling algorithm could be implemented as a pluggable orchestrator extension like Kubernetes.

5 Evaluation

We comprehensively evaluate Proteus across four dimensions: application portability (§5.1), performance (§5.2), memory virtualization (§5.3), and scheduler (§5.4).

Experimental setups. We construct a heterogeneous FPGA cluster comprising one leader node without FPGAs and three worker nodes with heterogeneous FPGA devices. Specifically, we use two AMD FPGAs and one Intel FPGA: AMD Alveo U50 [121] and U280 [119], and Intel Stratix 10 GX development kit [60]. Due to the lack of real hardware, the Stratix 10 FPGA is tested only in emulation mode (S10-emu).

Table 4 shows heterogeneous FPGA configurations on our testbed. While U50 has HBM only, U280 has both HBM and

	Application	Abbrv.	LoC (F-API)	Kernel's Fmax (MHz)			
				U50	U280	U280-DDR	S10-emu
Vitis_Accel_Examples	array_partition	ar_part	204 (49)	303	284	314	err
	burst_rw	b_rw	114 (28)	549	518	471	300
	dataflow_func	df_func	118 (30)	565	493	425	err
	dataflow_subfunc	df_subf	118 (30)	532	547	435	err
	helloworld	vadd	117 (33)	536	490	413	510
	lmem_2rw	lmem	119 (31)	522	501	443	500
	loop_reorder	loop_rr	138 (31)	350	350	310	err
	partition_cyclicblock	par_cyc	198 (49)	294	250	250	err
	shift_register	sft_reg	206 (53)	436	451	404	400
	systolic_array	sys_ar	144 (33)	290	250	250	err
	gmem_2banks	gmem	135 (30)	427	426	416	273
	wide_mem_rw	wmem	119 (31)	574	581	413	508
	common lib		713 (11)	-	-	-	-
Rosetta	3d-rendering	3d-rndr	3485 (22)	267	268	250	-
	digit-recognition	dgt-rc	248 (26)	304	250	250	-
	optical-flow	opt-fl	1652 (22)	376	367	346	-
	spam-filter	sp-fltr	417 (25)	526	460	438	-
	common lib		492 (29)	-	-	-	-
mb	wide_mem_rw_2x	wmem2x	195 (36)	549	533	486	-
	wide_mem_rw_4x	wmem4x	195 (36)	517	512	512	-

Table 5. Portability study. *err* indicates that the bitstream generation failed due to insufficient on-chip memory resources. *F-API* refers to LoC for FPGA-related API calls.

DDR. We treat the U280 as two distinct platforms, each with a different off-chip memory (U280-DRAM, U280-HBM), to highlight application portability and performance differences driven by Fmax and off-chip memory characteristics.

Applications. We use two benchmark suites designed for AMD FPGAs: Vitis Accel Examples [124] for various benchmarks, and Rosetta [140] for real-world applications. For the Intel FPGA, we port OpenCL kernels from the Vitis Accel Examples by removing or replacing vendor-specific functions. In addition, we implement two memory-bound benchmarks, wide_mem_rw_2x/4x, which create two and four copies of the wide_mem_rw kernel and execute them in parallel. This parallelization optimization is widely used in real-world applications such as stencil computation [142], fluid simulation [91], and hash calculation [134].

5.1 Portability

First, we examine Proteus's portability and Fmax variances.

Methodology. For each application, we measure the lines of code (LoC) of the entire host code and the FPGA-related API calls (F-API) to examine the engineering effort required to port them to different FPGAs. We also investigate the Fmax of applications' kernels on FPGAs. To achieve the highest possible frequency, we initially set a high target frequency for the IDE's compilation, i.e., 650 MHz. If compilation fails, we incrementally lower the target frequency by 50 MHz until it succeeds.

Results. Table 5 summarizes the results. We observe that each application contains 22-53 LoCs for FPGA operations. Rosetta makes fewer API calls because its API is simplified by a custom shared library (common lib). The results highlight that even porting only the host code is a non-trivial engineering effort. Proteus mitigates this burden by enabling an application to manage any FPGA without additional code changes after initial porting.

Regarding Fmax, we observe clear, often unpredictable variance across different FPGAs, with a gap of up to 168 MHz for wide_mem_rw. We also observe that U280-DDR results in lower Fmax due to differences in the physical layout of off-chip I/O pins. Surprisingly, U50, an inexpensive FPGA card, achieves the highest Fmax for most applications. *In summary*, unpredictable variances of Fmax across different FPGAs underscore the importance of Proteus's cross-FPGA portability.

5.2 Performance

Next, we evaluate the overall performance of Proteus.

Methodology. We measure the end-to-end execution time of applications for three cases: native execution, Arax [94], and Proteus. Arax virtualizes accelerators via a unified API and a user-space runtime that communicates with the vendor's stack. Note that Arax is a library-level abstraction and does not ensure multi-tenant isolation. Due to its limited OpenCL support and porting issues [10], Arax is evaluated only on U50 with 10 Vitis Accel Examples applications. For native and Proteus, we separately measure the time of core FPGA operations (data transfers and kernel execution) using hardware counters. Input data size is unified to 500 MiB, except for sp-fltr (~5 GiB) and dgt-rc (~10 MiB). We measure the average time and the deviation for each application across 10 runs.

To emulate S10, we use the emulation mode offered by Intel's OpenCL SDK [39]. As this mode does not provide performance numbers, we estimate S10's execution time using measurements from U280-DDR, which has a similar hardware type. For the kernel execution, we multiply U280-DDR's measurements by the relative Fmax difference between the two FPGAs. For PCIe data transfers, we use the time measured on U280-DDR. These estimated times replace the corresponding time measurements during the emulation.

End-to-end performance. Figures 8-10 show the total execution times of all applications. For Vitis Accel Examples (Figure 8), we observe performance overheads between 0% and 17.1% on AMD FPGAs, except for sys_ar (up to 37.4%). sys_ar incurs high overhead due to its intensive API calls, requiring 65k kernel invocations for 500 MiB of data, whereas most applications require only 1k invocations. ar_part, loop_rr, and par_cyc also exhibits higher overheads (5.9%-17.1%) with around 16k iterations. Longer execution times on U280-DDR are due to slower reconfiguration. For Rosetta applications

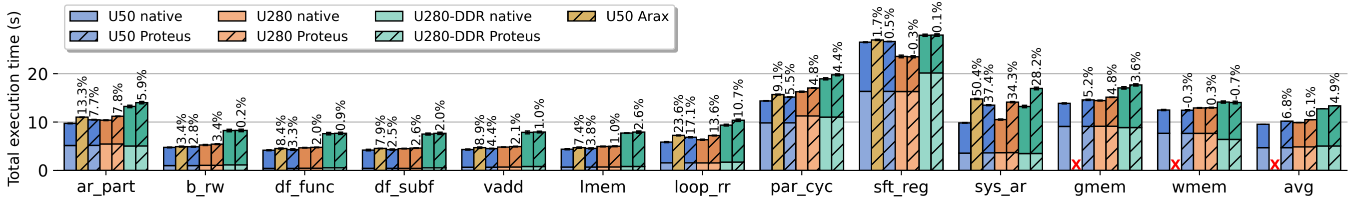


Figure 8. Execution time of Vitis Accel Examples for AMD FPGAs. The lower part of each bar shows the time spent on data transfer to/from the FPGA and kernel execution. The relatives (%) show overheads compared to the native execution.

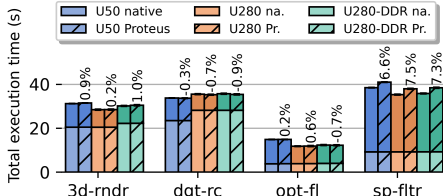


Figure 9. Rosetta for Xilinx FPGAs.

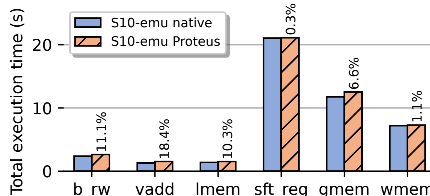


Figure 10. Intel FPGA emulation.

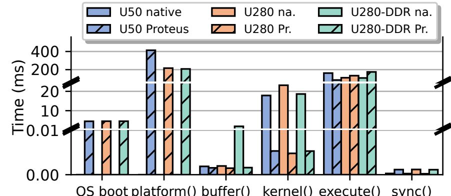


Figure 11. Proteus API overheads.

(Figure 9), they demonstrate smaller overheads (-0.6%-7.1%) because they have fewer kernel invocations. The Intel FPGA emulation (Figure 10) introduces similar overheads to AMD FPGAs (0.3%-18.4%). Some observed speedups are attributed to variances from repeated runs, ranging from 0.8% to 1.6%.

Figure 8 also shows Arax's overheads, 13.4% on average for the 10 ported applications, which is higher than Proteus (8.5%). These overheads are attributed to additional data copies at Arax's transport layer, resulting from data transfers between guest applications, the Arax runtime, and FPGAs. Proteus eliminates data copies because the Proteus hypervisor can directly access guest address space for DMA transfers.

Overhead breakdown. We further analyze the overhead breakdown of Proteus APIs during vadd execution, shown in Figure 11. The two leftmost items are specific to Proteus, representing Proteus OS's bootup time and the creation of the platform abstraction layer. For other API calls, Proteus induces minimal overheads (0.9µs-66.7ms). Proteus is sometimes slightly faster than native because its hypervisor handles FPGA requests asynchronously while guest applications keep running. *In summary*, Proteus introduces reasonable overheads compared to native execution.

5.3 Memory Virtualization

Next, we evaluate Proteus memory management, particularly its three core functionalities: data placement optimization, memory oversubscription, and task migration.

Data placement optimization. We evaluate the effectiveness of Proteus's data placement and transfer optimizations: multi-bank utilization (bank-opt) and stream processing (strm), shown in Figure 7(a) and (b). They are compared to a baseline where all kernel ports connect to a single bank (default), representing the default behavior of a vendor's compiler (i.e., AMD Vitis [125]). We use two memory-bound benchmarks, wmem2x/4x, to fully utilize off-chip memory bandwidths. We set the kernel clock frequency to 300 MHz and measure only FPGA time (data transfer and kernel execution) on U280 to highlight the speedups.

Figure 12 shows the results. The bank-opt achieves 1.24-1.64× speedups for HBM and 1.11-1.22× for DDR. The bank-opt is more effective for HBM because each kernel port has exclusive access to a separate bank. The strm further reduces the time by overlapping data transfers and kernel execution, i.e., 1.75-1.95× and 1.45-1.58× speedups for HBM and DDR, where wmem2x and 4x achieve equivalent performance because PCIe data transfers are dominant in the total time. *In summary*, Proteus improves the kernel's performance by optimizing data placement on heterogeneous memory.

Memory oversubscription. We next evaluate memory oversubscription. We implement a benchmark that saturates the off-chip memory bandwidth using the wmem kernel with two 2 GiB input buffers and one 2 GiB output buffer. To emulate oversubscription scenarios, Proteus sets artificial limits on off-chip memory capacity, ranging from 512 to 4,096 MiB, which are compared to the baseline with no limits (no-limit). We evaluate two buffer types: memory buffers (mem), which perform data transfers and kernel invocations sequentially, and stream buffers (strm), which overlap them. The stream buffers have eight chunks for all cases. For each setup, we measure the data transfer and kernel execution time on U280.

Figure 13 shows the results. Proteus's memory oversubscription introduces 0.1-9.7% overheads for memory-type buffers, and 1.7-17.6% for stream-type buffers. The overheads vary with memory capacity limits, which is attributed to the different chunk sizes configured by Proteus, affecting

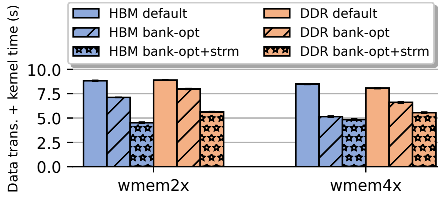


Figure 12. Memory optimization.

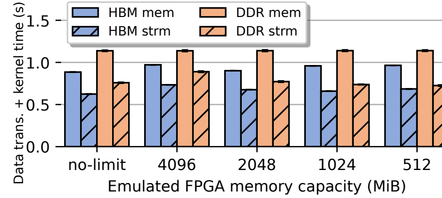


Figure 13. Memory oversubscription.

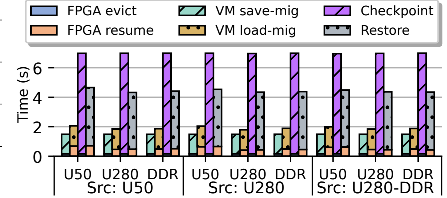
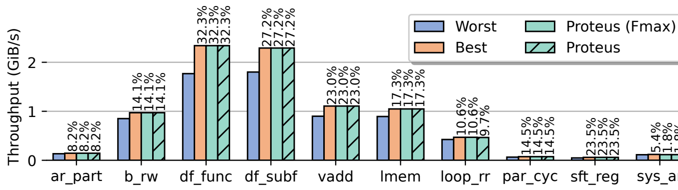
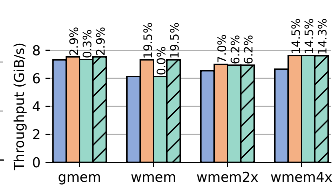


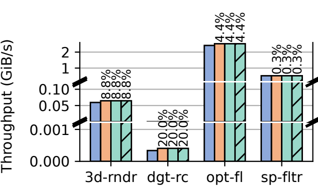
Figure 14. Migration overheads.



(a) Compute-bound.



(b) Memory-bound.



(c) Rosetta.

Figure 15. Throughputs achieved by Proteus's scheduling algorithm with relative gains (%) compared to the worst case.

memory access patterns. *In summary*, Proteus allows an application to use oversubscribed buffers under severe memory capacity limits without significant performance penalties.

Migration. We measure VM migration and checkpointing overheads using sft_reg as a benchmark, performing an FIR filter for an increased dataset (1 GiB). We evaluate all possible combinations of source and destination FPGAs and measure the individual times for the various steps: evicting/resuming FPGA states and saving/loading VM states.

Figure 14 shows the results for two scenarios: VM save/load using host DDR4 memory for storing the migration data, and checkpoint/restore using a persistent SATA SSD. We observe that any combination of source and destination FPGAs has similar overheads, and FPGA state management is not dominant (2.4-11.3% for eviction, 10.1-33.1% for resuming). Two reasonable factors cause these overheads: DMA transfers for saving/loading buffers, and the platform abstraction layer initialization only for resumption. *In summary*, Proteus allows deployed applications to be migrated across heterogeneous FPGAs with reasonable overheads.

5.4 Scheduling

We evaluate our performance-aware scheduling algorithm.

Methodology. To evaluate the accuracy of our scoring algorithm, we measure the overall FPGA execution throughput of applications running on the FPGA selected by the scheduler, including the time spent on host-to-FPGA data transfers. We compare Proteus with three baselines: the best and worst cases (Best, Worst) taken from Figure 8, i.e., an FPGA leading to the lowest/highest FPGA execution time for each application, and a naive approach, Proteus (Fmax), that only uses the Fmax as the performance metric.

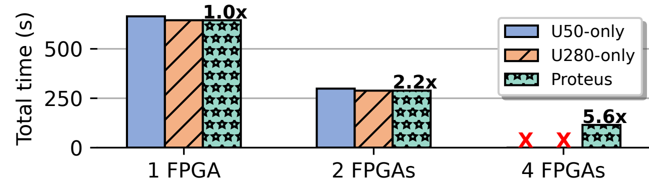


Figure 16. Multi-task workload scenario.

Results. Figure 15 shows the application throughputs on the FPGA selected by different algorithms. Proteus achieves 13.8% of the average throughput gain, which is 0.3% less than the best-case scenario that always chooses the fastest FPGA (14.1% gain). The Fmax-only approach achieves a 12.6% gain, which is sufficient for most CPU-bound applications, including Rosetta, but fails for memory-bound applications (gmem, wmem). It works for wmem2x/4x by chance because HBM FPGAs, which suit these applications, achieve the highest Fmax. We attribute the lack of performance gains for sys_ar to its high runtime overheads. *In summary*, Proteus scheduler accurately estimates the end-to-end kernel throughputs on heterogeneous FPGAs by considering off-chip memory characteristics.

5.5 Multi-task Workloads and Scalability

Lastly, we demonstrate Proteus's workload scalability.

Methodology. We assume a data center scenario in which an orchestrator deploys application replicas to handle multiple requests in parallel. We replicate this on a four-node heterogeneous cluster equipped with two U50 and two U280 FPGAs. Using the Proteus scheduler, we deploy 16 requests of Rosetta's spam-filter and measure the total execution time while varying the number of FPGAs. We compare Proteus with two homogeneous baselines: U50-only and U280-only.

Results. Figure 16 demonstrates that Proteus achieves up to a 5.6× speedup with four FPGAs compared to a single-FPGA setup, attributed to parallel computation and reduced waiting times of the scheduler. While the scalability of homogeneous baselines is limited to two FPGAs, Proteus leverages all FPGA resources. In addition, Proteus is faster than the U50-only as its performance-aware scheduling deploys tasks on faster FPGAs (U280) first. *In summary,* Proteus guarantees high workload scalability on a heterogeneous FPGA cluster.

6 Related Work

FPGA virtualization. Prior FPGA virtualization studies [4, 17, 18, 35, 95] aim to share a single FPGA among tasks by applying OS primitives, such as scheduling [24, 42, 51, 104, 112], memory virtualization [1, 27, 116], security [64, 70], and communication [52–54, 88]. These features are often integrated into Shells for hardware isolation [23, 65, 69, 76]. Funky [71] and F3 [86] propose FPGA orchestration and virtualization frameworks on homogeneous FPGA clusters for cloud-native applications. However, these studies do not address FPGA heterogeneity. AvA [136] and Arax [94] offer a software abstraction to virtualize various accelerators, but only ensure API compatibility. Proteus provides a more comprehensive abstraction of heterogeneous FPGAs, accounting for off-chip memory architectures and performance differences. Furthermore, unlike Arax, which is a simple user-space wrapper library that limits multi-node adoption and lacks multi-tenant isolation, Proteus provides hypervisor-enforced isolation and enables dynamic task assignment across distributed FPGAs.

Heterogeneous FPGA support. HeteroViTAL [138] and Zha et al. [139] extend ViTAL [137] to enable deploying a single, large virtual accelerator on combined partitions of heterogeneous FPGAs. Harmonia [82] achieves cross-vendor FPGA support by offering a modular Shell and customizable software stacks. While they require building custom FPGA stacks, Proteus guarantees application portability across existing FPGA stacks, including those provided in commercial clouds [93, 132]. Proteus also addresses unique challenges, such as data placement across heterogeneous FPGA memory and performance-aware task scheduling.

FPGA memory management. FPGA Shells such as Coyote [69], FSRF [76], and AmorphOS [65] design MMUs to provide memory isolation and virtualization for both host and FPGA memory. FSRF also supports memory oversubscription at a page granularity. In contrast, Proteus provides software-based oversubscription that works on any FPGA platform without dedicated hardware. HeteroFlow [118] proposes an HLS compiler for offline code analysis to optimize the kernel's dataflow across memories, while Proteus adopts a similar yet language-agnostic mechanism to optimize off-chip memory accesses, such as multi-bank utilization.

Task scheduling for heterogeneous accelerators. Task scheduling for heterogeneous systems has been studied for decades. Still, existing studies are either CPU-specific [9, 14, 41, 97, 108, 114], e.g., relying on metrics such as IPCs and cache miss ratios, or application-specific [21, 22, 44, 63, 77, 92, 113], e.g., machine learning on GPUs. In contrast, Proteus proposes an application-agnostic approach to predict the performance of FPGA acceleration across heterogeneous FPGAs. FlexCL [83] also predicts performance through HLS code analysis. Such a detailed analysis could be integrated with Proteus to further improve its scheduling effectiveness.

7 Conclusion

In this paper, we present Proteus, a comprehensive end-to-end system for heterogeneous FPGA virtualization in cloud environments. By decoupling applications from vendor-specific software stacks and hardware topologies, Proteus addresses the critical challenges of bitstream incompatibility, API heterogeneity, off-chip memory heterogeneity, and performance disparities across diverse FPGA products.

Proteus introduces four key innovations that transform a pool of diverse FPGA resources into a unified, flexible, and high-performance acceleration tier: a unikernel-based virtualization stack and hypervisor that abstract underlying hardware to ensure multi-tenant isolation. To streamline development, our platform-agnostic API eliminates vendor-specific host code, while a memory virtualization layer transparently optimizes data placement across DDR and HBM architectures, even supporting memory oversubscription. Finally, a performance-aware scheduler utilizes a predictive scoring model to match workloads with the most efficient FPGA resources based on estimated execution latency.

Our evaluation across AMD and Intel platforms demonstrates that Proteus enables seamless application portability with minimal performance overheads (4.9–6.8%). Furthermore, Proteus's automated optimizations provide up to 1.95× speedups in memory-bound scenarios and a 13.8% gain in overall throughput through intelligent scheduling. As cloud providers continue to adopt increasingly diverse hardware accelerators, Proteus provides the necessary abstraction layer to achieve high resource utilization and developer productivity in the heterogeneous era.

Acknowledgements

We thank our shepherd, Prof. Yubin Xia, and anonymous reviewers for their helpful comments. This work was supported in part 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 Intel Trustworthy Data Center of the Future (TDCoF), and Google Research Grants. The authors acknowledge the financial support by the Federal Ministry of Research, Technology, and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

A Artifact Appendix

A.1 Abstract

Our artifact includes the codebase of Proteus, consisting of the unikernel and hypervisor, as well as the applications we used for our evaluation, i.e., Vitis Accel Examples and Rosetta. Additionally, we provide a repository containing the raw data and scripts for creating plots and tables.

A.2 Description & Requirements

A.2.1 How to access. The artifact is publicly available at https://doi.org/10.5281/zenodo.18848257. You can also find detailed instructions for building the software and replicating the experiments from the paper in the README there.

A.2.2 Hardware dependencies. To run all experiments from the paper, you need four x86-64 servers connected to the same local network, two AMD/Xilinx Alveo U50 [121], and two Alveo U280 [119] FPGA boards.

A.2.3 Software dependencies. We have tested Proteus on Ubuntu 20.04 x86-64 with Linux kernel 5.8.0. Building the software requires gcc, clang, cmake, nasm, OpenCL headers, and XRT [2]. Experiment (E2.1) requires the Intel FPGA SDK for OpenCL [39].

A.2.4 Benchmarks. All benchmark applications are included in the artifact.

A.3 Set-up

Get the artifact:
```
git clone --recurse-submodules \
https://github.com/TUM-DSE/proteus.git
```
Alternatively, download `proteus.tar.xz` from https://doi.org/10.5281/zenodo.18848257.
Set up environment:
```
cd proteus && source env.sh
```
Prepare directories:
```
cd funky-unikernel && mkdir $INCLUDEOS_PREFIX \
&& mkdir build && cd build
```
Set up tap device:
```
$PROTEUS_DIR/funky-unikernel/funky-scripts/\
setup_tap_device.sh
```

In `funky-unikernel/build`:
```
cmake .. -DCMAKE_INSTALL_PREFIX=\
${INCLUDEOS_PREFIX}
```
Build dependencies, including the hypervisor:
```
make PrecompiledLibraries
```
Build the unikernel:
```
make -j $(nproc)
```
Install in `$INCLUDEOS_PREFIX`:
```
make install
```

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Proteus' cross-FPGA portability reduces users' engineering effort and enables applications to run on the best-performing FPGA. This is proven by experiment (E1) described in 5.1 whose results are illustrated in Table 5.
- (C2): Proteus introduces reasonable overheads compared to native execution. This is proven by experiments (E2.1) and (E2.2) described in 5.2 whose results are illustrated in Figures 8-11.
- (C3.1): Proteus improves kernels' performance by optimizing data placement on heterogeneous memory. This is proven by experiment (E3.1) described in 5.3 whose results are illustrated in Figure 12.
- (C3.2): Proteus allows an application to use oversubscribed buffers under severe memory capacity limits without significant performance penalties. This is proven by experiment (E3.2) described in 5.3 whose results are illustrated in Figure 13.
- (C3.3): Proteus allows deployed applications to be migrated across heterogeneous FPGAs with reasonable overheads. This is proven by experiment (E3.3) described in 5.3 whose results are illustrated in Figure 14.
- (C4.1): The Proteus scheduler accurately estimates the end-to-end kernel throughputs on heterogeneous FPGAs. This is proven by experiment (E4.1) described in 5.2 whose results are illustrated in Figure 15.
- (C4.2): Proteus guarantees high workload scalability on a heterogeneous FPGA cluster. This is proven by experiment (E4.2) described in 5.5 whose results are illustrated in Figure 16.

A.4.2 Experiments. We assume you followed the README until "Simple example application" and expect the working directory `$PROTEUS_DIR/funky-unikernel/funky-scripts/evaluation` for all experiments after (E0).

Experiment (E0): Plots [5 human-minutes + 5 compute-minutes]: Generate plots from the data in `$PROTEUS_DIR/proteus-eval/data`.

[Preparation] Install Python requirements:
```
cd $PROTEUS_DIR/proteus-eval && \
pip install -r requirements.txt
```

[Execution] Run all scripts:
```
./scripts/run-all.sh
```

[Results] The plots are saved in `$PROTEUS_DIR/proteus-eval/plots`. Follow the other experiments to recreate the data.

Experiment (E1): Portability [10 human-minutes + 5 compute-minutes]: Get bitstream frequencies, total LoC, and FPGA-related LoC of applications (C1).

[Execution] Get the bitstream frequencies for U50 and U280:
```
./get_bitstream_freq.sh | tee frequencies.csv
```
The frequencies for the Intel Stratix 10 have been collected manually from compilation reports.
Get the total LoC of the applications:
```
cd portability && ./count_loc.sh $PROTEUS_DIR/\
vitis-accel-examples/ocl_kernels $PROTEUS_DIR/\
funky-rosetta
```
Get the FPGA-related LoC:
```
./count_ocl_loc.sh | tee ocl-loc.csv
```

[Results] The frequencies are the "fast" values in `frequencies.csv`. Total LoC and FPGA-related LoC can be found in `portability/loc_*/loc.csv` and `ocl-loc.csv`.

Experiment (E2.1): End-to-end performance [10 human-minutes + 12 compute-hours]: Measure overhead of Proteus for application execution time (C2).

[Execution] Measure the time of native applications with each FPGA configuration and 10 runs per application:
```
./measure_time_native.sh 10 u50-fast u280-fast \
u280-ddr-fast arria10-fast
```
The same for Proteus:
```
./measure_time.sh 10 u50-fast u280-fast \
u280-ddr-fast arria10-fast
```

[Results] The times are saved in the csv files in `time_(native_)<date>_<time>`.

Experiment (E2.2): Overhead breakdown [10 human-minutes + 30 compute-minutes]: Measure overheads of individual Proteus API functions (C2).

[Execution] Get average overheads of 10 iterations:
```
./overheads.py 10
```

[Results] The times are save in `time_overheads_<date>_<time>/overheads.csv`.

Experiment (E3.1): Data placement optimization [10 human-minutes + 1 compute-hour]: Measure time of a memory-heavy application with different levels of memory optimizations (C3.1).

[Execution] Get average times of 10 iterations:
```
./mem_benchmarks.py 10
```

[Results] The times are saved in the csv files in `time_mem_<date>_<time>`.

Experiment (E3.2): Memory oversubscription [10 human-minutes + 30 compute-minutes]: Measure performance of memory oversubscription with various emulated FPGA memory capacities (C3.2).

[Execution] Get average times of 10 iterations:
```
./oversub.py 10
```

[Results] The times are saved in the csv files in `time_oversub_<date>_<time>`.

Experiment (E3.3): Migration [10 human-minutes + 30 compute-minutes]: Measure overheads of individual steps during migration (C3.3).

[Execution] Get average times of 10 iterations:
```
cd state_management && \
./run_benchmark.sh fpga_state_oh 10 && \
./run_benchmark.sh vm_state_oh 10 && \
./run_benchmark.sh migration_oh 10
```

[Results] The times are saved in the csv files in `{fpga_state,vm_state,migration}_oh_<date>_<time>`.

Experiment (E4.1): Scoring algorithm [5 human-minutes + 5 compute-minute]: Measure the accuracy of the scoring algorithm (C4.1).

[Execution] This is included when generating the plots.

[Results] Three plots: `$PROTEUS_DIR/proteus-eval/plots/scheduler-thrp-*.pdf`.

Experiment (E4.2): Multi-task workloads and scalability [1 human-hour + 1 compute-hour]: Measure the total execution time of multiple tasks using one, two, and four FPGAs (C4.2). These experiments require several manual steps. Please refer to the README.

References

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 25–28. https://doi.org/10.1145/1950413.1950421

[2] Inc. Advanced Micro Devices. [n. d.]. XRT Native APIs. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html. [Last accessed: March 24, 2026].

[3] 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

[4] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. 2014. ReconOS: An Operating System Approach for Reconfigurable Computing. *Micro, IEEE* 34 (01 2014), 60–71. https://doi.org/10.1109/MM.2013.110

[5] Amazon. [n.d.]. Amazon EC2 F2 Instances. https://aws.amazon.com/ec2/instance-types/f2. Last accessed: March 24, 2026.

[6] AMD. [n.d.]. Overlapping Data Transfers with Kernel Computation. https://docs.amd.com/r/en-US/ug1700-vitis-accelerated-data-center/Overlapping-Data-Transfers-with-Kernel-Computation. Last accessed: March 24, 2026.

[7] AMD. [n.d.]. U280 Gen3x16 XDMA base_1 Platform. https://docs.amd.com/r/en-US/ug1120-alveo-platforms/U280-Gen3x16-XDMA-base_1-Platform. Last accessed: March 24, 2026.

[8] AMD. [n.d.]. U50 Gen3x16 XDMA base_5 Platform. https://docs.amd.com/r/en-US/ug1120-alveo-platforms/U50-Gen3x16-XDMA-base_5-Platform. Last accessed: March 24, 2026.

[9] Hamid Arabnejad and Jorge G Barbosa. 2013. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE transactions on parallel and distributed systems* 25, 3 (2013), 682–694.

[10] Computer Architecture and VLSI Systems (CARV) Laboratory. [n.d.]. Arax codebase. https://github.com/CARV-ICS-FORTH/arax. Last accessed: March 24, 2026.

[11] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-Compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[12] 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

[13] Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Soham Chakraborty, Deepak Garg, and Pramod Bhatotia. 2024. Toast: A Heterogeneous Memory Management System. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3656019.3676944

[14] Rashmi Bajaj and Dharma P Agrawal. 2004. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems* 15, 2 (2004), 107–118.

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

[16] 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

[17] Gordon Brebner. 1996. A virtual hardware operating system for the Xilinx XC6200. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, Reiner W. Hartenstein and Manfred Glesner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–336.

[18] Robert Brodersen, Artem Tkachenko, and Hayden Kwok-Hay So. 2006. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*. 259–264. https://doi.org/10.1145/1176254.1176316

[19] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. 2020. Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead. *IEEE Access* 8 (2020), 225134–225180. https://doi.org/10.1109/ACCESS.2020.3039858

[20] 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

[21] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[22] Fahao Chen, Peng Li, Celimuge Wu, and Song Guo. 2022. Hare: Exploiting inter-job and intra-job parallelism of distributed machine learning on heterogeneous gpus. In *Proceedings of the 31st international symposium on high-performance parallel and distributed computing*. 253–264.

[23] Jiyang Chen, Harshavardhan Unnibhavi, Atsushi Koshiba, and Pramod Bhatotia. 2024. vFPIO: A Virtual I/O Abstraction for FPGA-accelerated I/O Devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1167–1184. https://www.usenix.org/conference/atc24/presentation/chen-jiyang

[24] Liang Chen, Thomas Marconi, and Tulika Mitra. 2012. Online scheduling for multi-core shared reconfigurable fabric. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 582–585. https://doi.org/10.1109/DATE.2012.6176537

[25] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 116–126. https://doi.org/10.1145/3431920.3439301

[26] 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

[27] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An in-Fabric Memory Architecture for FPGA-Based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/1950413.1950435

[28] Alibaba Cloud. [n.d.]. Alibaba Cloud FPGA Instances. https://www.alibabacloud.com/help/en/doc-detail/108504.html. Last accessed: March 24, 2026.

[29] Alibaba Cloud. [n.d.]. Create an ASIC-accelerated cluster. https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/create-a-managed-asic-accelerated-cluster. Last accessed: March 24, 2026.

[30] Google Cloud. [n.d.]. GPU platforms. https://cloud.google.com/compute/docs/gpus. Last accessed: March 24, 2026.

[31] Linux container projects. [n.d.]. LXC - Linux Containers. https://github.com/lxc/lxc. Last accessed: March 24, 2026.

[32] NVIDIA Corporation. [n.d.]. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Last accessed: March 24, 2026.

[33] Pudi Dhilleswararao, Srinivas Boppu, M. Sabarimalai Manikandan, and Linga Reddy Cenkeramaddi. 2022. Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey. *IEEE Access* 10 (2022), 131788–131828. https://doi.org/10.1109/ACCESS.2022.3229767

[34] Docker. [n.d.]. What is a Container? https://www.docker.com/resources/what-container/. Last accessed: March 24, 2026.

[35] Er Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *in European Control Converence*.

[36] Andre Esteva, Kat Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. 2021. Deep learning-enabled medical computer vision. *npj Digital Medicine* (2021).

[37] Intel FPGA. [n.d.]. Accelerator Functional Unit Developer's Guide for Intel FPGA Programmable Acceleration Card. https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html. Last accessed: March 24, 2026.

[38] Intel FPGA. [n.d.]. Intel FPGA Add-on for oneAPI Base Toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html. Last accessed: March 24, 2026.

[39] Intel FPGA. [n.d.]. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html. Last accessed: March 24, 2026.

[40] Intel FPGA. [n.d.]. Quartus Prime Design Software. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html. Last accessed: March 24, 2026.

[41] Richard F Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D Lima, et al. 1998. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*. IEEE, 184–199.

[42] Wenyin Fu and Katherine Compton. 2008. Scheduling Intervals for Reconfigurable Computing. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. 87–96. https://doi.org/10.1109/FCCM.2008.48

[43] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. 2025. *TNIC: A Trusted NIC Architecture: A hardware-network substrate for building high-performance trustworthy distributed systems.* Association for Computing Machinery, New York, NY, USA, 1282–1301. https://doi.org/10.1145/3676641.3716277

[44] Emmanouil Giortamis, Francisco Romao, Nathaniel Tornow, Dmitry Lugovoy, and Pramod Bhatotia. 2025. Qonductor: A Cloud Orchestrator for Quantum Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 728–745. https://doi.org/10.1145/3712285.3759785

[45] Google. [n.d.]. Cloud Tensor Processing Units (TPUs). https://cloud.google.com/tpu. Last accessed: March 24, 2026.

[46] Google. [n.d.]. The next wave of Google Cloud infrastructure innovation. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu. Last accessed: March 24, 2026.

[47] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 107–122. https://doi.org/10.1145/3567955.3567962

[48] Khronos Group. [n.d.]. OpenCL API Headers. https://github.com/KhronosGroup/OpenCL-Headers. Last accessed: March 24, 2026.

[49] Khronos Group. [n.d.]. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. Last accessed: March 24, 2026.

[50] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218718

[51] Brandon Kyle Hamilton, Michael Inggs, and Hayden Kwok Hay So. 2014. Scheduling Mixed-Architecture Processes in Tightly Coupled FPGA-CPU Reconfigurable Computers. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 240–240. https://doi.org/10.1109/FCCM.2014.75

[52] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. 2019. SemperOS: A Distributed Capability System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 709–722. https://www.usenix.org/conference/atc19/presentation/hille

[53] Matthias Hille, Nils Asmussen, Hermann Härtig, and Pramod Bhatotia. 2020. A heterogeneous microkernel OS for Rack-Scale systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 50–58. https://doi.org/10.1145/3409963.3410487

[54] Chun-Hsian Huang and Pao-Ann Hsiung. 2009. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embedded Systems Letters* 1, 1 (2009), 19–23. https://doi.org/10.1109/LES.2009.2028039

[55] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. 2022. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *IEEE Trans. Comput.* 71, 5 (2022), 1133–1144. https://doi.org/10.1109/TC.2021.3075765

[56] InAccel. [n.d.]. InAccel FPGA orchestrator. https://inaccel.com/fpga-manager/. Last accessed: March 24, 2026.

[57] Docker Inc. [n.d.]. Docker Hub. https://www.docker.com/products/docker-hub/. Last accessed: March 24, 2026.

[58] Google Inc. [n.d.]. Kubernetes (k8s). https://github.com/kubernetes/kubernetes. Last accessed: March 24, 2026.

[59] Intel. [n.d.]. Intel FPGA SDK for OpenCL BSP Support Center Resources. https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-guidance/opencl-bsp-support.html. Last accessed: March 24, 2026.

[60] Intel. [n.d.]. Intel Stratix 10 GX FPGA Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/stratix/10-gx.html. Last accessed: March 24, 2026.

[61] Intel. [n.d.]. Partitioning Buffers Across Different Memory Types (Heterogeneous Memory). https://www.intel.com/content/www/us/

en/docs/oneapi-fpga-add-on/optimization-guide/2023-0/partition-buffer-diff-mem.html. Last accessed: March 24, 2026.

[62] Intel. [n.d.]. Partitioning Buffers Across Multiple Interfaces of the Same Memory Type. https://www.intel.com/content/www/us/en/docs/programmable/683846/22-2/partitioning-buffers-across-multiple.html. Last accessed: March 24, 2026.

[63] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[64] Chenglu Jin, Vasudev Gohil, Ramesh Karri, and Jeyavijayan Rajendran. 2020. Security of Cloud FPGAs: A Survey. arXiv:2005.04867 [cs.CR] https://arxiv.org/abs/2005.04867

[65] 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. https://www.usenix.org/conference/osdi18/presentation/khawaja

[66] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. 2017. Migration of Long-Running Tasks between Reconfigurable Resources Using Virtualization. *SIGARCH Comput. Archit. News* 44, 4 (jan 2017), 56–61. https://doi.org/10.1145/3039902.3039913

[67] 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

[68] 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

[69] 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

[70] Atsushi Koshiba, Felix Gust, Julian Pritzi, Anjo Vahldiek-Oberwagner, Nuno Santos, and Pramod Bhatotia. 2023. Trusted Heterogeneous Disaggregated Architectures. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSys '23)*. Association for Computing Machinery, New York, NY, USA, 72–79. https://doi.org/10.1145/3609510.3609812

[71] Atsushi Koshiba, Charalampos Mainas, and Pramod Bhatotia. [n. d.]. Funky: Cloud-Native FPGA Virtualization and Orchestration. In *Proceedings of the 2025 ACM Symposium on Cloud Computing* (New York, NY, USA, 2026-01-13) *(SoCC '25)*. Association for Computing Machinery, 209–224. https://doi.org/10.1145/3772052.3772226

[72] 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

[73] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4598–4612. https://doi.org/10.1145/3658644.3690323

[74] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[75] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (Sept. 2021), 39 pages. https://doi.org/10.1145/3469660

[76] 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

[77] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: Compute allocation in hybrid clusters. In *Proceedings of the fifteenth european conference on computer Systems*. 1–16.

[78] 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

[79] Ilia Lebedev, Christopher Fletcher, Shaoyi Cheng, James Martin, Austin Doupnik, Daniel Burke, Mingjie Lin, and John Wawrzynek. 2012. Exploring Many-Core Design Templates for FPGAs and ASICs. *Int. J. Reconfig. Comput.* 2012, Article 8 (jan 2012), 1 pages. https://doi.org/10.1155/2012/439141

[80] Topi Leppänen, Leevi Leppänen, Joonas Multanen, and Pekka Jääskeläinen. 2024. Bitstream Database-Driven FPGA Programming Flow Based on Standard OpenCL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 12 (2024), 2257–2268. https://doi.org/10.1109/TVLSI.2024.3458062

[81] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3373087.3375320

[82] Luyang Li, Heng Pan, Xinchen Wan, Kai Lv, Zilong Wang, Qian Zhao, Feng Ning, Qingsong Ning, Shideng Zhang, Zhenyu Li, Layong Luo, and Gaogang Xie. 2025. Harmonia: A Unified Framework for Heterogeneous FPGA Acceleration in the Cloud. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 498–514. https://doi.org/10.1145/3676641.3716259

[83] Yun Liang, Shuo Wang, and Wei Zhang. 2018. FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs. *IEEE Trans. Comput.* 67, 12 (2018), 1750–1764. https://doi.org/10.1109/TC.2018.2840686

[84] 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

[85] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand,

and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (mar 2013), 461–472. https://doi.org/10.1145/2490301.2451167

[86] Charalampos Mainas, Martin Lambeck, Bruno Scheufler, Laurent Bindschaedler, Atsushi Koshiba, and Pramod Bhatotia. 2025. F3: An FPGA-accelerated FaaS Framework. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing* (University of Notre Dame Conference Facilities, Notre Dame, IN, USA) *(HPDC '25)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3731545.3731582

[87] 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

[88] Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, and Christophe Bobda. 2018. FPGA Virtualization in Cloud-Based Infrastructures Over Virtio. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 242–245. https://doi.org/10.1109/ICCD.2018.00044

[89] Microsoft. [n.d.]. NP size series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/fpga-accelerated/np-series. Last accessed: March 24, 2026.

[90] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. 2024. uIO: Lightweight and Extensible Unikernels. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 580–599. https://doi.org/10.1145/3698038.3698518

[91] Kohei Nagasu, Kentaro Sano, Fumiya Kono, and Naohito Nakasato. 2017. FPGA-based tsunami simulation: Performance comparison with GPUs, and roofline model for scalability analysis. *J. Parallel and Distrib. Comput.* 106 (2017), 153–169. https://doi.org/10.1016/j.jpdc.2016.12.015

[92] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[93] Open FPGA Stack (OFS). [n.d.]. Introduction to oneAPI Accelerator Support Package (ASP). https://ofs.github.io/ofs-2024.2-1/hw/common/reference_manual/oneapi_asp/oneapi_asp_ref_mnl/#15-introduction-to-oneapi-accelerator-support-packageasp. Last accessed: March 24, 2026.

[94] Manos Pavlidakis, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. 2022. Arax: a runtime framework for decoupling applications from heterogeneous accelerators. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3542929.3563467

[95] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. 2006. Hthreads: A Computational Model for Reconfigurable Devices. In *2006 International Conference on Field Programmable Logic and Applications*. 1–4. https://doi.org/10.1109/FPL.2006.311336

[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] Andrei Radulescu and Arjan JC Van Gemund. 2000. Fast and effective task scheduling in heterogeneous systems. In *Proceedings 9th heterogeneous computing workshop (HCW 2000)(Cat. No. PR00556)*. IEEE, 229–238.

[98] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: a static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 888–902. https://doi.org/10.1145/3519939.3523719

[99] Patrick Sabanic, Masanori Misono, Teofil Bodea, Julian Pritzi, Michael Hackl, Dimitrios Stavrakakis, and Pramod Bhatotia. 2025. Confidential Serverless Computing. arXiv:2504.21518 [cs.CR] https://arxiv.org/abs/2504.21518

[100] R. Sass, D. Andrews, E. Komp, W. Peck, F. Baijot, E. Anderson, J. Stevens, and J. Agron. 2006. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *2006 14th Annual IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE Computer Society, Los Alamitos, CA, USA, 89–98. https://doi.org/10.1109/FCCM.2006.40

[101] Amazon Web Services. [n.d.]. Deep Dive on Amazon EC2 VT1 Instances. https://aws.amazon.com/blogs/compute/deep-dive-on-amazon-ec2-vt1-instances/. Last accessed: March 24, 2026.

[102] Runbin Shi, Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2022. Exploiting HBM on FPGAs for Data Processing. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 36 (Dec. 2022), 27 pages. https://doi.org/10.1145/3491238

[103] Dilpreet Singh and Chandan K. Reddy. 2015. A survey on platforms for big data analytics. *Journal of Big Data* 2, 1 (2015), 8–8. https://doi.org/10.1186/S40537-014-0008-6

[104] C. Steiger, H. Walder, and M. Platzner. 2004. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11 (2004), 1393–1407. https://doi.org/10.1109/TC.2004.99

[105] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 199–212. https://www.usenix.org/conference/atc18/presentation/thalheim

[106] 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

[107] Donald Thomas and Philip Moorby. 1991. The Verilog Hardware Description Language. Springer Science & Business Media.

[108] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[109] Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, Han-Chieh Chao, Han-Chieh Chao, and Athanasios V. Vasilakos. 2015. Big data analytics: a survey. *Journal of Big Data* 2, 1 (2015), 21–.

[110] VMAccel. [n.d.]. FPGA Cloud. https://vmaccel.com/solutions/. Last accessed: March 24, 2026.

[111] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Shuhai: Benchmarking High Bandwidth Memory On FPGAS. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 111–119. https://doi.org/10.

1109/FCCM48280.2020.00024
[112] Guy Wassi, Mohamed El Amine Benkhelifa, Geoff Lawday, François Verdier, and Samuel Garcia. 2014. Multi-shape tasks scheduling for online multitasking on FPGAs. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. 1–7. https://doi.org/10.1109/ReCoSoC.2014.6861366
[113] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.
[114] Alexander Wieder, Parmod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2012. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 367–381. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/wieder
[115] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud'16)*. USENIX Association, USA, 71–76.
[116] Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. 2015. MATCHUP: Memory Abstractions for Heap Manipulating Programs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 136–145. https://doi.org/10.1145/2684746.2689073
[117] Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos Kotselidis. 2022. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 3869–3882. https://doi.org/10.14778/3565838.3565842
[118] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/3490422.3502369
[119] AMD Xilinx. [n.d.]. Alveo U280 Data Center Accelerator Card Data Sheet. https://docs.amd.com/r/en-US/ds963-u280/Summary. Last accessed: March 24, 2026.
[120] AMD Xilinx. [n.d.]. Alveo V80 Compute Accelerator. https://www.amd.com/en/products/accelerators/alveo/v80.html. Last accessed: March 24, 2026.
[121] AMD Xilinx. [n.d.]. AMD Alveo U50 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html. Last accessed: March 24, 2026.
[122] AMD Xilinx. [n.d.]. AMD Alveo U55C High Performance Compute Card. https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html. Last accessed: March 24, 2026.
[123] AMD Xilinx. [n.d.]. Data Center Acceleration Using Vitis User Guide (UG1700): HLS Synthesis Report. https://docs.amd.com/r/en-US/ug1700-vitis-accelerated-data-center/HLS-Synthesis-Report. Last accessed: March 24, 2026.
[124] AMD Xilinx. [n.d.]. Vitis Accel Examples. https://github.com/Xilinx/Vitis_Accel_Examples. Last accessed: March 24, 2026.
[125] AMD Xilinx. [n.d.]. Vitis Commands and Utilities – Vitis Reference Guide (UG1702). https://docs.amd.com/r/en-US/ug1702-vitis-accelerated-reference/Vitis-Commands-and-Utilities. Last accessed: March 24, 2026.
[126] AMD Xilinx. [n.d.]. Vitis High-Level Synthesis User Guide (UG1399): Implementation Report. https://docs.amd.com/r/en-US/ug1399-vitis-hls/Implementation-Report. Last accessed: March 24, 2026.
[127] AMD Xilinx. [n.d.]. Vitis Reference Guide (UG1702) – xclbinutil Utility. https://docs.amd.com/r/en-US/ug1702-vitis-accelerated-reference/xclbinutil-Utility. Last accessed: March 24, 2026.
[128] AMD Xilinx. [n.d.]. Vitis Tutorials: Hardware Acceleration (XD099), Using Multiple DDR Banks. https://docs.amd.com/r/en-US/Vitis-Tutorials-Vitis-Hardware-Acceleration/Using-Multiple-DDR-Banks?tocId=IWhuAHv~GOwMcd70TkINxg. Last accessed: March 24, 2026.
[129] AMD Xilinx. [n.d.]. Vivado ML Edition. https://www.xilinx.com/products/design-tools/vivado.html. Last accessed: March 24, 2026.
[130] AMD Xilinx. [n.d.]. Xilinx Base Runtime. https://github.com/Xilinx/Xilinx_Base_Runtime. Last accessed: March 24, 2026.
[131] AMD Xilinx. [n.d.]. Xilinx OpenCL extension. https://xilinx.github.io/XRT/master/html/opencl_extension.html. Last accessed: March 24, 2026.
[132] AMD Xilinx. [n.d.]. XRT and Vitis Platform Overview. https://xilinx.github.io/XRT/master/html/platforms.html. Last accessed: March 24, 2026.
[133] AMD Xilinx. [n.d.]. XRT Native APIs – Buffer allocation and deallocation. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html#buffer-allocation-and-deallocation. Last accessed: March 24, 2026.
[134] Yang Yang, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. 2020. A High Throughput Parallel Hash Table Accelerator on HBM-enabled FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 148–153. https://doi.org/10.1109/ICFPT51103.2020.00028
[135] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/young
[136] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. 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, 807–825. https://doi.org/10.1145/3373376.3378466
[137] 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
[138] Yue Zha and Jing Li. 2021. Hetero-ViTAL: a virtualization stack for heterogeneous FPGA clusters. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 470–483. https://doi.org/10.1109/ISCA52012.2021.00044
[139] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/3445814.3446699
[140] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta:

A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 269–278. https://doi.org/10.1145/3174243.3174255

[141] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 409–420. https://doi.org/10.1109/SC.2016.34

[142] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/3174243.3174248