Kernel-as-a-Service: A Serverless Interface to GPUs

Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez

UC Berkeley

Abstract

Serverless computing has made it easier than ever to deploy applications over scalable cloud resources, all the while driving higher utilization for cloud providers. While this technique has worked well for easily divisible resources like CPU and local DRAM, it has struggled to incorporate more expensive and monolithic resources like GPUs or other application accelerators. We cannot simply slap a GPU on a FaaS platform and expect to keep all the benefits serverless promises. We need a more tailored approach if we want to best utilize these critical resources.

In this paper we present Kernel-as-a-Service (KaaS), a serverless interface to GPUs. In KaaS, GPUs are first-class citizens that are invoked just like any other serverless function. Rather than mixing host and GPU code as is typically done, KaaS runs graphs of GPU-only code while host code is run on traditional functions. The KaaS system is responsible for managing GPU memory and schedules user kernels across the entire pool of available GPUs rather than relying on static allocations. This approach allows us to more effectively share expensive GPU resources, especially in multitenant environments like the cloud. We add support for KaaS to the Ray distributed computing framework and evaluate it with workloads including a TVM-based deep learning compiler and a BLAS library. Our results show that KaaS is able to drive up to 50x higher throughput and 16x lower latency when GPU resources are contended.

1 Introduction

Resource utilization is a critical requirement in cloud computing. For users, this means minimizing costs by allocating only what they need while scaling up and down quickly. For cloud providers, this means being able to quickly reclaim idle resources that can then be rented to other customers. Serverless computing helps achieve these goals by focusing on time-bounded invocations of *functionality* rather than explicit *allocation* of resources. In other words, users ask for some function or service to be invoked without considering how or where it will run. In the case of function-as-a-service (FaaS), this functionality takes the form of small stateless functions written in general-purpose languages like Python or Javascript.

Serverless computing has already seen significant adoption. Users get true pay-per-use and providers can quickly reallocate resources to other jobs. However, today's Function-as-a-Service (FaaS) systems remain narrow in scope by focusing on CPU-based workloads running general purpose code on familiar OS environments. To date, none of the major cloud providers offer a GPU-enabled FaaS service. This is somewhat surprising given the growing popularity of application accelerators. Why is it challenging to deploy accelerators like GPUs in a FaaS system? The problem lies in the techniques that make FaaS practical to implement. FaaS functions are small and limited in scope, they consume few resources when not executing, and use easily shared and subdivided resources. Providers are free to kill FaaS containers as needed to free resources, or aggressively cache them to reduce cold starts. Furthermore, providers can mitigate the performance costs of explicit state by maintaining shared caches of their data layer. GPUs upend these assumptions. Unlike CPUs, GPUs are expensive, difficult to share on a fine granularity, and have their own memory that must be managed by the user.

This is not to say that GPUs are fundamentally incompatible with the FaaS model; they simply require different techniques. Unlike general-purpose code, GPU functions (called *kernels*) have limited capabilities and few dependencies which greatly simplifies function startup. GPU functions also have more predictable inputs and outputs, enabling greater optimizations from the FaaS scheduler and data layer.

In this paper, we present a truly serverless interface to GPUs, called Kernel-as-a-Service (KaaS), that is able to take advantage of these properties to enable high-utilization of GPUs in the cloud with minimal performance overheads. In KaaS, GPUs become first-class citizens that are directly invoked through a GPU-specific function type (see Figure 2a). Rather than explicitly mixing host and device code, users register CUDA kernels that run independently with no user-provided host code. Since the KaaS system is in full control of the GPUs, it is able to explicitly manage device memory and multiplex kernels from many users at a fine granularity.

For compute-heavy tasks, KaaS experiences virtually no decrease in aggregate performance, even when there are far more users than available GPUs. For memory-heavy workloads, KaaS performance degrades gracefully when GPU memory requirements exceed capacity. In our evaluation, we compare KaaS to a traditional FaaS approach that allocates GPUs exclusively to functions during execution. Our results demonstrate 50x higher throughput and 16x lower latency for a compute-intensive linear system solver when the number of clients exceeds available GPUs by 4x. Even when total available GPU memory is exceeded by 1.5x, KaaS supports 53x greater throughput and 12.6x lower latency for a memory-intensive deep-learning inference benchmark.

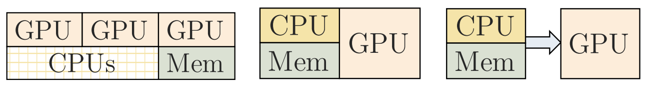
We present the following key contributions:

- A new GPU-native serverless function type, kernel task (kTask), that treats GPUs as first-class citizens.
- Three programming interfaces to KaaS functions: a low-level application programming interface (API), a TVM-based compiler, and a suite of built-in libraries.
- Two scheduling algorithms that take advantage of the unique properties of FaaS and KaaS functions.
- A Ray-based prototype of KaaS with drastically improved utilization of GPU resources in a multitenant environment.

In §2, we discuss how GPUs are deployed today. §3 presents the KaaS programming model while §4 describes our Ray-based prototype. In §5, we evaluate this prototype against a traditional mixed host/GPU approach with a diverse set of applications ranging from deep neural-network model serving to general purpose linear algebra. We compare our work to other approaches to multitenant GPU utilization in §6. We conclude in §7 with a discussion of how the KaaS model can be extended to new resource types and enable new technologies.

2 Cloud GPUs Today

Today, GPUs are exposed to users through a number of allocation schemes. We present the most relevant approaches here (see Figure 1).



(a) Serverful (b) GPU+FaaS (c) KaaS

Figure 1: Three possible GPU deployment strategies. In a serverful deployment, users get allocated a large collection of CPUs and GPUs for an unbounded period of time. In GPU+FaaS, users run a short stateless task that uses a small number of resources (e.g. a single GPU) only for the duration of the task. In KaaS, users submit CPU and GPU tasks separately where each task type runs in a resource-specific environment.

2.1 Serverful

Traditionally, GPUs have been deployed in a serverful manner as PCI-E attached cards in a virtual machine (figure 1a). Users are fully responsible for driving enough utilization to justify the added cost. This is challenging. Microsoft reports only 52% average utilization of their deep learning GPUs [22]. Importantly, the tight coupling of host and GPU resource allocations leads to wasted host resources as well. From the same trace, Microsoft reports only about 30% utilization of CPUs for GPU-enabled deep learning jobs while memory utilization is well over 90%. While the trace does not report total memory capacity, they report that 6.2% of node failures are due to exceeding host memory capacity. It is challenging to predict memory and CPU requirements, necessitating wasteful overprovisioning to avoid failures.

2.2 Service API

Rather than exposing accelerators directly to users, some cloud systems expose *services* that then utilize the accelerators. Of particular note is Microsoft's Catapult system that deploys FPGAs as network endpoints that can be programmed into "FPGA Microservices" [34]. For example, the Microsoft Brainwave project places deep neural network models directly on a network of FPGAs that can be addressed from conventional software [12]. However, Catapult is neither general purpose nor multi-tenant, limiting its deployment to internal tools or high-level customer-facing services [17]. Other services focus on a narrow domain such as deep learning [20, 40, 13]. These services run on dedicated clusters with specialized schedulers and data management policies. They are not able to take advantage of the broader serverless infrastructure for scheduling, data movement, and end-to-end performance guarantees. This lack of generality and integration can lead to many of the challenges experienced by serverful approaches in a multitenant environment where workloads have a diversity of resource requirements.

2.3 GPU Virtualization

Traditional serverful approaches one or more whole GPUs to clients persistently. This can be wasteful when applications cannot effectively use an entire GPU. The tight coupling of host and GPU resources can also lead to resource stranding where GPUs cannot be allocated to users due to insufficient host resources. One technique to avoid these issues is *GPU Virtualization* where clients are allocated fractions of a GPU. On a single node, NVidia provides multi-process server (MPS) and multi-instance GPU (MIG) for this purpose [31, 29]. API forwarding techniques like rCUDA or VMWare bitfusion additionally avoid stranding by allowing GPUs and hosts to be allocated from different physical servers [18, 44, 23]. While these techniques make resource allocation more flexible, they do not address the fundamental limitations of serverful approaches. GPUs are allocated persistently on a per-client basis regardless of client load. Host and GPU resource allocations also remain coupled. Indeed, GPU virtualization multiplies the number of host resources needed by increasing the number of supported GPU clients. These approaches also do not address broader distributed system concerns like scheduler integration or data layer management.

2.4 FaaS+GPU

Some systems have offered a more serverless FaaS+GPU approach. In these systems, users upload a CPU-oriented function and the provider ensures that the function runs in a container that includes a GPU (Figure 1b) [28, 30, 32, 25]. While these approaches present a familiar interface to accelerators, they give away many of the features that make serverless appealing.

As an example, the Ray distributed computing framework supports GPU-enabled remote functions called *tasks*, but requires that applications carefully design their tasks to manually share resources and clean up properly when they finish. In practice, this is difficult to achieve. Indeed, users are advised in the documentation to force Ray to restart workers on each task invocation to ensure resources are properly freed [36]. Rather than using serverless functions, users are often advised to fall back to non-serverless stateful actors to manually manage GPU resources.

Another challenge that arises with this approach comes from cold start mitigation strategies. CPU functions incur a significant startup latency due to container and language runtime initialization. A Python script that simply imports tensorflow and immediately exits takes 1.9 s when the OS buffer cache is warm. A true cold start that must read from disk takes even longer at 6.8 s[1]. To minimize the impact on users, cloud providers often keep function executors allocated in anticipation of a new request [42]. This policy is reasonable because SMT threads are cheap and processes are easily idled. The same policy applied to a GPU would cost at least 60x more since the provider would need to keep the much more expensive GPU idle[2]. While the increased cost of a GPU is justified when fully utilized, both resources provide the same utility when idle: zero.

3 A New Approach: Kernel-as-a-Service

Fundamentally, existing approaches struggle because they cede control of GPUs to applications by tightly coupling host and GPU code. We take a different approach, called Kernel-as-a-Service (KaaS), that explicitly decouples host functions from GPU kernels at the system level (Figure 1c). Following the principles of serverless computing, our system takes user descriptions of GPU *functionality* rather than providing GPU *allocations*. This puts the responsibility for GPU memory management and scheduling on the system rather than relying on users to make optimal use of these expensive resources.

This strategy elevates GPU functionality to a first-class entity, along with traditional FaaS functions and other serverless services (see Figure 2a). We refer to CPU-specialized functions as CPU tasks (cTasks) and GPU-specialized functions as kernel tasks (kTasks). Rather than providing a Python source file or Linux container, kTasks take the form of a graph of CUDA kernels to be executed. Graph inputs and outputs are provided as objects in the system data layer (e.g., keys in a key-value store). Other buffers like intermediate outputs and temporary buffers are described simply by their size. The system is then responsible for ensuring that all required buffers are available in GPU memory before beginning execution of the kernel graph. kTasks are not permitted to dynamically allocate memory or access the data layer, leading to highly predictable resource requirements. Furthermore, kTasks do not include any host code whatsoever, which simplifies the software environment on KaaS executors and prevents external side effects.

[1]Run on an Amazon EC2 p3.2xlarge instance with a general-purpose SSD (gp2).

[2]Based on the capital costs of the 56 SMT core Xeon Platinum 8380 ($10k MSRP[1]) and an Nvidia A100 ($12k[2]).

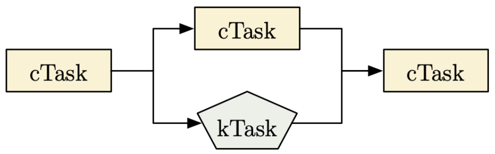
3.1 Why KaaS?

KaaS provides a number of important benefits over more traditional approaches to GPU multitenancy. Figure 3 summarizes these properties. The KaaS approach only explicitly uses GPUs in its programming, and presumably billing, model. This *on-demand* allocation means that users are charged only for the time their code actually ran on the GPU rather than paying for long lived allocations. Users also do not need to allocate an entire server or VM just to use its GPUs. In serverful approaches like GPU virtualization, the user must pay for idle CPU, memory, and disk resources while the GPU is running. They must also explicitly manage resource deployment and scaling. While service APIs free users from resource allocation concerns, they exist in application-specific environments that do not share the data and scheduling semantics of general purpose serverless frameworks. KaaS supports arbitrary GPU applications in a way that integrates natively with general-purpose serverless frameworks while allocating GPU resources to users only when needed.

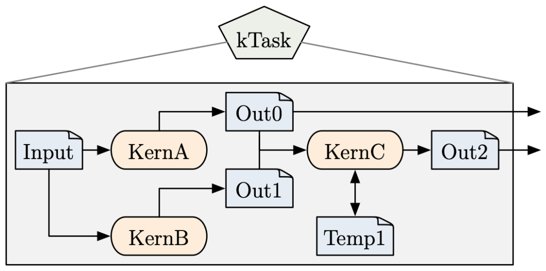
3.2 KaaS Interface

kTasks appear to the rest of the system like any other function. They provide named functionality without any explicit resource allocation, and interact with a common data layer for inputs and outputs. While the KaaS approach is not tied to any specific serverless framework, it does assume a graph of functions communicating through a common data layer. Figure 2a depicts a typical application in such a system. Each node in this graph can be implemented in many different ways, and node implementations can be modified or replaced without affecting the rest of the application. While this project focuses on GPU-typed functions, the KaaS approach can be applied to other function types such as deep learning-specific accelerators and specialized systems on chip (SoCs). We explore this in more detail in §7.

While the external interface is assumed to be common to all function types, their internal implementation can be specialized. Figure 2b depicts the user's view of a kTask. Users register a library of precompiled CUDA kernels that can then be combined into an execution graph when invoked. These kernels can come from common highly optimized libraries like NVidia's Cutlass linear algebra library [16], code compiled by frameworks like TVM [11],



(a) Serverless graph including both GPU and CPU-specialized tasks. While different task types have different implementations, at the application graph level they all behave similarly; they take inputs, perform computation, and produce outputs.



(b) kTasks are defined as a dataflow graph of individual CUDA kernels with predefined inputs, outputs, and temporary buffers. Inputs and outputs go through the data layer of the broader serverless infrastructure.

Figure 2: KaaS introduces a new GPU-specialized function type called a kTask.

	General Purpose	**On-Demand Allocation**	**Serverless Integration**
KaaS	✓	✓	✓
GPU Virtualization	✓	✗	✗
Service APIs	✗	✓	✗

Figure 3: Features of GPU multitenancy approaches. *General purpose* systems can run arbitrary GPU code rather than focusing on a specialized domain. Systems that *allocate GPUs on-demand* support pay-per-use, transparent autoscaling, and do not couple GPU allocation lifetimes with other resources. Finally, multi-tenant GPU systems can *integrate natively* into a broader general-purpose serverless environment.

or custom application-specific kernels. Each kernel is provided a set of GPU memory addresses representing inputs, temporaries, and outputs. These data objects are identified using the global data layer's semantics. Users may optionally specify a fixed number of iterations for a particular request. Future work will allow for more dynamic control flow similar to TVM's Relay IR or Dandelion's EDGE graph representation [39, 41].

3.2.1 Nearest-Neighbors Example

We now describe an example kTask: N nearest neighbor. In this algorithm, we iteratively multiply an adjacency matrix A by a vector X representing starting vertices. On each iteration, the set of visited vertices in this iteration is accumulated into a vector V representing the set of nearest neighbors. Algorithm 1 describes this more formally.

Algorithm 1 An iterative N nearest-neighbor search over an adjacency matrix A. X_1 contains an initial set of vertices while V_{N+1} contains the final set of nearest neighbors.

```
1: for $i \leftarrow 1, N$ do
2:     $X_{i+1} \leftarrow A \cdot (X_i - V_i)$
3:     $V_{i+1} \leftarrow V_i + X_i$
4: end for
```

Figure 4 shows how this algorithm could be implemented in KaaS. The A matrix along with an initial set of starting vertices X_{inp} are passed in as objects in the data layer. Since A is large and constant, it can be cached by the KaaS executor in GPU memory while the smaller X_{inp} may change on each invocation. Some buffers like X_{tmp} and X_{iter} represent intermediate values and can simply be allocated by the KaaS executor. Finally, V_{iter} represents an output of the system and will be written back to the key specified in the KaaS request. The kernels *vsub*, *vadd*, and *matmul* can be implemented in a number of ways, including user-provided code or a built in and optimized library like graphBLAS [8]. Users would likely call this function as part of a larger application with potentially long gaps between invocations. Unlike traditional approaches, users would not need to explicitly provision GPUs in their system and would only pay for the time the GPUs were actually running. In between user requests, the KaaS system would be free to execute functions from other users.

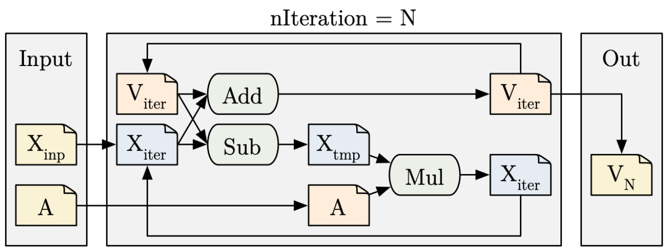


Figure 4: KaaS graph of a nearest-neighbors algorithm (equation 1). X_{inp} and A are read from the system data layer, while V_{iter} will be written back as an output. X_{tmp} and X_{iter} are ephemeral buffers that will never be copied off the GPU.

3.2.2 Challenges

These benefits do not come for free. The CUDA ecosystem has evolved around a model of locally-attached GPUs. Even multi-GPU workloads require host involvement to manage data transfers and scheduling. These assumptions are often baked into highly-tuned, but opaque, libraries. The KaaS model upends many of these assumptions. Additionally, placing GPUs behind a distributed abstraction will lead to additional invocation latency. Applications that finely interleave host and device code may not be able to amortize this additional latency.

The rise of serverless computing and modern machine learning frameworks suggests a promising path out of these challenges. Users have shown a willingness to decouple applications further in cloud applications, and backwards compatibility often takes a backseat to agility. The widely-adopted microservice architecture provides a good example of this [7]. These systems are a radical departure from traditional monolithic designs but enable rapid development, flexible deployment, and organizational flexibility. Another important trend is the rise of GPU-enabled application frameworks [11, 3, 33, 35]. Many GPU applications, and deep learning in particular, are no longer interacting with the GPU directly. Instead, these applications are increasingly relying on frameworks and libraries to generate GPU code transparently. In many cases, these frameworks already support, and optimize for, remote execution. KaaS can leverage these trends to make radical changes to accelerator interfaces with minimal impact on users.

4 Implementation

We now present an implementation of KaaS using the Ray distributed computing framework [28]. In addition to the core system, we provide a number of methods for users to implement kTasks, including a TVM-based deep learning compiler, a pre-made BLAS library based on Nvidia Cutlass, and hand-written applications.

Throughout this section, we assume a scenario in which there are multiple *clients* submitting requests for a particular function. We use the term *function* to refer to a particular logical function, independent of any particular instantiation of that function. For simplicity, we assume clients send requests to only one function and that different clients use different functions.

4.1 Ray Implementation

4.1.1 Ray Background

Ray provides users with a Python API to run stateless functions called *tasks*, or stateful *actors* across a cluster. Data are communicated using references to objects in an immutable object store called *Plasma*. References are a form of *future* [5] and can be created before their associated object is available. Ray takes advantage of this to create lazily-executed graphs of tasks/actors that are scheduled only when their input references are available. Ray maintains a number of processes on each node called *workers* that execute tasks and actors. A single worker may execute many tasks, but actors are always run on a dedicated worker. To control resource usage, users may annotate tasks or actors with resource requirements such as the number of GPUs required. Ray then ensures they are run on a worker that has been allocated those resources.

4.1.2 GPU-Enabled Functions in Ray

Ray does not enforce isolation of resources for tasks. Instead, it relies on applications to ensure they do not exceed their limits. Furthermore, tasks running on the same worker share resources at a fine granularity and must ensure that all resources are freed before exiting. This is difficult to ensure in practice, so the Ray documentation recommends forcing worker restarts on every invocation of GPU-enabled tasks [36].

Actors are persistent and run on dedicated workers. They are less sensitive to resource management concerns and can cache GPU state between invocations. However, there cannot be more GPU-enabled actors than available GPUs in the system and users must manually manage

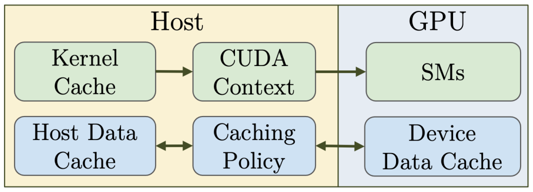


Figure 5: Design of the KaaS executor. CUDA kernel code and a data cache are maintained on the host. The GPU maintains its own cache, independent of the host, and runs user kernels to completion in a CUDA stream.

their actors to ensure that they do not exhaust available resources.

As a baseline, we enhance Ray with a new safe GPU-enabled task type called Exclusive Task (eTask). eTasks are written in Python in the same way as regular Ray actors and tasks. Unlike Ray native tasks, eTasks run on a dedicated worker per task with exclusive control of a GPU. They can opportunistically cache state between invocations. However, because eTasks have exclusive control of their GPU, the system may need to terminate them to free resources for new eTasks. eTasks provide an interface similar to many FaaS platforms like AWS Lambda or Google Cloud Functions.

4.1.3 KaaS Implementation in Ray

To implement KaaS in Ray, we provide an additional GPU-typed function called a *kTask*. kTasks behave similarly to Ray's CPU-based tasks; they take in object references, output new references, and execute based on input availability. However, rather than providing Python source, kTasks are defined by a request object consisting of an execution graph of kernels to invoke and buffer specifications describing input, temporary, and output buffers as well as literals for simple pass-by-value inputs.

kTasks are run by an alternative worker implementation called the *KaaS executor* (Figure 5). Much like regular workers, the KaaS executor is responsible for managing kTask code and caching objects from the object store. A single executor can handle any kTask without needing to restart. kTasks are routed to KaaS executors based on a centralized scheduler that augments Ray's internal scheduler to consider input availability, data locality, and per-GPU load.

Internally, the KaaS executor manages code through a kernel cache that handles linking CUDA libraries and preparing kernels for invocation. Upon invocation, the linked kernel is launched using regular CUDA APIs on a single stream. Currently, kernels are invoked serially, though future implementations could support concurrent invocation of non-dependent kernels.

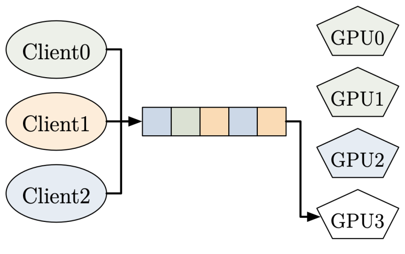
Data are managed through tiered host and GPU memory caches that extend Ray's built-in data layer. Objects are first loaded from Ray's object store into a data cache in host memory before being loaded into GPU memory. Ephemeral intermediate buffers are also cached in GPU memory to avoid frequent calls to CUDA's expensive memory allocator. The current design is a hybrid inclusive/exclusive cache where inputs are kept in both host and GPU caches, but outputs and intermediates exist only in the GPU cache. When GPU memory capacity is exceeded, the GPU cache first evicts from the set of objects with only one use before considering more frequently used objects. Both sets use a least-recently-used policy.

4.1.4 GPU Worker Pool

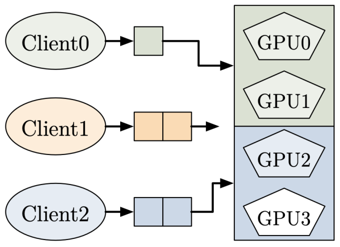
Both kTasks and eTasks require custom workers to run on each GPU. For eTasks, these workers are user-defined actors. For kTasks, this is the KaaS executor. We implemented a custom worker pool mechanism in Ray to manage the scheduling of these workers. The worker pool includes two scheduling policies:

- **CFS-Affinity:** The CFS-Affinity policy is based on the completely fair scheduler from Linux [10] and ensures that clients get a fair share of GPU runtime. It allows client requests to run on any worker, but preferentially routes requests from the same client to the same GPU to maximize data locality.
- **Exclusive:** The exclusive policy ensures that no two functions run on the same worker, while repeated invocations of the same function are always routed to the same set of workers. To maintain this property, the exclusive policy must kill existing workers to make room for new requests if the number of unique functions exceeds the number of GPUs.

The CFS-Affinity policy, depicted in Figure 6a launches a worker on each GPU at boot time and never restarts them. Client requests share the same workers (e.g. the KaaS executor). It maintains a running count of each client's accumulated GPU time weighted by GPU affinity. For non affinitized GPUs, the client's runtime is penalized by 10x their average request latency. When a GPU becomes idle, the scheduler searches the clients for the one with the smallest weighted runtime to run. This algorithm ensures that all clients receive similar GPU time while remaining work-conserving. By attempting to route requests to the same GPU when possible, it maximizes



(a) **CFS-Affinity Policy** Requests from any client can go to any GPU. Requests are processed according to the CFS algorithm, but priorities are weighted by a GPU affinity. In this scenario, the request from Client1 will be routed to GPU3.



(b) **Exclusive Policy** There must be a client-specific worker on each GPU. Clients 0 and 2 have pools of GPUs assigned to them while Client 1 currently has no assigned GPUs. In this scenario, the exclusive policy would need to re-balance the pools, taking GPU3 from Client 2 and cold-starting a worker for Client 1 on it.

Figure 6: **KaaS Schedulers:** In this scenario, clients are submitting a series of requests for a client-specific function. there are three independent clients and four GPUs. GPUs $0-2$ are currently running requests. GPU3 was previously running a request from Client 2 but is now idle.

data locality in GPU memory. KaaS can safely use the CFS-Affinity policy because the permanent worker is the KaaS executor which can service requests from multiple kTasks without needing to restart. eTasks require strict isolation between workers and cannot use this policy.

The Exclusive policy in Figure 6b is more complex. It must ensure that requests always run on a dedicated per-client worker rather than sharing a GPU execution environment. To do this, the Exclusive policy maintains independent per-client *pools* of workers. Internally, these pools follow the Balance policy, but they only service requests from one client. If a client submits a request that cannot be serviced from its pool immediately, the policy may consider shrinking an existing pool to free GPU resources for the new request. It begins by finding the largest pool to use as an eviction candidate p_{victim}. If there are multiple pools with the largest size, the least-recently evicted pool is chosen. If the requesting client's pool, p_{req}, is smaller than p_{victim}, the algorithm will evict a GPU from p_{victim} and assign it to a new worker in p_{req}. If p_{victim} has idle GPUs, they are simply re-assigned to p_{req}. Otherwise, the algorithm selects a busy GPU and waits for the currently executing request to finish before re-assigning the GPU to p_{req}. If p_{req} is in the set of largest pools, the algorithm simply blocks the request until a worker from p_{req} becomes available. We use the Exclusive policy for all eTask experiments.

4.2 Application Support

There are several ways to specify kTasks:

4.2.1 Low-Level API

KaaS exposes a python API for describing kTask requests. Figure 7 shows the datastructures used to describe kTasks. These requests consist of a list of kernels to invoke and their associated inputs, outputs, and temporary buffers. Kernels may also take literal arguments that are passed by value in the KaaS request. The kernels themselves are described using a filesystem path to the compiled CUDA code and a kernel name within that file along with the appropriate CUDA grid and block dimensions to use when launching. In addition to kernels, we provide a simple control-flow mechanism for fixed-length iteration. These requests are then serialized by Ray and sent to the KaaS executor for processing. Callers immediately receive a reference to the future response, just like any other Ray task.

kTask inputs are passed as Ray object references, while output references are provided in the kTask return value.

kaasReq	
Kernels	list of kernelSpecs to run
nIters	Number of times to run the request

bufferSpec	
Key	Object store key
Size	Buffer size in bytes
Ephemeral	Expose buffer to object store?

literalSpec	
Type	Data type
Value	Value

kernelSpec	
Library	Path to pre-compiled CUDA kernels
Kernel	Kernel name within Library
Grid & Block Dims	Grid & block dimensions
smemSize	Shared memory size.
Literals	List of the literalSpec arguments
Arguments	List of bufferSpec arguments. Can be marked as input, output, or temporary

Figure 7: Low-level KaaS API. kTasks are described by a `kaasReq` data structure that contains a list of CUDA kernels to run and their arguments. Arguments can either be objects in the object store or passed by value as literals.

Internal buffers are only valid for the duration of the request and are not associated with the Ray object store. Upon completion of the request, any output buffers are written back to the Ray object store and a reference to this output is returned to the caller.

4.2.2 TVM-Based Deep Learning Compiler

Rather than manually writing kTasks, users can generate kTasks using high-level frameworks and compilers. We modified the TVM deep-learning compiler to include KaaS as a backend target [11]. While we chose TVM for our prototype, KaaS can behave as a backend to any deep learning framework that generates static graphs of CUDA kernels. At a high level, TVM functions by generating a DAG of operators to execute the model. Each of these operators consist of one or more CUDA kernels. This scheme has a 1-1 correspondence with KaaS graphs, with the only difference being that each operation needs to be expanded into one or more kernel nodes.

Our approach to converting the TVM runtime graph to a kTask is straightforward. First, we use TVM to generate the CUDA kernel library and the static runtime graph. We then extract the necessary information, such as the grid and block dimensions for each kernel. This information is then used to generate the KaaS code for the request.

4.2.3 BLAS Library

The CUDA ecosystem has benefited greatly from high quality libraries of common kernels such as cuBLAS and cuDNN [14, 15]. These libraries are often authored by service providers or device manufacturers. Likewise, we expect a similar ecosystem would grow around a realistic KaaS deployment. For this project, we ported one such library based on Nvidia's Cutlass BLAS library [16]. Cutlass is a C++ template library that allows users to instantiate handles to specialized linear algebra kernels. We provide a KaaS interface to Cutlass through built in functions. Rather than uploading specific kernels, users can simply reference one of the system-provided functions. KaaS then manages interfacing with the library and managing device resources as needed.

While porting Cutlass to KaaS was non-trivial, it was a one-time cost borne by the system provider rather than individual users.

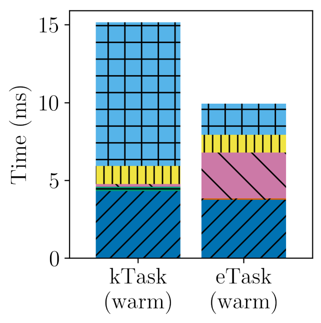
5 Evaluation

5.1 Experimental Setup

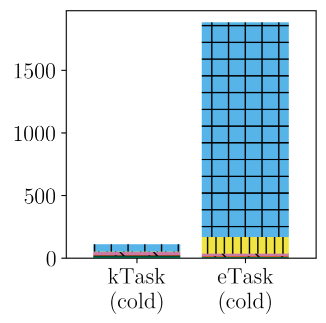
Our experiments were performed on a single p3.8xlarge Amazon EC2 instance with 32 vCPUs and 4 Nvidia V100 GPUs (with 16 GB memory each).

5.2 Micro-Benchmarks

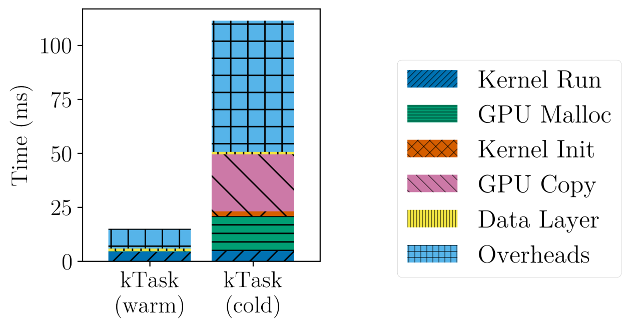
For this analysis, we use a simple chained matrix multiplication micro-benchmark. Inputs are taken from the data layer and fed through a series of multiplications on the GPU before writing the final matrix output back to the data layer. Each multiplication uses a constant matrix that is read from the data layer but will not change between invocations (enabling caching). We configured the benchmark to use three layers with square matrices with a side length of 1024 single precision floats. The baseline eTask implementation required the numpy, pickle, and pycuda python modules. For cold starts, the system has not seen any requests for the benchmark. For eTasks, this means that it must create a new worker and initialize any python dependencies before running the eTask code. For kTasks, the system already has KaaS executors initialized since they are independent of any particular request. However, those executors must parse the request, link against the specified CUDA libraries, and load all data from the data layer. On warm starts, both task types already have a worker ready to execute the request immediately.



(a) Warm Start



(b) Cold Start



(c) KaaS Cold vs Warm Start

Figure 8: Microbenchmark Results: *Kernel Run* is the time spent actually executing the request's kernels. *Kernel Init* is the time to link any needed precompiled CUDA libraries. *GPU Malloc* and *GPU Copy* encompass all device data management while *Data Layer* measures host data management. *Overheads* measure any additional tasks performed by the worker including request parsing, worker initialization, and Ray framework overheads.

Figure 8 shows the results of this experiment. We begin by observing that kTasks and eTasks have similar warm-start performance (Figure 8a). This is not surprising as both implementations must perform the same steps. Any differences are mostly down to implementation details. Next, we move our attention to cold starts (Figure 8b). Here we see the drastic impact of starting a new Python process for eTasks. Even though the microbenchmark only imports a minimal set of packages, it still takes an additional 400 ms to load (a 46x increase). In contrast, the only cold-start overheads experienced by kTasks come from warming the data caches with the constant matrices. To see this in more detail, Figure 8c compares warm and cold starts experienced by kTasks. As expected, the time to load an additional three matrices adds significant time to both the data layer and GPU memory management phases, but there are no other overheads.

5.3 Multitenant Workload

As we saw with the microbenchmarks, kTasks behave similarly to eTasks for a single warm workload. However, when there are multiple clients, the behavior changes dramatically. As discussed in §4.1.4, the eTask approach requires a policy that assigns GPUs to workloads exclusively, falling back to terminating and restarting workers as the number of workloads exceeds the available GPUs. These cold starts prevent the system from fully utilizing its GPU resources. KaaS, in contrast, is able to share a limited pool of GPUs more effectively.

We now evaluate four real-world applications in a multi-tenant environment with two scenarios: online and offline. In the online scenario, we evaluate how the system supports latency-sensitive workloads like model serving or interactive data analytics where requests arrive according to a Poisson process. To understand sustained throughput, we evaluate an offline scenario where all workloads submit requests as fast as possible. Two of our workloads, resnet50 and BERT, are deep-learning inference workloads generated by the TVM interface. These workloads consist of many small kernels operating over many small buffers. Deep learning models also have significant data re-use through the model weights. The cGEMM workload is a chained complex number matrix multiply using our Cutlass library interface. It multiplies a large constant 10000×25000 matrix by a narrow 100×10000 matrix that changes on each invocation. Both contain 32 bit complex floats. This results in small inputs and outputs but a large amount of cacheable data. Finally, the Jacobi workload uses the low-level KaaS interface to implement an iterative solver using the Jacobi method [21]. This

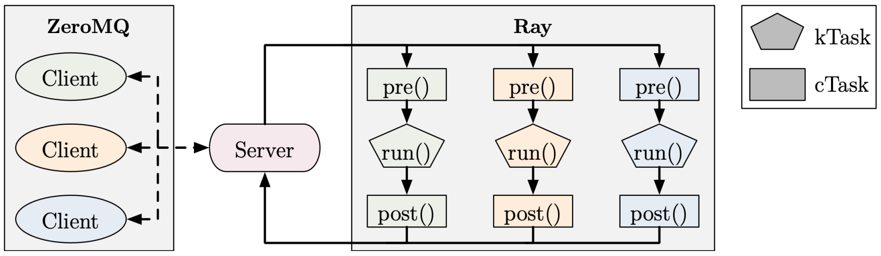


Figure 9: **Multitenant Environment:** Clients submit requests over a ZeroMQ socket to a server that submits per-request function pipelines to Ray. Each request pipeline involves optional CPU-only pre and post-processing steps with a GPU-based execution phase. The server proxies pipeline outputs back to the client.

	resnet50	**BERT**	**cGEMM**	**Jacobi**
Constant Memory	129 MB	1.3 GB	2.0 GB	0 MB
Dynamic Memory	6 MB	6 MB	8 MB	1 MB
GPU Time	4 ms	92 ms	39 ms	52 ms
CPU Time	10 ms	132 ms	0 ms	0 ms

Table 1: End to end workload properties.

workload uses KaaS's iteration control-flow mechanism to run a fast update kernel for 3000 iterations. The input matrix contains 512 × 512 `float32s` and is accumulated into a 512 × 1 `float64` output. There is no data re-use between invocations. Table 1 summarizes the properties of these workloads.

In addition to the GPU-based functions, the workloads also include pre and post processing functions that occur exclusively on the host with no GPU requirements. These functions form a graph that is lazily executed as inputs become available. Finally, this experiment uses multiple external clients that communicate with the Ray-based service through a network protocol using ZeroMQ [47]. Figure 9 shows the complete setup.

5.3.1 Offline Workloads

We begin with a throughput-oriented scenario where workloads from many clients submit requests as quickly as possible with no regard for latency. In this case, we measure the aggregate throughput of the system. This indicates whether or not the GPU resources remain highly utilized as the number of clients increases.

Figures 10 and 11 show the results from this experiment. We see that kTasks and eTasks both perform well when there are sufficient GPU resources to support the workloads. However, as we exceed the number of available GPUs, the eTask approach's throughput drops dramatically due to frequent worker cold starts. kTasks do not suffer significant slowdowns as we exceed available GPUs because the system is able to handle requests from different clients without restarting the KaaS executor.

The kTask implementation of cGEMM only begins to slow after about 20 replicas. This is the point at which we begin to exceed available GPU memory for cached weights and KaaS must begin evicting and re-loading constant buffers. Unlike the sharp performance drop seen with eTasks, there is a more gradual performance impact with kTasks since the cache can be managed at a fine granularity. Even at 32 replicas, we still see significantly higher throughput with KaaS. This comes down to the difference in cold-start behaviors between the two approaches. For eTasks, the system must boot a fresh python process and initialize any packages and state needed to run the model, while kTasks need only reload model weights from the host data cache.

Unlike cGEMM, resnet50 is a small model that fits easily within GPU memory. It experiences no significant slowdowns, even at 16 replicas. However, single-client warm-start performance is somewhat slower for kTasks than the eTask implementation. This is due to implementation differences between the two systems. The eTask implementation uses TVM's highly optimized C++ runtime while KaaS uses a simple Python implementation. While these differences are easily amortized by the larger kernels used by BERT or cGEMM, resnet50 consists of many very small kernels that accentuate system overheads. Fundamentally, the same work is done in both implementations and a more optimized KaaS executor implementation should behave similarly to TVM.

5.3.2 Online Workloads

In this experiment, we simulate a latency-sensitive online environment using the MLPerf Inference load generator in server mode [37]. Clients submit requests to

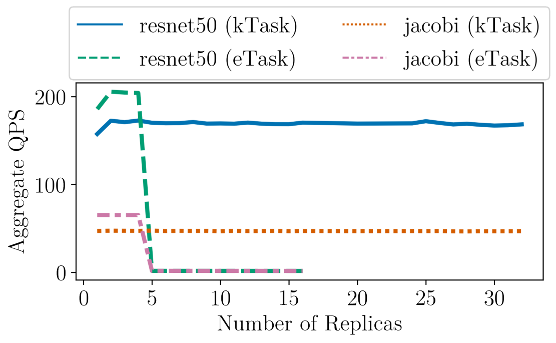


Figure 10: **Aggregate throughput of low-memory workloads.** These workloads fit easily within GPU memory and see no aggregate throughput loss in KaaS as we increase the number of replicas. eTask workloads require exclusive access to GPUs and must begin cold-starting workers after 4 replicas.

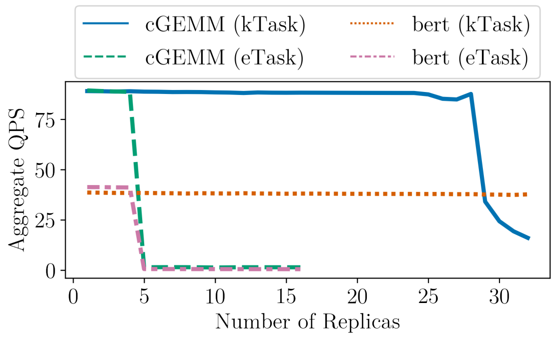


Figure 11: **Aggregate throughput of high-memory workloads.** These workloads consume a large amount of GPU memory. When the aggregate memory requirements exceed available GPU memory, KaaS must begin evicting and re-loading objects from the GPU memory cache. This leads to a gradual decline in performance as cache pressure increases.

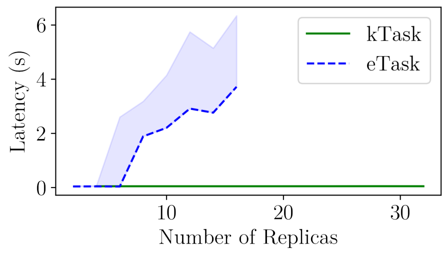
the framework according to a Poisson process with their mean arrival rate set to 80% of peak throughput (to ensure stability). We report the median and 90th percentile response latency for each configuration.

Figure 12 shows the results of this experiment. We see very similar behavior to the offline scenario. When the number of clients does not exceed available resources, both approaches behave reasonably well. However, when we exceed available resources, the eTask approach is forced to cold-start new actors for nearly every request, resulting in very poor tail latency. In effect, the tail latency becomes a measure of cold start time. This latency depends primarily on the python dependencies and initialization tasks for each workload. kTasks are able to effectively share GPUs, resulting in no significant impact on tail latency, even as we exceed the number of available GPUs. However, the cGEMM kTask begins to slow down after about 20 replicas as the aggregate constant memory exceeds GPU memory. Still, even at 32 replicas, the cGEMM kTask's tail latency is far below the eTask version due to the difference in cold start performance.

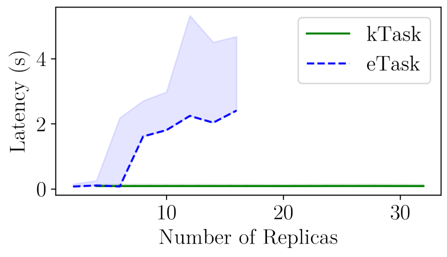
Figure 13 shows the CDFs of response latency at key points in the configuration space. Similar to Figure 12, we configure the experiment to submit requests at 80% of peak throughput to ensure queue stability. The maximum throughput of eTasks is significantly lower than kTasks so we plot CDFs for kTasks at both an equivalent submission rate to eTasks, and at peak kTask throughput. At 4 replicas, each replica has a dedicated GPU. In this case, both kTasks and eTasks behave similarly with only minor differences due to implementation details. As we exceed 4 GPUs, the system must begin sharing GPUs between replicas. Even with only one extra replica, we see the eTask tail latency suffer significantly since replicas may need to evict an existing eTask before cold-starting. As we increase the concurrency to 16 replicas, even the best 10th percentile latency suffers as replicas become nearly guaranteed to require a cold start. In contrast, kTasks experience very consistent latencies as the number of replicas increase. Even at 16 replicas, there is no noticeable reduction in latency.

6 Related Work

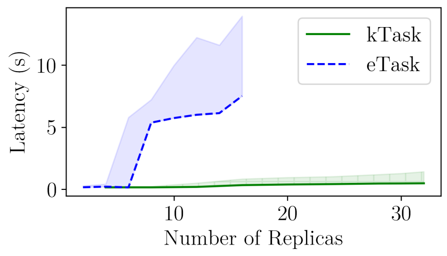
The need to share and manage accelerators in a distributed system is not a new one. In §2 we covered several broad categories of techniques to achieve this. In this section, we briefly cover several specific approaches.



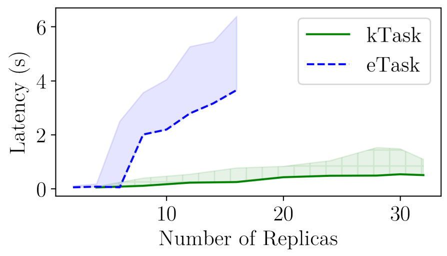
(a) resnet50



(b) Jacobi



(c) BERT



(d) cGEMM

Figure 12: **Online benchmark results** The median response latency is plotted along with the 90th percentile tail.

6.1 Single GPU Sharing

There are a number of techniques for sharing an individual GPU among multiple processes. Nvidia GPUs support multiple forms of sharing including CUDA contexts, Multi-Process Server (MPS), or Multi-Instance GPU (MIG) [29, 31]. CUDA contexts allow sharing but do not enforce resource usage limits. MPS and MIG provide static allocations of resources from a particular physical GPU to processes. Techniques like rCUDA are similar to MPS or MIG but proxy requests across a network to a remote GPU [18, 23]. In all cases, individual clients are assigned a specific GPU resource and must manage their own GPU resources explicitly for the lifetime of their allocation. While sharing increases the number of supported clients, those clients are still responsible for driving utilization of their allocation. The clients also require host CPU and memory resources, even if they are completely GPU bound. As the number of clients increase, CPU and memory underutilization can become significant. KaaS allows for flexible run-time resource allocations and integrates the serverless data layer into GPU memory allocations. Client code can run on any available physical GPU in the system on a per-request granularity without user intervention. One thing we can add here is how we solved rCUDA's usability problem with the TVM compiler and standard library.

6.2 Domain Specific Systems

Some domain-specific services, particularly for deep learning applications, are able to share accelerators among multiple clients. Google Cloud's TPU interfaces decouple TPU devices from host servers and manage communication and allocation. However, applications are still assigned specific devices for their lifetime and are responsible for driving adequate utilization [43].

RAMMER is a deep learning compiler and associated runtime that improves device utilization by abstracting deep learning models and accelerators into finer grained components and optimizing their scheduling [27]. KaaS applications also provide graphs of CUDA kernels which could be further optimized by systems like RAMMER.

PipeSwitch [4] and Salus [46] are two frameworks to facilitate sharing of a GPU between multiple cooperating deep learning processes on a single node. Like KaaS, they abstract GPU resources by handling memory allocation and kernel scheduling for the clients. Unlike KaaS, PipeSwitch still allows direct GPU access from host applications and requires applications to cooperate. Salus is more similar to the KaaS executor because it accepts graphs of kernel requests and associated memory needs. Neither system integrates into a larger serverless environment, requiring a strong association between host process and server/GPU. KaaS allows for higher-level resource allocation/de-allocation and co-scheduling of traditional serverless functions, kTasks, and data movement.

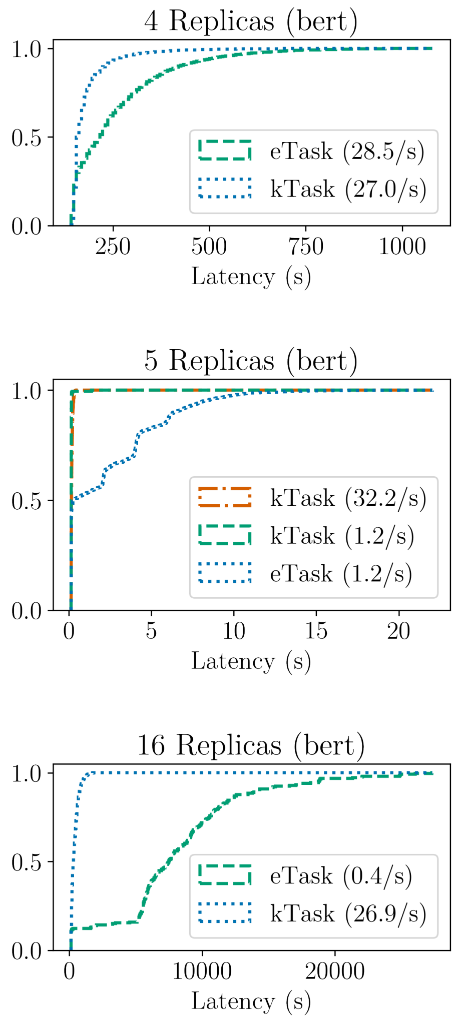


Figure 13: CDFs of BERT response latency for different numbers of replicas.

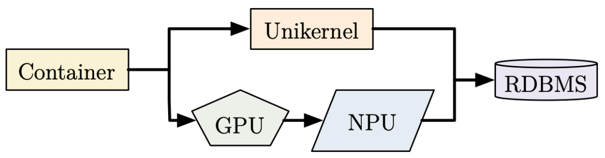


Figure 14: **Hetergeneous function graph** While each function can be implemented in different ways, the general graph structure is common to all.

There are also a number of systems specifically tailored to deep learning inference ([40, 20]) or training ([45]). These systems exploit the unique properties of deep learning, as well as cooperation from deep learning frameworks, to drive utilization and performance. Unlike these systems, KaaS is a general purpose serverless GPU interface for mutually unaware clients. It integrates natively into a broader general-purpose compute environment.

7 Resource-Specialized Function Types

KaaS serves as a proof-of-concept for a larger vision of *heterogeneous FaaS*. In a heterogeneous FaaS system, users design their application around a common data layer and function graph interface. However, the functions themselves can be implemented using any number of techniques. For example, some functions may want to take advantage of specialized unikernel operating systems while others may require more traditional full-featured containers [9]. As new specialized hardware technologies emerge, they can be integrated simply as a new function type. In fact, users or the serverless framework could decide between multiple implementations of the same functionality at runtime based on current system state as proposed by Infaas [40]. Even though the functions themselves are implemented in radically different ways, the common execution framework means that applications do not need significant changes to use new technologies.

7.1 Serverless Accelerators

FPGAs are an appealing target for serverless computing due to their high cost and ability to support diverse workloads. Microsoft has deployed FPGAs in their datacenters to support a wide range of workloads from deep learning to network function offload [34]. Ringlien, et al., propose ways that FPGAs could be made more multitenant [38]. As heterogeneous FaaS becomes more common, we could envision *serverless SoCs* that is optimized for KaaS-like systems. Since KaaS does not run user-provided code on the host, this serverless SoC would need only a low-power CPU and specialized protocol accelerators [24, 26]. Most of the power and area would be dedicated to the application accelerator. This could drastically improve power, cost, and deployment flexibility. These benefits have already been realized in the storage space [19, 6]. Microsoft's Catapult system and Google's TPU pods similarly demonstrate the potential benefits of resource-specific hardware platforms [43, 34].

8 Conclusion

Serverless computing is a wonderful thing; it simplifies programming at scale, drives higher resource utilization, and frees users from complex provisioning decisions. As expensive application accelerators like GPUs rise in importance in modern workloads, these properties are more important than ever. Simply adding GPUs to existing serverless techniques will not be sufficient; they are too different. In this paper, we presented KaaS, a truly serverless interface to GPUs that can integrate them naturally into the serverless ecosystem while preserving all the benefits we've come to know and love. KaaS frees users from tricky explicit allocations, effectively utilizes precious GPU resources in a multitenant environment, and frees system implementers to take full advantage of the unique properties of these devices. With KaaS, we were able to provide 50x higher throughput and 16x lower latency than FaaS when the number of tenants exceeded available resources.

References

[1] Intel. Oct. 2021. URL: https://www.intel.com/content/www/us/en/products/sku/204087/intel-xeon-platinum-8380h-processor-38-5m-cache-2-90-ghz/specifications.html.

[2] Dihuni. Oct. 2021. URL: https://www.dihuni.com/product/nvidia-a100-80gb-900-21001-0020-000-gpu-pci-e-4-0/.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. "PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 499–514. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/bai.

[5] Henry C. Baker and Carl Hewitt. "The Incremental Garbage Collection of Processes". In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977, pp. 55–59. ISBN: 9781450378741. DOI: 10.1145/800228.806932. URL: https://doi.org/10.1145/800228.806932.

[6] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. "Pelican: A Building Block for Exascale Cold Data Storage". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 351–365. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/balakrishnan.

[7] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables DevOps: Migration to a cloud-native architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[8] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. *The GraphBLAS C API Specification*. 2021. URL: https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. "SEUSS: skip redundant paths to make serverless fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Association for Computing Machinery, Apr. 2020, pp. 1–15. ISBN: 978-1-4503-6882-7. DOI: 10.

1145/3342195.3392698. URL: https://doi.org/10.1145/3342195.3392698.

[10] *CFS Scheduler*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[12] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. URL: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/.

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. "Clipper: A low-latency online prediction serving system". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[14] *cuBLAS Library User Guide*. DU-06702-001_v11.5. Nvidia. Oct. 2021. URL: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[15] *cuDNN Library Developer Guide*. PG-06702-001_v8.2.4. Nvidia. Aug. 2021. URL: https://docs.nvidia.com/deeplearning/cudnn/pdf/cuDNN-Developer-Guide.pdf.

[16] *CUTLASS*. Version 2.7. Nvidia. URL: https://github.com/NVIDIA/cutlass.

[17] *Deploy ML models to field-programmable gate arrays (FPGAs) with Azure Machine Learning*. Microsoft, Oct. 2021. URL: https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[18] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters". In: *2010 International Conference on High Performance Computing Simulation*. 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126.

[19] *FlashBlade Unified Fast File and Object Storage*. Tech. rep. PureStorage, 2021. URL: https://www.purestorage.com/docs.html?item=/type/pdf/subtype/doc/path/content/dam/pdf/en/solution-briefs/sb-pure-flashblade-uffo.pdf.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 443–462. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/gujarati.

[21] *Jacobi CUDA Graphs*. NVIDIA. URL: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/jacobiCudaGraphs.

[22] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 947–960. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jeon.

[23] *Juice Labs*. 2021. URL: https://www.juicelabs.co/.

[24] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. "A Hardware Accelerator for Protocol Buffers". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478. ISBN: 9781450385572. DOI: 10.1145/3466752.3480051. URL: https://doi.org/10.1145/3466752.3480051.

[25] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. "GPU Enabled Serverless Computing Framework". In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Mar. 2018, pp. 533–540. DOI: 10.1109/PDP2018.2018.00090.

[26] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. "Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 36–51. ISBN: 9781450383172. DOI: 10.1145/3445814.3446696. URL: https://doi.org/10.1145/3445814.3446696.

[27] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 881–897. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ma.

[28] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. "Ray: A distributed framework for emerging AI applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[29] *Multi-Process Service*. Nvidia. Oct. 2021. URL: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[30] *Nuclio*. iguazio, 2021. URL: https://github.com/nuclio/nuclio.

[31] *NVIDIA MULTI-INSTANCE GPU USER GUIDE*. Tech. rep. Nvidia, 2020. URL: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf.

[32] *OpenFaaS*. Version 0.21.1. OpenFaaS Ltd. URL: https://www.openfaas.com/.

[33] 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. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[34] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 114–122. ISSN: 0001-0782. DOI: 10.1145/2996868. URL: https://doi.org/10.1145/2996868.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[36] *Ray v1.6.0 Documentation*. URL: https://docs.ray.io/en/releases-1.6.0/using-ray-with-gpus.html#workers-not-releasing-gpu-resources.

[37] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Di-

amos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. *MLPerf Inference Benchmark*. 2020. arXiv: 1911.02549 [cs.LG].

[38] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. "System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 293–300. DOI: 10.1109/FPL.2019.00054.

[39] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. "Relay: A New IR for Machine Learning Frameworks". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 58–68. ISBN: 9781450358347. DOI: 10.1145/3211346.3211348. URL: https://doi.org/10.1145/3211346.3211348.

[40] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/romero.

[41] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. "Dandelion: a compiler and runtime for heterogeneous systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Nov. 2013, pp. 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. URL: https://dl.acm.org/doi/10.1145/2517349.2522715.

[42] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[43] Zak Stone. *Now you can train TensorFlow machine learning models faster and at lower cost on Cloud TPU Pods*. https://cloud.google.com/blog/products/ai-machine-learning/now-you-can-train-ml-models-faster-and-lower-cost-cloud-tpu-pods. Accessed: 2019-04-12.

[44] *VMware vSphere Bitfusion User Guide*. Tech. rep. 2021. URL: https://docs.vmware.com/en/VMware-vSphere-Bitfusion/4.0/vsphere-bitfusion-user-guide-45.pdf.

[45] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

[46] Peifeng Yu and Mosharaf Chowdhury. "Fine-Grained GPU Sharing Primitives for Deep Learning Applications". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 98–111. URL: https://proceedings.mlsys.org/paper/2020/file/f7177163c833dff4b38fc8d2872f1ec6-Paper.pdf.

[47] *ZeroMQ*. URL: https://zeromq.org/get-started/.