Kernel-as-a-Service: A Serverless Programming Model for Heterogeneous Hardware Accelerators

Anonymous Author(s)

Abstract

With the slowing of Moore's law and decline of Dennard scaling, computing systems increasingly rely on specialized hardware accelerators in addition to general-purpose compute units. Increased hardware heterogeneity necessitates disaggregating applications into workflows of fine-grained tasks that run on a diverse set of CPUs and accelerators. Current accelerator delivery models cannot support such applications efficiently, as (1) the overhead of managing accelerators erases performance benefits for fine-grained tasks; (2) exclusive accelerator use per task leads to underutilization; and (3) specialization increases complexity for developers.

We propose adopting concepts from Function-as-a-Service (FaaS), which has solved these challenges for general-purpose CPUs in cloud computing. Kernel-as-a-Service (KaaS) is a novel serverless programming model for generic compute accelerators that aids heterogeneous workflows by combining the ease-of-use of higher-level abstractions with the performance of low-level hand-tuned code. We evaluate KaaS with a focus on the breadth of the idea and its generality to diverse architectures rather than on an in-depth implementation for a single accelerator. Using proof-of-concept prototypes, we show that this programming model provides performance, performance efficiency, and ease-of-use benefits across a diverse range of compute accelerators. Despite increased levels of abstraction, when compared to a naive accelerator implementation, KaaS reduces completion times for fine-grained tasks by up to 96.0% (GPU), 68.4% (FPGA), 98.6% (TPU), and 34.9% (QPU) in our experiments.

1 Introduction

Modern computing tasks increasingly require integrating elements from data analytics and machine learning (ML) [20, 21]. Combined with the decline of Moore's law this leads to increased hardware heterogeneity in compute clusters and the cloud with the adoption of accelerators such as graphics processing units (GPUs), or field-programmable gate arrays (FPGAs) to meet the increasing performance demands of applications while adhering to power constraints [19]. Additionally, the introduction of unconventional accelerators such as quantum processing units (QPUs) or wafer-scale engines [81], and increasingly federated and distributed cluster deployments [49], lead to the disaggregation of applications into workflows of smaller, more fine-grained tasks that can be dynamically composed and deployed on heterogeneous infrastructure [93]. This heterogeneous landscape limits the usability of special-purpose frameworks and creates opportunity and demand for a high-performance programming paradigm that incorporates any accelerator equally well. The predominant accelerator deployment paradigm carves out tasks that can benefit from hardware accelerations, namely, the *kernels*[1], and requires the developer to tailor them to a specific accelerator interface. This approach limits our ability to decompose and recompose workflows into constituent computational tasks that map independently to hardware, because these tasks maintain the control of computational resources in the individual application level, not the system's complete view of accelerator demand. We argue this model is inadequate for broad adoption for heterogeneous workflows and architectures, leading to pessimal resource allocation, poor abstraction choices, and gratuitous overhead.

Figure 1. Workflow example with three tasks: image preprocessing, bitmap conversion, and ML inference on the image raster. These tasks run most efficiently on heterogeneous hardware: preprocessing on CPUs, bitmap conversion on FPGAs, and inference on GPUs.

Furthermore, buying or renting (e.g., as part of a cloud VM) a complete compute accelerator leads to expensive underutilization of hardware for small tasks on increasingly capable accelerators [95]. Newly introduced space-sharing models, e.g., Multi-Process Service (MPS) for Nvidia GPUs [26, 55] or Coyote for FPGAs [40], emulate the abstraction of processes on hardware accelerators, yet still impose a per-process runtime initialization overhead, such as the GPU context creation. This overhead is prohibitive for fine-grained, short-lived tasks, as demonstrated in our motivating example workflow in Fig. 1. It combines small tasks of image processing that are best run on diverse hardware: preprocessing on a general-purpose CPU, bitmap conversion on an FPGA, and deep-learning inference on a GPU. The results in Fig. 2 show 24.1% (FPGA) and 98.3% (GPU) initialization overheads that lead to overall worse performance when using accelerators over a CPU-only implementation. While optimized implementations of this workflow can reduce this performance overhead, they require expensive tuning, increase complexity, and multiply development costs.

In this paper, we propose translating the benefits of the Function-as-a-Service (FaaS) programming model to the domain of heterogeneous hardware accelerators. The FaaS

[1]By *kernel*, we specifically mean a core part of the computation that executes on the accelerator and not, e.g., an operating system kernel.

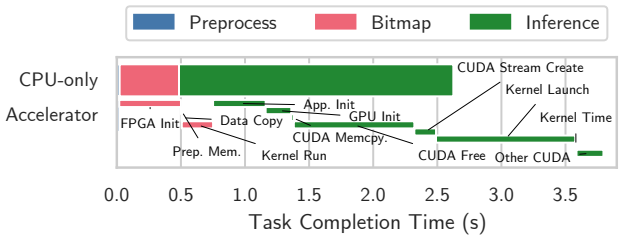


Figure 2. Completion time of tasks in Fig. 1 on two 10-core Intel® Xeon® E5-2650 v3 CPUs (2.30GHz), 1.5TB memory, Xilinx® Alveo™ U250 FPGA with 32GB memory, and Nvidia A100 80GB GPU. Comparing a CPU-only implementation with normal use of accelerators, the overhead of executing small tasks makes hardware acceleration infeasible: Copying data and running the kernel accounts for only 75.9% (FPGA) and 1.7% (GPU) task completion time, with the majority of the overhead attributed to accelerator initialization.

model has demonstrated that increased sharing of resources between applications can increase resource efficiency of running workflows while decreasing management complexity [30]. Although approaches to integrating specific accelerators with FaaS platforms exist [50, 67], we ask the opposite question: how can the abstraction of FaaS benefit hardware accelerators to support workflows in federated heterogeneous computing? To answer this question, we make the following contributions in this paper:

- We introduce the Kernel-as-a-Service (KaaS) serverless programming model, transferring the ideas and benefits of FaaS to hardware accelerators (§3).
- We present a proof-of-concept implementation of KaaS for GPUs, FPGAs, TPUs, and QPUs (§4).
- Using this prototype, we evaluate the KaaS concept for performance, elasticity, and scalability and demonstrate it on a heterogeneous range of accelerators (§5).
- We critically discuss the implications and limitations of this approach, and suggest further research directions in this area (§6).

2 Background

Compute Acceleration and Kernels. The well-known slowing of Moore's law brings about a technological shift in cloud and high-performance computer architectures [19]. Although CPUs still show modest annual gains in performance, the most significant performance-per-Watt improvements are now obtained using heterogeneous architectures, via special-purpose compute accelerators [74]. These accelerators take many forms, primarily as specialized architectures optimized for a subset of computations. Today, the most common of these architectures are general-purpose GPUs, but other types of accelerators are on the rise, especially for ML workloads. Examples include Tensor Processing Units (TPUs), Data Processing Units (DPUs), Quantum Processing Units (QPUs), and FPGAs. Instead of offering the all-around excellent programmability and performance for general workloads of CPUs, they all offer significantly better performance-per-Watt for specific workloads only, which often require expert programming and tuning [91].

Concomitantly with the hardware evolution, we also witness an evolution in software. To benefit from heterogeneous architectures, programmers replace monolithic programs with a hybrid model, where some logic runs on the host CPU and some on accelerators. This accelerated code is often represented as *kernels*, a fundamental algorithm that has been ported to run efficiently on the accelerator [39]. Matrix multiplication, e.g., is a well-known kernel that maps efficiently to GPUs and is commonly used in deep-learning libraries.

This programming model, however, still carries limitations such as difficulty to program, overhead, and poor scalability. To address some of these limitations, we propose a further evolution of this programming model. The key idea of this paper is to encapsulate kernels—analogous to the evolution seen in CPUs with the FaaS model—and to apply the encapsulation to a broad set of architectures.

Function-as-a-Service. FaaS is a "serverless" delivery model for general-purpose cloud compute resources [72] that is the latest evolution of infrastructure sharing among cloud applications: where virtual machines have abstracted physical servers and containers have abstracted from the notion of exclusive compute nodes, FaaS platforms abstract away the entire backend runtime by providing standardized environments shared across applications [30]. In FaaS, developers provide only function code that is transparently executed by the service provider in response to external events or invocations. This model abstracts resource management, e.g., invocation routing and resource (de-)allocation, reducing application footprints and decreasing development and operational costs [47]. It also fosters the specialization of teams for individual services and tasks in a workflow, increasing agility and developer productivity [12, 52]. The vertical integration of infrastructure and platform increases resource efficiency for providers through more efficient bin-packing [28, 60].

FaaS functions are short-lived and stateless, which leads to a theoretically unlimited horizontal scalability, as functions are not hindered by synchronization [82]. Combined with short creation times, a FaaS system can achieve high elasticity by scaling out as the number of invocations increase and releasing resources once invocations end [41]. Further, the burden of securing and isolating services is shifted to the FaaS platform provider. As function invocations must pass through the platform, additional security measures such as access control can be implemented and updated outside the actual application, separating concerns [53].

3 Kernel-as-a-Service Design

We propose KaaS, a serverless programming model for hardware accelerators based on FaaS.

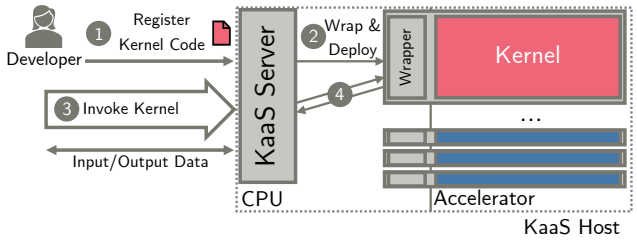


Figure 3. KaaS follows the FaaS programming model but extends it to heterogeneous hardware accelerators: ① Developers register accelerator kernel code on the KaaS host. ② KaaS then wraps this kernel code to expose it as a KaaS kernel, deploying it and other kernels on the host. ③ Applications invoke kernels over the network in a request/response pattern. ④ The KaaS server directs those invocations to copies of the kernel on the accelerator.

3.1 Overview

In KaaS, applications register kernels that implement a task for a shared pool of accelerators. Through a shim, applications can then invoke that kernel, optionally providing input arguments and receiving output data from the kernel, as shown in Fig. 3. Thus, an entire workflow can be disaggregated into constituent kernels, e.g., in our motivating example (Fig. 1), one kernel can handle image preprocessing on CPU cores for all users, another kernel converts all images to bitmaps on FPGAs, and a third classifies images on GPUs. Each kernel is assigned to a computing resource based on its needs and on availability. Applications are written as a collection of portable and device-agnostic kernels, as opposed to mixed and hand-tuned code. These kernels can persist in memory (as FaaS containers do), and handle their workloads through function invocations, with minimal initialization overhead and optimized communication pipelining. Conceivably, each kernel can also manage the accelerator on its own, including resource management, obviating the need for the user or the OS to coordinate multiple resources. Resource-management considerations such as fairness, data isolation, scheduling, and service-level agreements can thus vary across devices and deployments. Note that this large exploration space is an active research area and outside the scope of this paper; our main goals in this experimental evaluation are to demonstrate the potential benefits from overhead elimination and to prove the feasibility of the KaaS concept. That said, this disaggregated model of resource management opens the door to even higher efficiencies with fine-grained scheduling, e.g., offered by MPS on GPUs, with the applications and OS remaining agnostic of the sharing.

The KaaS approach is analogous to how functions in FaaS platforms are first registered and then invoked dynamically. Today, without explicit sharing support (Fig. 4a), applications either use the accelerator for its entire lifetime or "rent" it for a specified amount of time before a new application can use it, e.g., using cloud VMs. Existing space-sharing paradigms, e.g., MPS on GPUs or Coyote on FPGAs, split accelerators by allocating fixed device portions for different processes (Fig. 4b). With KaaS, we propose increasing the level of sharing to also share the runtime across different kernels (Fig. 4c). The platform delegates the control of an accelerator to the KaaS runtime that uses it for a single application kernel. The runtime can be written by the platform provider or application developer, and from the application's point of view, looks like any other library call. In fact, some user-level code can be simplified because it no longer needs to manage hardware resources at a low or non-portable level, as those tasks are delegated to the optimized runtime. Once assigned to an accelerator, the KaaS runtime alone controls allocation of the device resources, much like an OS manages all devices on a server, including scheduling, security, and interfaces.

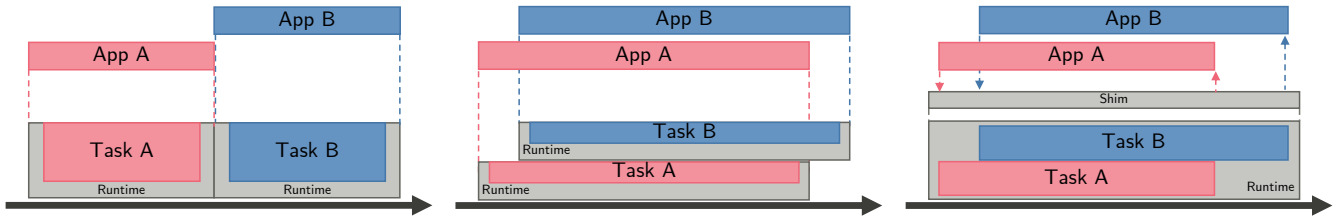
3.2 Performance

Although an additional layer of virtualization cannot improve on the optimal performance of a compute accelerator, abstractions can help developers write more efficient implementations of their services. This is especially important in heterogeneous computing, where managing the complexity of novel compute architectures is a major challenge inhibiting broad adoption of accelerators. Although libraries such as *TensorFlow* for ML [1] or tools such as *PyLog* [32] already provide an abstraction for programming accelerators, integrating these tools into larger workflows is still challenging. As our motivating example in §1 shows, initialization overheads from tools, libraries, and accelerator runtimes negate the theoretical benefits of accelerating compute tasks, especially as task granularity grows finer.

A key benefit of abstracting the runtime management of accelerator tasks in KaaS is moving this overhead out of the critical path of workflows. In FaaS, the first function invocation causes a *cold start*, where libraries and runtime are initialized at the cost of additional latency [82]. The majority of requests can then be served by a warm copy of the function instance at near-native latency, as runtime and libraries are already initialized. Similarly, an initial KaaS kernel invocation might be *cold*, incurring the discussed initialization overheads, but subsequent invocations benefit from running in existing runtimes. Crucially, this performance improvement is achieved *without additional tuning by developers.*

3.3 Efficiency

In addition to the performance benefits for a specific kernel or application, KaaS can provide efficiency gains for the system as a whole. First, a fine-grained hardware sharing model allows providers to serve multiple kernel invocations from the same device. The fine-grained resource allocation of kernel instances in both space (number and footprint of instances) and in time (how long an instance is active) means that only resources that are in use are actually deployed, and



(a) Sharing in Time **(b)** Sharing in Space **(c)** Shared Runtimes (KaaS)

Figure 4. Increasing levels of sharing: In time sharing, tasks run exclusively on the accelerator, each providing their own runtime (Fig. 4a). With space-sharing, using approaches such as MPS, multiple processes can run on the device in parallel, although potentially slowing down each individual task (Fig. 4b). With KaaS, multiple tasks are launched into a shared runtime through a shim, reducing the management overhead of individual runtimes and abstracting from the device (Fig. 4c).

few compute resources are idle, requiring less hardware overall [13, 60, 66]. Second, this resource sharing model could lead to a fine-grained pay-per-use model similar to FaaS, lowering the barrier to entry for applications running on hardware accelerators, compared to overprovisioned cloud machines that incur costs for a full accelerator device. Third, as KaaS kernels are transparently invoked over the network, even across nodes and devices, applications benefit from improved elasticity and horizontal scalability. A larger number of concurrent kernel invocations can be served by a proportionally larger number of kernel instances. If a device cannot serve the concurrent kernel requests, additional device can be added to serve these requests. A workflow executed in response to incoming data, e.g., scientific images, can process each data item independently and scale with the rate of incoming data, similarly to FaaS workflows [37].

3.4 Usability

In KaaS, we map the concept of workflows to the composition of heterogeneous kernels and traditional software. Abstracted kernels can be dynamically recombined or updated for new hardware. Combined with their transparent execution, this yields a flexible deployment method for distributed and heterogeneous infrastructure.

The potential reusability improvements are more pronounced in KaaS than in FaaS: Hardware accelerators can be difficult to program, especially with diverging best practices, interfaces, and performance optimizations for different types, makes, or even generations of accelerators. With KaaS, componentization and composability create a separation of concerns: Each standalone kernel can be implemented and optimized for its target hardware independently of the rest of the application or workflow. The underlying target hardware may also be upgraded without changing the entire application. This abstraction also allows for transparent polyglot software, e.g., a C-based FPGA kernel implementation and a Python-based GPU kernel integrated in a workflow. Further, library kernels similar to optimized implementations found in `numba` [42] or PyLog [32] can be integrated easily. From disaggregation also follows simplified dependency management, as each task of a workflow requires only the specific dependencies for its own target accelerator.

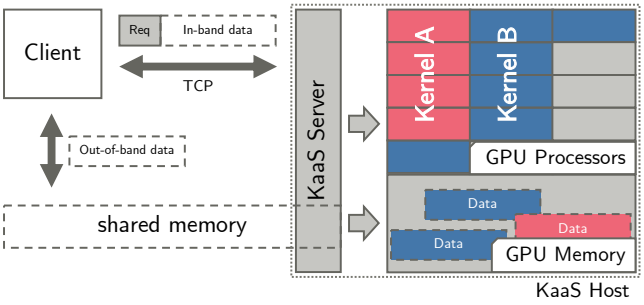


Figure 5. Our GPU prototype runs a KaaS server alongside an Nvidia GPU. Using Nvidia MPS, the server can launch concurrent instances of registered kernels on the GPU. Clients access these kernels through a request over TCP, transferring data in-band (serialized) or out-of-band (e.g., using shared memory within a host). Data is copied to GPU memory using the CUDA interface. Kernels are registered as Python functions and called transparently to the KaaS server.

4 Prototype Implementation

The proposed approach is a model for execution, and we do not prescribe a specific implementation. That said, we develop a set of minimal prototypes to evaluate its feasibility across a range of hardware accelerators. Please note that our prototypes serve only as a proof-of-concept, and we have decided against integrating them in any of the available open-source FaaS platforms, which incur high overheads in workflow management [7, 64, 71, 75] that would obscure performance measurements of our runtime.

4.1 Components

As shown in Fig. 5 for the GPU implementation, there are three main components to the KaaS architecture: the KaaS server, task runners on the accelerator, and a client API.

KaaS Server. The KaaS server is a Python service running on the CPU of a host equipped with accelerators. It exposes

configuration and invocation endpoints and manages task runners on accelerators. When a kernel is invoked, the KaaS server starts a new runner to execute it (cold start) or uses an existing runner with sufficient resources (warm start).

Task Runner. The task runners are Python-based host processes combining developer-provided kernel code with a wrapper that exposes a TCP server. Using this endpoint, kernel invocations are sent from KaaS server to task runner, where the provided code is executed once and results are returned. Task runners can share an accelerator using the abstractions provided by its API, such as MPS on Nvidia GPUs.

Client API. Similarly to the connection between KaaS server and task runners, the client API is also exposed over TCP. Clients send their kernel invocations using this endpoint, but with the KaaS abstraction, clients need not be aware of which specific hardware executes their invocation. As in FaaS, clients can use the TCP-based invocation to send data directly to kernels using in-band data transfer (faster for small data) or send only pointers to data using out-of-band data transfer. This allows transferring larger data without copying over the network. For our initial KaaS prototype, which focuses on single-node hosts, a shared memory region may be defined by the client, which can then be accessed by the task runner by providing a pointer to that region.

Despite our initial focus on single-node deployments, the loose, TCP-based coupling between client, KaaS server, and accelerator host means that a distributed deployment is also feasible. Instead of shared host memory, RDMA may be used for out-of-band data transfer here (§6).

4.2 Accelerators

While the basic architecture of KaaS is fixed, we must adapt the task runner implementation for each accelerator API. Specifically, we develop prototypes for accelerator versions we expect to encounter in datacenters, such as GPUs, FPGAs, TPUs, and QPUs.

GPU. This prototype supports Nvidia GPU kernels written in Python using `numba` [42] for the CUDA interface. Our task runners share the GPU using Nvidia MPS, allowing multiple runners to launch kernel invocations in parallel.

FPGA. Our FPGA prototype uses a backend based on the *PyLog* [32] Python-based FPGA programming and synthesis library. Incoming requests are fed to the Xilinx FPGA kernel through its PYNQ interface [4]. PyLog does not currently allow for spatial sharing of FPGAs, but an extension with systems such as Coyote, which divide FPGAs into isolated regions that can be individually configured, is possible [40].

TPU. Each Google Cloud tensor processing unit (TPU) board contains multiple dual-core TPU chips, which can be controlled individually [38]. Running multiple processes on the same TPU chip, however, leads to errors. Our KaaS prototype for TPUs thus allocates one task runner per TPU chip and distributes requests across these chips evenly.

QPU. Quantum compute units (QPU) are still in their infancy, and it is unclear if they can ever be widely deployed [79]. Nevertheless, it is useful to explore how these QPUs may be integrated into future compute architectures and software stacks. We build our KaaS prototype with support for the IBM Qiskit runtime [33, 88] in order to deploy quantum programs on both simulators and existing real quantum computers. This interface, however, does not allow fine-grained control over hybrid applications that combine traditional and quantum computations.

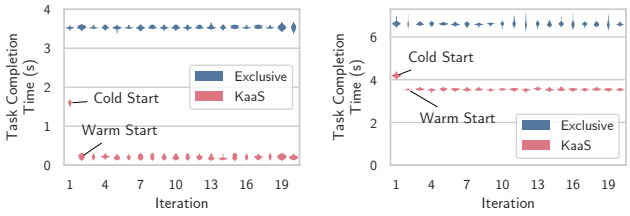
5 Evaluation

We use our prototype implementations to evaluate six aspects of KaaS on diverse hardware accelerators. We explore the general performance of kernels in KaaS (§5.1), evaluate possible energy efficiency gains (§5.2), test the possibility of transparent, remote invocations (§5.3), investigate scalability (§5.4), measure autoscaling capabilities (§5.5), and benchmark kernels with KaaS on GPU, FPGA, TPU, and (simulated and real) QPUs (§5.6). Unless otherwise noted, our experiments use the GPU prototype of KaaS on a server with two 20-core Intel® Xeon® E5-2698 v4 CPUs (2.20GHz), 1TB of memory, and four Nvidia Tesla P100 PCIe GPUs with 56 streaming multiprocessors and 16GB of memory each. We measure the total task completion time and the kernel time, i.e., time spent just on data copy and computation. Total task completion time further includes launching the client Python program, or generating the input data where applicable. For each metric, our plots show the mean and 95% confidence interval for ten samples of each measurement.

5.1 Performance

To evaluate the performance impact of cached kernel copies in KaaS, we implemented a matrix-multiplication kernel on top of our KaaS GPU prototype using `numba` CUDA [42]. We control task granularity using the matrix dimensions $N \times N$ that we use as input to our kernel.

Cold & Warm Starts. We first compare completion times for a small and large matrix multiplication task in KaaS and "exclusive" GPU usage. In the exclusive model, the program has full access to the GPU, without Nvidia MPS enabled. Our small task multiplies two 500×500 matrices (52 million FLOP), while the large task multiplies two $10{,}000 \times 10{,}000$ matrices (402 billion FLOP), both repeated 20 times. The results in Fig. 6 show how KaaS kernels have to pay a cold-start penalty on their first invocation that accounts for 87.1% and 15.5% of their total task completion time for our small and large computation. Nevertheless, even though the accelerator has to be initialized on that first invocation, the



(a) Small task size (matrix dimensions 500 × 500)

(b) Large task size (matrix dimensions 10,000 × 10,000)

Figure 6. The task completion time remains unchanged for small (Fig. 6a) and large (Fig. 6b) tasks across 20 iterations in exclusive GPU use. Using KaaS incurs a "cold start" for the first invocation, yet subsequent invocations benefit from an existing warm copy. This decreases invocation overhead by 94.1% and 46.4% for small and large tasks, respectively.

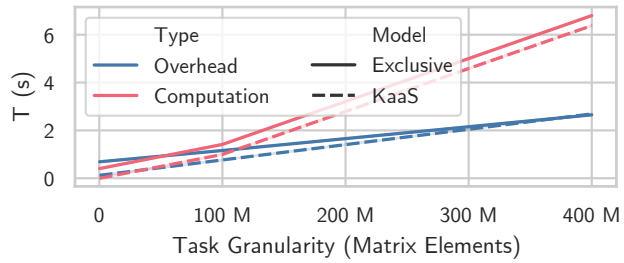


Figure 7. Breaking down the time spent on completing the matrix multiplication task with different input dimensions reveals a significant reduction in computation overhead (e.g., host memory allocation or importing libraries) for small tasks using KaaS compared to exclusive GPU use. For a 500 × 500 matrix, this overhead is reduced from 689ms to 123ms.

kernel is already registered in host memory and large dependencies such as numba are initialized, making a KaaS cold start 54.6% and 36.9% shorter than exclusive execution. A further factor is that our client code has no need to import the numba dependency, as all heavy computation is offloaded to the accelerator through KaaS. Subsequent invocations in KaaS are 94.1% and 46.4% faster than the exclusive model for small and large tasks. As we consider "cold" starts the outlier in frequent invocations of kernels, we focus on "warm" performance in the subsequent evaluation.

Warm Start Overheads. We further explore this overhead with additional task sizes in Fig. 7. *Computation* is the kernel time of the task execution, whereas *Overhead* is the difference between this time and the total task completion time. We also remove the time to generate our random input matrix, which takes between 2ms and 3,619ms, depending on the matrix dimensions. The KaaS approach reduces general computation time by 406ms to 419ms, regardless of task size. We expect this reduction to be caused by the additional CUDA initialization that has to be performed for each execution in

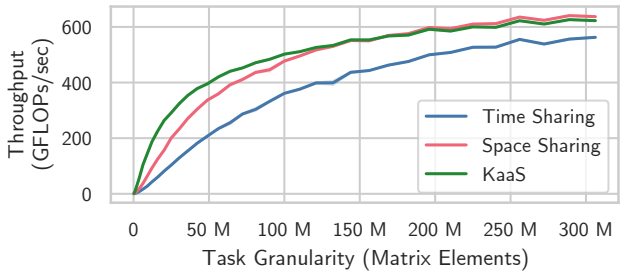


Figure 8. For small tasks, the reduction in overhead per task execution leads to higher accelerator throughput with KaaS over space sharing with MPS and time sharing with exclusive GPU access. As we increase task size, throughput for KaaS converges with that of MPS, which our prototype is based on. Since single instances of matrix multiplication cannot saturate the GPU, time sharing is always inferior to parallelization regarding throughput in our tests.

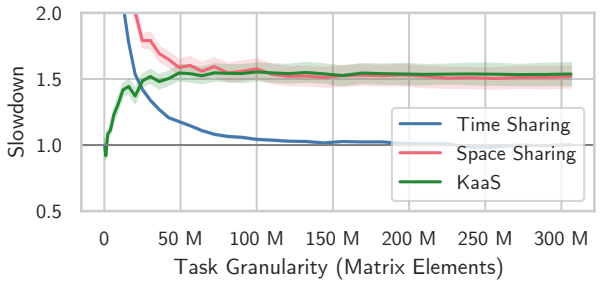


Figure 9. Comparison of per-task kernel completion time for eight parallel task executions with that of an isolated KaaS execution at all given task granularities. Observe that while KaaS can effectively multiplex the available accelerators at small task input sizes, space and time sharing incur large slowdowns as a result of the additional "cold start" overhead. As expected, we again see that KaaS and MPS converge for larger tasks sizes, where exclusive GPU use yields the best per-task execution time for coarse granularities.

the baseline model. For the small task of multiplying matrices of dimension 500 × 500, this reduction leads to a 469× faster execution. While the overhead reduction is significant in small tasks, e.g., from 689ms to 123ms with 500 × 500 matrices, the overhead for both tested models are equal for the largest tested task (matrix dimensions 20,000 × 20,000). Here, the KaaS model incurs overhead for the additional data movement that is needed for kernel invocation. These results show that the kinds of optimizations KaaS enables are more relevant to finely-grained tasks.

Performance by Level of Sharing. Finally, we consider the throughput and slowdown of time sharing, space sharing, and KaaS for our kernel at different task granularities, as controlled by the input dimensions. We increase request concurrency to eight, which yields two concurrent computations per GPU installed in our system. Time sharing is

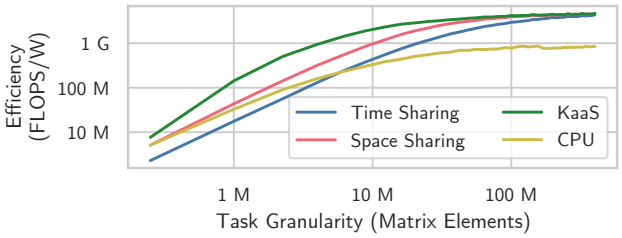


Figure 10. The reduced overall execution time also reflects in increased energy efficiency of KaaS compared to time and space sharing. We additionally compare a CPU-based implementation that is more energy-efficient than GPU time sharing for small tasks but always performs worse than KaaS.

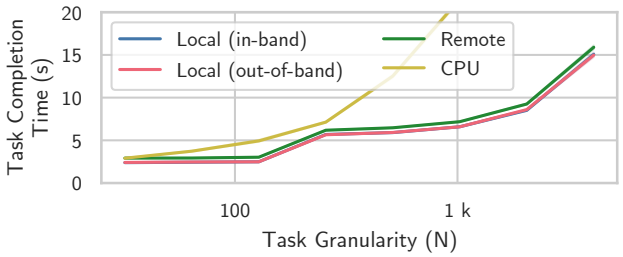


Figure 11. Total task completion time of the GA kernel compared between (1) client executing the kernel on KaaS on another host with serialized data transfer (*Remote*), (2) with the host itself executing the kernel on KaaS with serialized data transfer (*Local in-band*), and (3) out-of-band shared memory transfer (*Local out-of-band*), and (4) with local *CPU* execution. Results show that despite additional network transfer overhead, remotely calling a kernel GPU through KaaS can perform better than local CPU execution.

the exclusive model we used before, which forces a concurrent execution to wait for all other computations to finish before it starts. Additionally, we enable Nvidia MPS on our system to facilitate space sharing, allowing multiple kernel invocations to occur on the device at the same time.

In Fig. 8 we show throughput measurements for square input matrices ranging from 250k to 324M elements. Throughput is measured as FLOP spent on matrix multiplication on the GPU divided by total task completion time. In our tests, KaaS has a larger throughput advantage for smaller task sizes. With increasing task sizes, throughput converges to that of spatial sharing, as our prototype is based on MPS. As the time sharing model cannot benefit from parallelization or overlapping computation and data movement, throughput remains inferior even for larger tasks. Note that we measure real throughput, including task execution overhead, and can thus not achieve the theoretical maximum advertised throughput of our GPUs.

We further show the slowdown in kernel time for each of these experiments in Fig. 9. This slowdown compares the kernel time with that of an isolated execution of the kernel with KaaS at a specified input matrix dimension. While additional concurrent executions of the kernel can increase throughput with parallelization, the slowdown metric captures the impact of that concurrency on the execution time of individual tasks. The time sharing model can achieve computation without slowdown for larger tasks, as no concurrent kernel executions cause resource contention. For smaller tasks, i.e., lower than 50M matrix elements, the lower computation overhead in KaaS leads to faster kernel completion times than in spatial sharing. Again, the results show that faster kernel execution in KaaS through the concept of cached kernel copies benefits smaller tasks in particular.

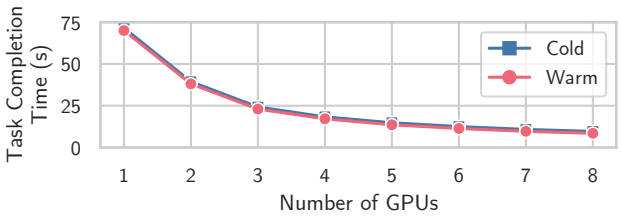
5.2 Energy Efficiency

Using FLOPS/W as a metric for performance efficiency [76], we measure the energy consumption of eight concurrent matrix multiplication task executions. Again, we compare KaaS with the time and space sharing models and vary task granularity with the input data dimensions. We additionally compare a CPU-only execution using the same `numba` implementation as in our GPU tests. Although the hardware of CPU and GPU is not directly comparable, we use this as an indication of whether accelerators are also beneficial from a performance-efficiency point of view. We use the Denki extension of Performance Co-Pilot (PCP) [48] to measure power draw, collecting metrics using the processors' Running Average Power Limit (RAPL) [57] at 1ms intervals and GPU power consumption at 10ms intervals. We divide the kernel computation FLOPS by the measured drawn energy, which includes invocation overheads. GPU idle power consumption is not included in CPU measurements.

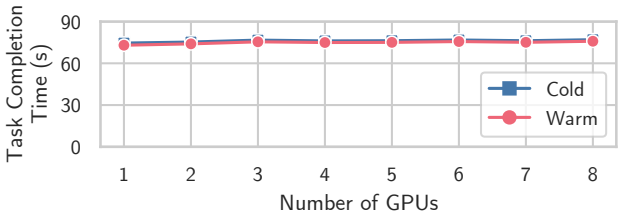
The results in Fig. 10 show a convergence of performance efficiency between the GPU usage models for larger tasks, where CPU performance lags behind at 4 GFLOPS/W compared to 0.7 GFLOPS/W for GPUs. For smaller tasks, however, performance efficiency varies between usage models. The shorter total task completion times of KaaS lead to the best efficiency and at the extreme, only KaaS can beat a CPU-only execution in terms of energy requirements. Nevertheless, the efficiency for smaller tasks is orders of magnitude lower than that of large tasks given the remaining constant overheads.

5.3 Transparent Remote Invocation

As we invoke KaaS kernels over the network, we can also invoke them across nodes, e.g., from a host without accelerators to a server that exposes a GPU through KaaS. We evaluate this potential with the genetic algorithm kernel (*GA*, detailed in §5.6.1). We connect a client with two 32-core AMD EPYC™ 7513 CPUs (2.60Ghz) and 4TB memory to the original GPU testbed with a 1Gbps Ethernet connection (0.15ms RTT). We compare four scenarios to evaluate the efficiency of remote kernel invocations with task granularities between $N = 32$ and $N = 4{,}096$. First, the client invokes the kernel on the GPU server with serialized transfer of input



(a) In strong scaling, overall task size remains constant (8,000 batches of eight images). Results show near-linear scaling with additional GPU devices.



(b) In weak scaling, we vary task size proportionally to the number of GPU devices (8,000 batches of eight images *per* GPU), distributed by round-robin scheduling. Results show near-linear scaling.

Figure 12. Scaling the inference kernels across GPUs within a host reveals near-linear strong (Fig. 12a) and weak (Fig. 12b) scaling. This is a result of the small task size that can be transparently scheduled across GPUs without communication overheads between devices.

and output data (*remote* invocation). Second, we compare local invocations on the GPU host with serialized data transfer to account for the effect of using the network (*local/in-band* invocation). Third, we use the original data transfer over shared memory in the local invocation to measure the overhead of serialization (*local/out-of-band* invocation). Fourth, we execute the kernel on the client CPU to compare if offloading to a GPU-enabled KaaS server offers a benefit over local execution.

We show the overall task completion times for these four experiments at different task granularity in Fig. 11. While we cannot observe a difference in execution time between in-band and out-of-band data transfer, accessing KaaS over the network adds a delay of 490ms to 832ms to the kernel invocation. Nevertheless, it is clear that for the hardware we use, local CPU execution is inferior to remote GPU execution of the kernel. CPU execution is five times slower than remote kernel invocation at the largest task size we test, although admittedly similar in run time for smaller tasks, likely because the kernel's time is too small (48.6%) to impact the overall execution time.

5.4 Scalability

We evaluate strong and weak scaling of cold and warm invocations in KaaS using a *PyTorch* [58] *ResNet-50* [29] inference kernel on a node with eight Nvidia Tesla V100 SXM2 GPUs with 80 streaming multiprocessors and 32GB memory each.

Strong Scaling. We first investigate strong scaling, observing the completion time of 8,000 batches of eight input images to the inference kernel on between one and eight GPU devices. As shown in Fig. 12a, there is a static mean 1.22s cold start overhead. As GPUs can be initialized in parallel, this affects task completion times in all experiments equally. When removing this constant overhead ("warm" start), scaling is nearly linear, with total task time decreasing to an eighth with eight GPUs (8.49s compared to 70.02s).

Weak Scaling. Weak scaling is similar to scale-out in FaaS, where more resources can be added to handle additional demand. We measure the completion time of the inference of $N \times 8{,}000$ batches of eight images on N GPUs (again between one and eight). We use a round-robin scheduler that equally distributes the input batches to devices. The results in Fig. 12b again show near-linear scaling, with total task completion time proportional to the number of input vectors divided by the number of devices (between 74.52s for one GPU and 76.95s for eight GPUs). Further, we again note a constant offset of (parallelized) GPU initialization between "cold" and "warm" starts.

5.5 Autoscaling

A basic autoscaling technique in FaaS is monitoring in-flight requests and scaling the number of function handlers accordingly [44]. We adopt a similar technique in KaaS, yet we note that this serves mainly to illustrate that task runners in KaaS can be automatically scaled to meet elastic demand – just as in FaaS, more sophisticated techniques are possible.

We run a matrix multiplication kernel using input matrices of dimension $10{,}000 \times 10{,}000$ on KaaS on the same eight-GPU host as in §5.4. We issue a growing number of parallel kernel invocations to this system, increasing by one parallel client every ten seconds. Preliminary experiments have shown that each GPU can handle four such tasks in parallel without significant impact to total task execution time. KaaS will monitor the number of in-flight requests and start an additional task runner on new GPU when all existing task runners already handle four requests. We measure task completion time, throughput, and number of task runners started by KaaS.

The results in Fig. 13 confirm how automatically scaling task runners to accommodate a growing request rate can dynamically adapt resource requirements. As we increase the number of parallel clients, new task runners are started. Interestingly, KaaS does not start a new runner at exactly every increment of four parallel clients: Instead, as some work (receiving a response, logging it, and preparing the next invocation) is done on the client, KaaS can use this turnaround time to schedule more parallel work on the GPU than the four in-flight request we limit it to. At the time we reach 32 parallel clients (the theoretical limit), only seven GPU task runners are allocated that handle all parallel requests.

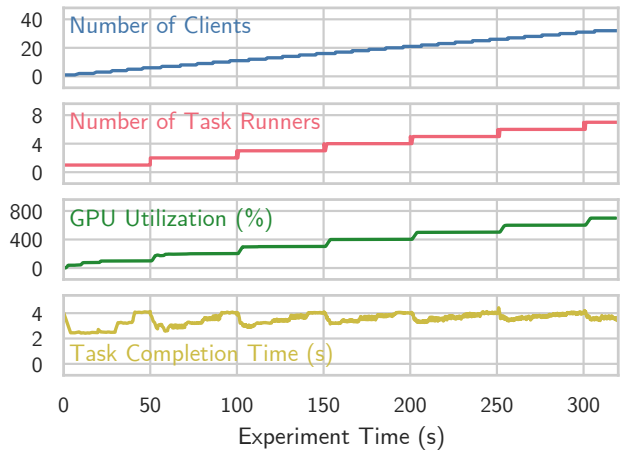


Figure 13. Scaling KaaS resources across eight GPUs with a growing number of parallel requests: KaaS automatically allocates new task runners to meet demand, keeping GPU utilization high on each allocated GPU and task completion time steady for clients.

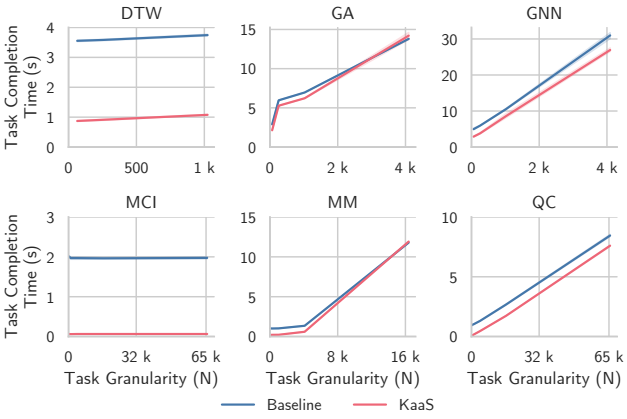


Figure 14. Comparison of task completion times of dynamic time warping (*DTW*), genetic algorithm (*GA*), graph neural network training (*GNN*), Monte Carlo integration (*MCI*), matrix multiplication (*MM*), and quantum computing simulator (*QC*) kernels between Nvidia GPUs with MPS (baseline) and KaaS. KaaS successfully reduces task execution times by up to 96%. Only GA with a high number of generations does not benefit from KaaS with execution time increasing by 5.8%.

Furthermore, we can also observe that mean task completion time remains steady (although of course it dips slightly when a new runner is started), leading us to conclude that the KaaS model can successfully automatically scale resources to meet demand.

5.6 Heterogeneity

Finally, we evaluate our prototypes for a heterogeneous range of hardware accelerators using kernels specific to these devices. As outlined in §4, we consider GPUs (§5.6.1), FPGAs (§5.6.2), TPUs (§5.6.3), and QPUs (§5.6.4)

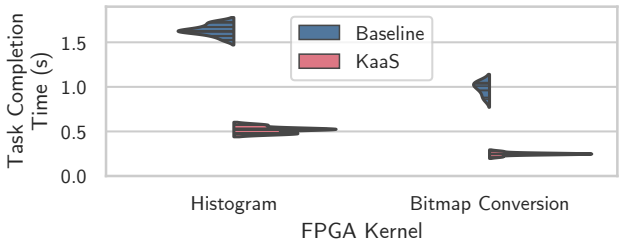
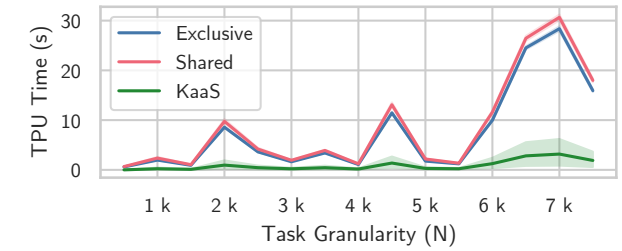


Figure 15. The mean total task completion time of the kernel on our FPGA is reduced by 68.5% (*Histogram*) and 74.9% (*Bitmap Conversion*) through the use of KaaS, which keeps the FPGA and PyLog initialized for subsequent executions.

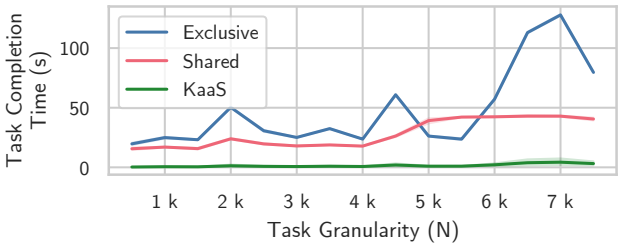
5.6.1 GPU. We compare a baseline of space sharing with KaaS in our GPU prototype using different kernels. In addition to the matrix multiplication kernel (*MM*), we add dynamic time warping (*DTW*), a genetic algorithm (*GA*), graph neural network training (*GNN*), Monte Carlo integration (*MCI*), and a quantum computing simulator (*QC*). DTW is implemented as *softDTW* [15] to calculate the optimal distance between 200 batches of random ten-element sequences of a given input length N. The GA iteratively mutates a population of N 100-element vectors ten times, using a fitness function optimized for GPUs. GNN training uses the *Deep Graph Library* (DGL) [83] and *PyTorch* to perform node classification with a 2-layer graph convolutional network on the DGL *Core Graph Dataset*, and we adapt the number of training iterations as N. The MCI implementation estimates the value of the definite integral $\int_1^{10} 1/x \; dx$ using N samples [86]. The QC simulation uses the *StateVector* method of the *AerSimulator* of the Qiskit runtime [33, 88] to simulate quantum circuits of N CX gates combined in circuits.

The results in Fig. 14 show that KaaS can consistently decrease the task completion times in our tests, up to a reduction of 96% for MCI. The kernel implementations benefit from the optimizations we have shown for the matrix multiplication kernel in §5.1. Only GA with a high number of generations (4,096) shows execution times increasing by up to 5.8% with KaaS. Interestingly, our measurements here reveal variability in the performance of GPUs in our cluster: While the baseline always uses the first installed GPU (default `numba` behavior), KaaS will try to allocate tasks evenly across the GPUs, leading to a difference in task completion time of 1.85s (14.3% slower) between GPUs. While the KaaS invocations on the first GPU are 0.86s (6.2%) faster than in the baseline, the high variability, likely exacerbated by the iterative nature of GA with frequent data copy between host and accelerator, leads to a worse *mean* task completion time.

5.6.2 FPGA. As an example for using the KaaS model on FPGAs, we employ two *PyLog* kernels: *Histogram* computes a histogram of integer values between 0 and 255 for a random array of length 2,097,504. We further use the *Bitmap Conversion* of Fig. 1. We compare the total execution time between



(a) TPU Time



(b) Task Completion Time

Figure 16. Our TPU kernel performs four parallel 2D convolution using matrices of dimensions $N \times N$ on a four-chip Google Cloud TPU. We compare an exclusive use with each kernel execution queuing to use the TPU, a shared use with each kernel instance using a fixed, distinct TPU chip, and an extension of our KaaS prototype for TPUs.

running these kernels by directly accessing the FPGA from the test Python program (*Exclusive*) and executing on the KaaS FPGA prototype. Note that we do not measure FPGA IP configuration, which is on the order of tens of seconds and is usually completed outside of kernel invocations. Our test system is a Xilinx® Alveo™ U250 data center FPGA with 64GB off-chip memory connected over PCIe to a host with four 18-core Intel® Xeon® E7-8890 v3 CPUs (2.50Ghz) and 3TB memory. The results in Fig. 15 show that the same benefits previously explored for GPUs extend to FPGAs, with mean FPGA time reduced by 68.5% (*Histogram*) and 74.9% (*Bitmap Conversion*). Note that these reductions are still orders of magnitude from the performance of hand-tuned FPGA kernels as KaaS cannot optimize the performance of Python and *PyLog*. For reference, hand-tuned kernels show completions times between 80 and 100ms on our test system.

5.6.3 TPU. Our kernel performs a 2D convolution on an $N \times N$ matrix using the appropriate implementation in the TensorFlow library [2]. We test this on a Google Cloud `v3-8` TPU VM in the `us-central1-a` region with a four-chip, eight-core TPU v3 with 16GB of memory per chip, two 24-core Intel® Xeon® CPUs (2.00GHz), and 340GB memory.

We show the impact of different accelerator use models in Fig. 16, with four parallel instances of the kernel. In exclusive TPU use, each kernel execution blocks the entire TPU. As shown in Fig. 16a, this leads to a faster kernel execution on the TPU compared to shared TPU use, where each concurrent instance of the task is assigned one of the four TPU chips for execution. Interestingly, the TPU execution time does not scale proportionally with the input data size. This effect remains consistent across repetitions of the experiment, and we attribute it to internal optimizations that TensorFlow makes in choosing a convolution implementation based on the input parameters. While this effect is also present with our KaaS implementation, the overall kernel execution time is reduced between 81.3% and 99.6% compared to the exclusive execution. As with other accelerators we have tested, we attribute this reduction to the removal of TPU management overhead from the critical path of kernel execution.

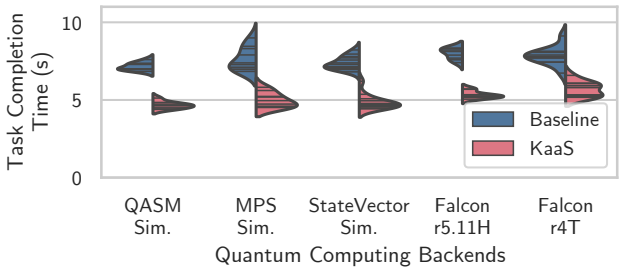


Figure 17. The KaaS model reduces mean task completion time by 34.9% for the QASM simulator, 34.8% for the MPS simulator, 34.3% for the StateVector simulator, 33.3% on Falcon r5.11H, and 27.3% on the Falcon r4T.

A large part of the total task completion time shown in Fig. 16b is the time required to import the necessary libraries, most notably TensorFlow. As this cannot happen in parallel in the exclusive TPU use, as TensorFlow initialization also initializes the available TPU, the overall task time follows a scaled pattern of the kernel execution time. In the shared model, some parallelization is possible as each concurrent kernel invocation has access to a specific TPU chip. The KaaS prototype performs best, as initialization overhead of TPU and TensorFlow is moved out of the critical path, reducing overall task completion time by between 95.9% and 98.6%.

5.6.4 QPU. As a preliminary evaluation of KaaS on QPUs, we implement a single point electronic structure calculation using the Variational Quantum Eigensolver (VQE) [62]. The "quantum kernel" in this application is an estimator primitive, while transpilation occurs on classical hardware. We specifically compare "cold starts" of our quantum operation as a baseline against the possibility of calling into cached copies of the kernel, part of the KaaS approach. As VQE is an iterative process, reductions in kernel overhead accumulated over multiple repetitions can decrease overall execution duration.

We deploy three simulation environments: Quantum Assembly Language (*QASM*), a 32-qubit simulator, Matrix Product State (*MPS*), a 100-qubit simulator, and *StateVector*, a 32-qubit Schrödinger wave function simulator. Further, we deploy on two quantum computers through IBM Quantum, namely on *Falcon r4T* and *Falcon r5.11H* processors with five and seven superconducting qubits, respectively [65].

The results in Fig. 17 are preliminary yet promising, as they show the benefits of the KaaS approach with regard to task completion times and encourage further investigation.

6 Discussion & Future Work

In this section, we critically discuss questions on generality, applicability, and limitations of the KaaS approach.

Applications. The applicability of KaaS is, of course, limited to workloads that can benefit from the introduction of hardware acceleration. Beyond this general limitation, the caveats of running applications as compositions of individual functions and kernels must be considered, similarly to the FaaS execution model [37, 68]. For example, long-running tasks that can span across multiple homogeneous compute resources, e.g., by exploiting heavy parallelism, may be better served by traditional accelerator programming models.

Nevertheless, we foresee a range of domains adopting heterogeneous infrastructure and distributed workflows in the future, especially in the fields of high-performance computing, data analytics, and AI [19]. Examples of these demanding workflows with a heterogeneous range of tasks are already starting to emerge in research: visual simultaneous localization and mapping (SLAM) [9, 78] for AR and metaverse applications combines real-time image decoding and feature extraction tasks that can be accelerated with GPUs and FPGAs [18, 24] in addition to general CPU components. ML-steered ensemble simulations combine the power of simulation tasks, e.g., in quantum chemistry [10], with machine learning steering to reduce the total number of necessary computations [85], with each task benefiting from a different kind of compute accelerator. Earth observation workflows combine processing on domain-specific compute units on satellites with image processing units on Earth for optimized data processing pipelines [16, 31, 80].

Implications for Computer Architectures. Changing how applications interface with hardware accelerators also creates opportunities for hardware-software co-design. One current trend is disaggregating compute resources [5, 49], where computation in a system is not centered on CPUs but rather heterogeneous hardware communicating over an interconnect. Here, KaaS can serve as an abstraction to these resources, as the composition of kernels happens at the logical software level compared to the composition of disparate compute resources at the hardware [59].

Higher levels of sharing also increase flexibility in accelerator deployment. For example, a larger GPU is easier to subdivide, but smaller devices could transparently be used for smaller kernels and service providers could increase margins if these devices can be created more economically. Variability in GPUs can also be abstracted from at the platform level, obviating the need for load balancing by applications [77].

Our current KaaS prototypes rely on TCP connections to invoke kernels using a frontend running on CPUs. A more efficient co-designed implementation may rely on advanced methods for signaling, such as RDMA [14], to further reduce the invocation overhead in both delay and required host resources. Here, smartNICs [11] could be employed for load-balancing, data de-serialization, or accelerator configuration, increasing the efficiency of kernel invocations further by completely eliminating the need for a CPU.

Data Movement. In KaaS as in FaaS, the composition of individual tasks into larger workflows requires data movement between components. This data-shipping architecture of FaaS has been well-researched [51, 64, 68, 70, 96]. While avoidable through data-aware scheduling and the adoption of a function-shipping architecture, current hardware architectures require data movement in KaaS as data must still be copied to hardware accelerator memory for kernel executions. Although our evaluation has shown that this price is worth paying for higher kernel performance and performance efficiency, we believe that new hardware architectures using technologies such as fabric-attached memory will further improve performance for heterogeneous workflows [69]. The possibility of kernel fusion, where two adjacent kernels targeting the same accelerator are combined to minimize data movement, could also be explored in the future [71].

Security & Isolation. The disaggregation of accelerators described in §3 could also lead to a more flexible security model. Without KaaS, applications have no assurances that other users of the accelerator are safe, so the typical security model requires either exclusive use for a single application [3] or some sort of static partitioning such as offered by Multi-Instance GPU [43, 56], which both come with their own inefficiencies and limitations. In contrast, in the KaaS model an accelerator could be allocated to run only a single application, under the assumption that some workloads are large and busy enough to justify dedicating entire accelerators to a single kernel. Access to both code and data on the accelerator is strictly controlled by the runtime only, so sharing a device across users can offer additional safety assumptions which lead to finer-grained resource allocation and isolation and, consequently, higher efficiencies.

That said, the model still allows for a range of security policies including exclusive use of the accelerator by a single user, so it does not limit users to a single security policy. With additional development, this model also open the door to an integrating resource management, quality-of-service (QoS), and security with a flexibility that is not typically offered by off-the-shelf operating systems. For example, CHERI enables both data and execution memory protection in hardware [54], e.g., for fabric-attached memory [8]. Recently, CHERI has been delivered on ARM processors and for the Kitten operating system [27]. Hardware protection for KaaS would allow transparent partitioning of accelerators in both memory and

cores, leaving interconnects as a common resource, yet still requiring additional support for QoS.

Scheduling and Resource Management. Incidentally, the same delegation of selected OS responsibilities to the KaaS runtime also offers renewed potential for additional flexibility in scheduling and resource management. The runtime can collect focused execution histories and can thus better estimate the resource and QoS requirements of kernel requests, leading to better decisions on how to allocate computational resources in space and time [73, 97]. Even at the datacenter level, KaaS opens the door to extending energy-efficient resource management approaches based on predictive usage, such as dynamically shutting down portions of a compute cluster [13, 66]. These aspects are outside the scope of this paper and remain open for future research.

Dynamic Optimizations. Through centralizing resource management in KaaS and the resulting economy of scale, dynamic optimizations are possible that are not feasible for individual applications. For example, the overlapping of multiple kernel invocations, possibly of different tenants or applications, across accelerators could reduce idle time of accelerators, further decreasing energy consumption. Further, the provider could dynamically replace kernel implementations or hardware resources where possible. Without reconfiguring an application or workflow, newer generations or different SKUs of a hardware accelerator could be swapped in to increase performance efficiency [59, 60].

7 Related Work

The FaaS paradigm has taken off recently with an explosion of commercial offerings and open-source software, followed by extensive academic research [72]. Like application kernels in heterogeneous computing, serverless functions are often fine-grained and require careful orchestration to optimize resource utilization [34, 41, 47]. Some work also specifically addresses the use of accelerators (predominantly, GPUs) in serverless computing [46, 50, 67, 87].

Despite the similarities, our work is not strictly about cloud or serverless computing—from which we borrow our main ideas—but rather about heterogeneous computing using accelerators. Offloading an algorithm's core to run on accelerators to obtain higher performance and energy efficiency is an active area of research across computer science. Examples include dense [23] and sparse [25] matrix operations (especially in the context of deep learning), computer vision algorithms [63], and genetic algorithms [36].

While kernel offloading by a single application is a relatively well studied problem, there remain several technical challenges to the efficient sharing of accelerator resources across applications or VMs [35, 45, 92]. This sharing can take place in the time dimension (multiple processes accessing the complete accelerator in different time slots); in the space dimension (multiple processes accessing subsets of the accelerator at the same time); or both. As a commercial offering, HPE *GPUaaS* [22] allows dynamically (re-)assigning full GPUs to containers in a private cluster. For security reasons, they do not allow space-sharing of a single GPU. *GSLICE* [17] spatially multiplexes the GPU to increases system throughput. *Pagoda* [90] maximizes GPU throughput by co-scheduling kernels. Zhang et al. [94] carefully allocates the computation resource to colocated applications, maximizing their throughput while adhering to QoS constraints.

In time sharing (preemptive multitasking), Wu et al. [89] propose *FLEP*, which transforms GPU kernels into preemptible forms which can be interrupted during execution and yield all or part of a GPU's execution units. A different emphasis on preemption to maintain real-time constraints is recently proposed by Ayala-Barbosa and Mendez-Monroy [6]. An example of combining time and space sharing is introduced by Wang et al. as *Simultaneous Multikernel*, which aims to maximize accelerator utilization by co-scheduling kernels with mutually compatible resource requirements [84].

Pemberton et al. [61] propose a low-code interface for GPU kernel tasks where developers compose applications of pre-existing, optimized GPU primitives, a similar level of abstraction as KaaS. Requiring the use of pre-built primitives instead of custom code or third-party libraries allows higher control over security and isolation with current GPU APIs, yet trades off the generalizability we target with KaaS.

These approaches are important in their own right, yet have a narrow focus on GPU performance and programmability. With KaaS, in contrast, we propose a programming interface and abstractions for applications that take advantage of heterogeneous hardware, with CPUs, GPUs, FPGAs, and others as equal execution targets for kernels and fine-grained, dynamic resource allocation to multiple applications.

8 Conclusion

With Kernel-as-a-Service, we have introduced a new serverless programming model for hardware accelerators by adopting concepts of the FaaS paradigm. Our experimental evaluation of KaaS prototypes on GPU, FPGA, TPU, and QPU demonstrates the viability and efficiency of this approach beyond device-specific frameworks. We have shown how finely-grained tasks in particular can derive performance, efficiency, and usability benefits in KaaS.

The future of computing systems will require managing disaggregated software on heterogeneous hardware, and KaaS is a first step towards an abstraction between these layers. Further research will require an even broader scope of evaluation of this paradigm with large-scale clusters and larger applications. We also plan to explore how FaaS and KaaS can be efficiently integrated to provide a seamless experience for developers. Finally, we intend to investigate possibilities of hardware/software co-design for security, isolation, and performance.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Google Research. Retrieved May 10, 2023 from https://www.tensorflow.org/

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2022. `tf.nn.conv2d` | *TensorFlow v2.11.0*. Google Research. Retrieved December 1, 2022 from https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

[3] Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. 2018. Managing Heterogeneous Resources in HPC Systems. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and Run-Time Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms* (Manchester, United Kingdom) *(PARAM-DITAM '18)*. Association for Computing Machinery (ACM), New York, NY, USA, 7–12. https://doi.org/10.1145/3183767.3183769

[4] AMD Xilinx. 2022. *Pynq: Python Productivity for Zynq*. Retrieved December 2, 2022 from http://pynq.io

[5] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '14)*. USENIX, Berkeley, CA, USA.

[6] Jose Antonio Ayala-Barbosa and Paul Erick Mendez-Monroy. 2022. A new preemptive task scheduling framework for heterogeneous embedded systems. In *Proceedings of the 2022 8th International Conference on Computer Technology Applications* (Vienna, Austria) *(ICCTA '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 77–84. https://doi.org/10.1145/3543712.3543756

[7] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery (ACM), New York, NY, USA, 89–103. https://doi.org/10.1145/3133850.3133855

[8] Kirk M. Bresniker, Paolo Faraboschi, Avi Mendelson, Dejan Milojicic, Timothy Roscoe, and Robert N.M. Watson. 2019. Rack-Scale Capabilities: Fine-Grained Protection for Large-Scale Memories. *Computer* 52, 2 (Feb. 2019), 52–62. https://doi.org/10.1109/MC.2018.2888769

[9] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José MM Montiel, and Juan D. Tardós. 2021. ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial, and Multimap SLAM. *IEEE Transactions on Robotics* 37, 6 (May 2021), 1874–1890. https://doi.org/10.1109/TRO.2021.3075644

[10] Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. 2019. Quantum Chemistry in the Age of Quantum Computing. *Chemical Reviews* 119, 19 (Aug. 2019), 10856–10915. https://doi.org/10.1021/acs.chemrev.8b00803

[11] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. 2018. Beyond SmartNICs: Towards a Fully Programmable Cloud. In *Proceedings of the 19th International Conference on High Performance Switching and Routing* (Bucharest, Romania) *(HPSR)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/HPSR.2018.8850757

[12] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. funcX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International symposium on high-performance parallel and distributed computing* (Virtual Event, USA) *(HPDC '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

[13] Marcin Copik, Marcin Chrapek, Alexandru Calotoiu, and Torsten Hoefler. 2022. *Software Resource Disaggregation for HPC with Serverless Computing*. Technical Report. Scalable Parallel Computing Lab, ETH Zürich, Zurich, Switzerland.

[14] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *Proceedings of the 37th IEEE International Parallel &Distributed Processing Symposium* (St. Petersburg, FL, USA) *(IPDPDS '23)*. IEEE, New York, NY, USA.

[15] Marco Cuturi and Mathieu Blondel. 2017. Soft-DTW: A Differentiable Loss Function for Time-Series. In *Proceedings of the 34th International Conference on Machine Learning* (Sydney, NSW, Australia) *(ICML '17)*. Journal of Machine Learning Research, 894–903.

[16] Bradley Denby and Brandon Lucia. 2020. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. 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 (ACM), New York, NY, USA, 939–954. https://doi.org/10.1145/3373376.3378473

[17] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. https://doi.org/10.1145/3419111.3421284

[18] Aditya Dhakal, Xukan Ran, Yunshu Wang, Jiasi Chen, and K. K. Ramakrishnan. 2022. SLAM-Share: Visual Simultaneous Localization and Mapping for Real-Time Multi-User Augmented Reality. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies* (Rome, Italy) *(CoNEXT '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 293–306. https://doi.org/10.1145/3555050.3569142

[19] Nicolas Dube, Duncan Roweth, Paolo Faraboschi, and Dejan Milojicic. 2021. Future of HPC: The Internet of Workflows. *IEEE Internet Computing* 25, 5 (Aug. 2021), 26–34. https://doi.org/10.1109/MIC.2021.3103236

[20] Jorge Ejarque, Rosa M. Badia, Loïc Albertin, Giovanni Aloisio, Enrico Baglione, Yolanda Becerra, Stefan Boschert, Julian R. Berlin, Alessandro D'Anca, Donatello Elia, et al. 2022. Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence. *Future generation computer systems* 134 (Sept. 2022), 414–429. https://doi.org/10.1016/j.future.2022.04.014

[21] Donatello Elia, Sandro Fiore, and Giovanni Aloisio. 2021. Towards HPC and Big Data Analytics Convergence: Design and Experimental Evaluation of a HPDA Framework for eScience at Scale. *IEEE Access* 9 (May 2021), 73307–73326. https://doi.org/10.1109/ACCESS.2021.3079139

[22] Hewlett Packard Enterprise. 2020. *Enabling GPU as a Service – A Cloud-Like Experience for GPU Infrastructure using Containers (Solution Brief)*. Retrieved September 11, 2023 from https://www.hpe.com/psnow/doc/a00075067enw

[23] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Grenoble, France) *(HWWS '04)*. Association for Computing Machinery (ACM), New York, NY, USA, 133–137. https://doi.org/10.1145/1058129.1058148

[24] Marcel Flottmann, Marc Eisoldt, Julian Gaal, Marc Rothmann, Marco Tassemeier, Thomas Wiemann, and Mario Porrmann. 2021. Energy-efficient FPGA-accelerated LiDAR-based SLAM for embedded robotics. In *Proceedings of the 2021 International Conference on Field-Programmable Technology* (Auckland, New Zealand) *(ICFPT '21)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/ICFPT52863.2021.9609934

[25] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, GA, USA) *(SC '20)*. IEEE, New York, NY, USA, 1–14. https://doi.org/10.1109/SC41405.2020.00021

[26] Rajesh Gandham, Yongpeng Zhang, Kenneth Esler, and Vincent Natoli. 2021. Improving GPU throughput of reservoir simulations using NVIDIA MPS and MIG. In *Proceedings of the Fifth EAGE Workshop on High Performance Computing for Upstream* (Online). European Association of Geoscientists & Engineers, Houten, The Netherlands, 1–5. https://doi.org/10.3997/2214-4609.2021612025

[27] Nicholas Gordon, Kevin Pedretti, and John R. Lange. 2022. Porting the Kitten Lightweight Kernel Operating System to RISC-V. In *Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers* (Dallas, TX, USA) *(ROSS 2022)*. IEEE, New York, NY, USA, 1–7. https://doi.org/10.1109/ROSS56639.2022.00008

[28] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T. Kandemir, and Chita R. Das. 2020. Fifer: Tackling underutilization in the serverless era. (Aug. 2020). arXiv:2008.12819

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV, USA) *(CVPR 2016)*. IEEE, New York, NY, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[30] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing* (Denver, CO, USA) *(HotCloud '16)*. USENIX Association, Berkeley, CA, USA.

[31] Anahita Hosseinkhani and Behnam Ghavami. 2021. Improving Soft Error Reliability of FPGA-based Deep Neural Networks with Reduced Approximate TMR. In *Proceedings of the 2021 11th International Conference on Computer Engineering and Knowledge* (Mashhad, Iran) *(ICCKE)*. IEEE, New York, NY, USA, 459–464. https://doi.org/10.1109/ICCKE54056.2021.9721442

[32] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-Mei Hwu. 2021. Pylog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (Oct. 2021), 2015–2028. https://doi.org/10.1109/TC.2021.3123465

[33] IBM Quantum. 2022. *Qiskit*. Retrieved December 2, 2022 from https://qiskit.org/

[34] Al Amjad Tawfiq Isstaif and Richard Mortier. 2023. Towards Latency-Aware Linux Scheduling for Serverless Workloads. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies* (Rome, Italy) *(SESAME '23)*. Association for Computing Machinery (ACM), New York, NY, USA, 19–26. https://doi.org/10.1145/3592533.3592807

[35] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the USENIX Annual Technical Conference* (Renton, WA, USA) *(ATC '19)*. USENIX Association, Berkeley, CA, USA, 947–960.

[36] Fauzi Mohd Johar, Farah Ayuni Azmin, Mohamad Kadim Suaidi, Abdul Samad Shibghatullah, Badrul Hisham Ahmad, Siti Nadzirah Salleh, Mohamad Zoinol Abidin Abd Aziz, and Mahfuzah Md Shukor. 2013. A review of genetic algorithms and parallel genetic algorithms on graphics processing unit (GPU). In *Proceedings of the 2013 International Conference on Control System, Computing and Engineering* (Penang, Malaysia) *(ICCSCE '13)*. IEEE, New York, NY, USA, 264–269. https://doi.org/10.1109/ICCSCE.2013.6719971

[37] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report. EECS Department, University of California, Berkeley, Berkeley, CA, USA. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[38] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (June 2020), 67–78. https://doi.org/10.1145/3360307

[39] Hamidreza Khaleghzadeh, Ziming Zhong, Ravi Reddy, and Alexey Lastovetsky. 2017. Out-of-core implementation for accelerator kernels on heterogeneous clouds. *The Journal of Supercomputing* 74, 2 (Sept. 2017), 551–568. https://doi.org/10.1007/s11227-017-2141-4

[40] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Online) *(OSDI '20)*. USENIX Association, Berkeley, CA, USA, 991–1010.

[41] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. 2020. Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 1576–1585. https://doi.org/10.1145/3341105.3373948

[42] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, TX, USA) *(LLVM '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–6. https://doi.org/10.1145/2833157.2833162

[43] Baolin Li, Tirthak Patel, Siddarth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. Using Multi-Instance GPU for Efficient Operation of Multi-Tenant GPU Clusters. (July 2022). arXiv:2207.11428

[44] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WoSC '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 37–42. https://doi.org/10.1145/3366623.3368139

[45] Teng Li, Vikram K. Narayana, Esam El-Araby, and Tarek El-Ghazawi. 2011. GPU Resource Sharing and Virtualization on High Performance Computing Systems. In *Proceedings of the 2011 International Conference on Parallel Processing* (Taipei, Taiwan) *(ICPP '11)*. IEEE, New York, NY, USA, 733–742. https://doi.org/10.1109/ICPP.2011.88

[46] Fabio Maschi, Dario Korolija, and Gustavo Alonso. 2023. Serverless FPGA: Work-In-Progress. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies* (Rome, Italy) *(SESAME '23)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–4. https://doi.org/10.1145/3592533.3592804

[47] Anil Mathew, Vasilios Andrikopoulos, and Frank J. Blaauw. 2021. Exploring the cost and performance benefits of AWS Step Functions using a data processing pipeline. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing* (Leicester, United Kingdom) *(UCC '21)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–10. https://doi.org/10.1145/3468737.3494084

[48] Ken McDonell. 2022. *Performance Co-Pilot.* Retrieved October 1, 2022 from https://pcp.io/

[49] Dejan Milojicic, Paolo Faraboschi, Nicolas Dube, and Duncan Roweth. 2021. Future of HPC: Diversifying Heterogeneity. In *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition* (Grenoble, France) *(DATE '21)*. IEEE, New York, NY, USA, 276–281. https://doi.org/10.23919/DATE51398.2021.9474063

[50] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel and Distrib. Comput.* 139 (May 2020), 32–42. https://doi.org/10.1016/j.jpdc.2020.01.004

[51] Anna Maria Nestorov, Josep Lluís Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. 2022. Floki: A Proactive Data Forwarding System for Direct Inter-Function Communication for Serverless Workflows. In *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds* (Quebec City, QC, Canada) *(WoC '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 13–18. https://doi.org/10.1145/3565384.3565890

[52] Sam Newman. 2015. *Building Microservices.* O'Reilly Media, Inc., Sebastopol, CA, USA.

[53] Kim Nguyen and Sam Chung. 2021. Low Maintenance, Low Cost, Highly Secure, and Highly Manageable Serverless Solutions for Software Reverse Engineering. In *Proceedings of the Conference on Information Systems Applied Research* (Washington, DC, USA) *(CONISAR '21)*. Information Systems and Computing Academic Professionals, 1–10.

[54] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy* (San Francisco, CA, USA) *(SP '20)*. IEEE, New York, NY, USA, 1003–1020. https://doi.org/10.1109/SP40000.2020.00055

[55] NVIDIA. 2023. *Multi-Process Service.* Retrieved May 25, 2023 from https://docs.nvidia.com/deploy/mps/index.html

[56] NVIDIA. 2023. *NVIDIA Multi-Instance GPU.* Retrieved May 25, 2023 from https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

[57] Jacob Pan. 2013. *RAPL (Running Average Power Limit) driver.* Retrieved December 2, 2022 from https://lwn.net/Articles/545745/

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. 8024–8035.

[59] Nathan Pemberton. 2022. *The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation.* Ph. D. Dissertation. University of California, Berkeley, Berkeley, CA, USA. Advisor(s) Randy Katz.

[60] Nathan Pemberton and Johann Schleier-Smith. 2019. The serverless data center: Hardware disaggregation meets serverless computing. In *Proceedings of the First Workshop on Resource Disaggregation* (Providence, RI, USA) *(WORD '19)*.

[61] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. 2022. Kernel-as-a-Service: A Serverless Interface to GPUs. (Dec. 2022). arXiv:2212.08146

[62] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1, Article 4213 (July 2014), 7 pages. https://doi.org/10.1038/ncomms5213

[63] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. 2019. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In *Proceedings of the International 2019 IEEE International Conference on Embedded Software and Systems* (Las Vegas, NV, USA) *(ICESS '19)*. IEEE, New York, NY, USA, 1–8. https://doi.org/10.1109/ICESS.2019.8782524

[64] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[65] IBM Quantum. 2021. *IBM Quantum Processor Types.* Retrieved May 24, 2023 from https://quantum-computing.ibm.com/services/resources/docs/resources/manage/systems/processors

[66] Issam Raïs, Anne-Cécile Orgerie, and Martin Quinson. 2016. Impact of Shutdown Techniques for Energy-Efficient Cloud Data Centers. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing* (Granada, Spain) *(ICA3PP '16)*. Springer, Heidelberg, Germany, 203–210. https://doi.org/10.1007/978-3-319-49583-5_15

[67] Sebastián Risco and Germán Moltó. 2021. GPU-Enabled Serverless Workflows for Efficient Multimedia Processing. *Journal of Applied Sciences* 11, 4 (Feb. 2021), 1438. https://doi.org/10.3390/app11041438

[68] Felix Ritter, Tobias Boskamp, A. Homeyer, Hendrik Laue, Michael Schwier, Florian Link, and H.-O. Peitgen. 2011. Medical Image Analysis. *IEEE Pulse* 2, 6 (Dec. 2011), 60–70. https://doi.org/10.1109/MPUL.2011.942929

[69] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, and Antonio Corradi. 2021. A Shared Memory Approach for Function Chaining in Serverless Platforms. In *Proceedings of the 2021 IEEE Symposium on Computers and Communications* (Athens, Greece) *(ISCC '21)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/ISCC53001.2021.9631385

[70] Marc Sánchez-Artigas and Germán T. Eizaguirre. 2022. A Seer Knows Best: Optimized Object Storage Shuffling for Serverless Analytics. In *Proceedings of the 23rd conference on 23rd ACM/IFIP International Middleware Conference* (Quebec City, QC, Canada) *(Middleware '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 148–160. https://doi.org/10.1145/3528535.3565241

[71] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2022. Fusionize: Improving Serverless Application Performance through Feedback-Driven Function Fusion. In *Proceedings of the 10th IEEE International Conference on Cloud Engineering* (Asilomar, CA, USA) *(IC2E 2022)*. IEEE, New York, NY, USA, 85–95. https://doi.org/10.1109/IC2E55432.2022.00017

[72] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *Comput. Surveys* 54, 11 (Jan. 2022), 1–32. https://doi.org/10.1145/3510611

[73] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference* (Virtual Event, USA) *(ATC '20)*. USENIX Association, Berkeley, CA, USA, 205–218.

[74] John Shalf. 2020. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A* 378, 2166 (Jan. 2020), 20190061. https://doi.org/10.1098/rsta.2019.0061

[75] Prateek Sharma. 2022. Challenges and Opportunities in Sustainable Serverless Computing. In *Proceedings of the 1st Workshop on Sustainable Computer Systems Design and Implementation* (La Jolla, CA, USA) *(HotCarbon '22)*. USENIX Association, Berkeley, CA, USA.

[76] Sushant Sharma, Chung-Hsing Hsu, and Wu-chun Feng. 2006. Making a Case for a Green500 List. In *Proceedings of the Proceedings 20th IEEE International Parallel & Distributed Processing Symposium* (Rhodes, Greece) *(IPDPS '06)*. IEEE, New York, NY, USA. https://doi.org/10.1109/IPDPS.2006.1639600

[77] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D. Sinclair, and Shivaram Venkataraman. 2022. Not All GPUs Are Created Equal: Characterizing Variability in Large-Scale, Accelerator-Rich Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, TX, USA) *(SC '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–15.

[78] Sebastian Thrun. 2007. Simultaneous localization and mapping. In *Robotics and cognitive approaches to spatial mapping*. Springer, 13–41.

[79] Paramita Basak Upama, Md Jobair Hossain Faruk, Mohammad Nazim, Mohammad Masum, Hossain Shahriar, Gias Uddin, Shabir Barzanjeh, Sheikh Iqbal Ahamed, and Akond Rahman. 2022. Evolution of Quantum Computing: A Systematic Survey on the Use of Quantum Computing Tools. In *Proceedings of the 2022 IEEE 46th Annual Computers, Software, and Applications Conference* (Virtual Event, USA) *(COMPSAC '22)*. IEEE, New York, NY, USA, 520–529. https://doi.org/10.1109/COMPSAC54236.2022.00096

[80] Ava Vali, Sara Comai, and Matteo Matteucci. 2020. Deep Learning for Land Use and Land Cover Classification Based on Hyperspectral and Multispectral Earth Observation Data: A Review. *Remote Sensing* 12, 15 (Aug. 2020), 2495. https://doi.org/10.3390/rs12152495

[81] Blesson Varghese and Rajkumar Buyya. 2018. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems* 79 (Feb. 2018), 849–861. https://doi.org/10.1016/j.future.2017.09.020

[82] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *Proceedings of the 2021 USENIX Annual Technical Conference* (Virtual Event, USA) *(ATC '21)*. USENIX Association, Berkeley, CA, USA, 443–457.

[83] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. (Sept. 2019). arXiv:1909.01315

[84] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2015. Simultaneous Multikernel: Fine-Grained Sharing of GPUs. *IEEE Computer Architecture Letters* 15, 2 (Sept. 2015), 113–116. https://doi.org/10.1109/LCA.2015.2477405

[85] Logan Ward, Ganesh Sivaraman, J. Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C. Redfern, Rajeev S. Assary, Kyle Chard, Larry A. Curtiss, Rajeev Thakur, and Ian Foster. 2021. Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing. In *Proceedings of the 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments* (St. Louis, MO, USA) *(MLHPC '21)*. IEEE, New York, NY, USA, 9–20. https://doi.org/10.1109/MLHPC54614.2021.00007

[86] Stefan Weinzierl. 2000. Introduction to Monte Carlo methods. (June 2000). arXiv:hep-ph/0006269

[87] Sebastian Werner and Trever Schirmer. 2022. Hardless: A Generalized Serverless Compute Architecture for Hardware Processing Accelerators. In *Proceedings of the 10th IEEE International Conference on Cloud Engineering* (Asilomar, CA, USA) *(IC2E 2022)*. IEEE, New York, NY, USA, 79–84. https://doi.org/10.1109/IC2E55432.2022.00016

[88] Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit tool chain: Working with and developing for real quantum computers. In *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition* (Florence, Italy) *(DATE '19)*. IEEE, New York, NY, USA, 1234–1240. https://doi.org/10.23919/DATE.2019.8715261

[89] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. *ACM SIGPLAN Notices* 52, 4 (April 2017), 483–496. https://doi.org/10.1145/3093336.3037742

[90] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. *ACM SIGPLAN Notices* 52, 8 (Aug. 2017), 221–234. https://doi.org/10.1145/3155284.3018754

[91] Mohamed Zahran. 2016. Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives. *Queue* 14, 6 (Nov. 2016), 31–42. https://doi.org/10.1145/3028687.3038873

[92] 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 (ACM), New York, NY, USA, 845–858. https://doi.org/10.1145/3373376.3378491

[93] Peng Zhang, Jianbin Fang, Canqun Yang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Optimizing Streaming Parallelism on Heterogeneous Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (March 2020), 1878–1896. https://doi.org/10.1109/TPDS.2020.2978045

[94] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. 2021. Toward QoS-Awareness and Improved Utilization of Spatial Multitasking GPUs. *IEEE Trans. Comput.* 71, 4 (March 2021), 866–879. https://doi.org/10.1109/TC.2021.3064352

[95] Chen Zhao, Wu Gao, Feiping Nie, and Huiyang Zhou. 2021. A Survey of GPU Multitasking Methods Supported by Hardware Architecture. *Transactions on Parallel and Distributed Systems* 33, 6 (Sept. 2021), 1451–1463. https://doi.org/10.1109/TPDS.2021.3115630

[96] Haidong Zhao, Zakaria Benomar, Tobias Pfandzelter, and Nikolaos Georgantas. 2022. Supporting Multi-Cloud in Serverless Computing. In *Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (Vancouver, WA, USA) *(UCC '22)*. IEEE, New York, NY, USA, 285–290. https://doi.org/10.1109/UCC56403.2022.00051

[97] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, MO, USA) *(SC '21)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–15. https://doi.org/10.1145/3458817.3476215