Efficient GPU Sharing for Serverless Workflows

Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein,
Andre Beck, Paarijaat Aditya, Manohar Vanga, Volker Hilt
Nokia Bell Labs

ABSTRACT

Serverless computing has emerged as a new cloud computing paradigm, where an application consists of individual functions that can be separately managed and executed. However, the function development environment of all serverless computing frameworks at present is CPU-based. In this paper, we propose to extend the open-sourced KNIX high-performance serverless framework so that it can execute functions on shared GPU cluster resources. We have evaluated the performance impacts on the extended KNIX system by measuring overheads and penalties incurred using different deep learning frameworks.

CCS CONCEPTS

• **Information systems** → **Computing platforms**; *Cloud based storage*; • **Computing methodologies** → *Graphics processors*; • **Computer systems organization** → *Multiple instruction, single data*; *Cloud computing*.

KEYWORDS

serverless; neural networks; deep learning; image processing; GPU

ACM Reference Format:
Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein,, Andre Beck, Paarijaat Aditya, Manohar Vanga, Volker Hilt, Nokia Bell Labs. 2021. Efficient GPU Sharing for Serverless Workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing (HiPS '21), June 25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3452413.3464785

1 INTRODUCTION

Serverless computing is emerging as a key paradigm in cloud computing. In serverless computing, the unit of computation is a function. When a service request is received, the serverless platform allocates an ephemeral execution environment for the associated function to handle the request. This model, also known as Function-as-a-Service (FaaS), shifts the responsibilities of dynamically managing cloud resources to the provider, allowing the developers to focus only on their application logic. It also creates an opportunity for the cloud providers to improve the efficiency of their infrastructure resources.

HiPS '21, June 25, 2021, Virtual Event, Sweden

ACM ISBN 978-1-4503-8388-2/21/06…$15.00
https://doi.org/10.1145/3452413.3464785

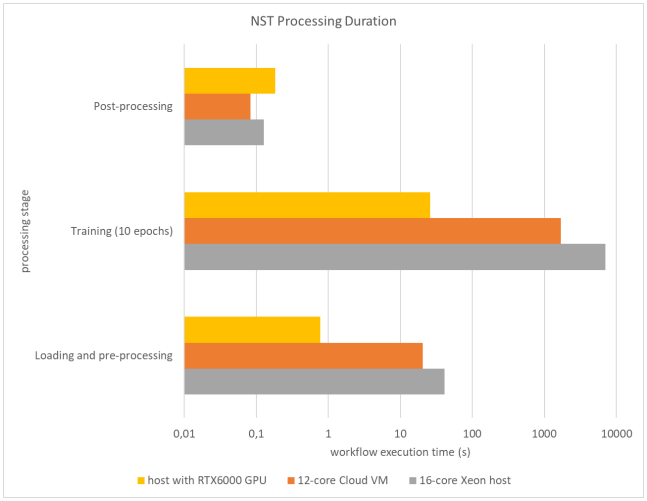


Figure 1: Execution duration of Neural Style Transfer Serverless Workflow on different KNIX deployments

The serverless computing paradigm has already created a significant interest in industry and academia. There have been a number of commercial serverless offerings (e.g., Amazon Lambda [2], IBM Cloud Functions [17], Microsoft Azure Functions [8], and Google Cloud Functions [10]), as well as several new proposals (e.g., OpenLambda [1] and OpenFaaS [13]).

Currently, these publicly available offerings only provide CPU access. However, with the rise of Deep Learning (DL) applications, the necessity for large-scale computations utilising other types of accelerators such as GPU (Graphical Processing Units) and TPU (Tensor Processing Unit) have emerged. Although the existing serverless platforms work well for simple applications, they are currently not well-suited for more complex services with high computational requirements as they occur in DL applications. Considering the advantages of serverless computing, it is natural to extend the framework for such applications.

As a DL application example, consider an image processing pipeline for Neural Style Transfer operation [14], which does inference using models of style transfer topology and needs to execute a number of consecutive functions:

- pre-processing: load and verify model and image data
- processing: create loss functions, train the model
- post-processing: transform data and output the results

We ran this sample workflow on different deployments of the KNIX platform, and found that the total time spent for the total execution of this workflow dramatically depends on the availability of a suitable accelerator, in particular for the function performing

training steps (see Figure 1). resulting in an increase of total execution time of several order of magnitude when omitting such accelerators. As a result of these differences, the use and adoption of serverless computing by a broader range of applications omitting available accelerators is severely limited.

Within the infrastructure layer, model training is the compute-intensive work in a typical machine learning (ML) pipeline, but predictions, or inference, accounts for up to 90% of total operational costs [6]. As another example, autonomous vehicle (AV) development may require thousands of GPUs to train machine learning models, but require millions of CPUs or GPUs to validate their software/models with log data or synthetic simulations [7]. Inference workloads have two main challenges:

- stand-alone GPU instances are designed for model training and are typically oversized for inference. Even at peak load, a GPU's compute capacity may not be fully utilised.
- different models need unique amounts of GPU, CPU, and memory resources. Selecting a GPU instance type that is big enough to satisfy the requirements of the most demanding resource often results in under-utilisation of other resources such as memory or CPU.

In this paper we propose an open-sourced FaaS framework allowing users to execute their applications in form of workflows in the cloud, using heterogeneous and shared CPU and GPU cluster resources. Users can request execution of parts of their applications to be executed on GPUs, and additionally can have fine-grained control the amount of GPU core and memory to be used for each function as part of the workflow. The development of sophisticated applications is simpler and more intuitive employing workflows containing multiple functions because a developer can define and manage the workflow of an application independently from its business logic, so that making changes to one does not affect the other. With the proposed framework, such workflows can simultaneously leverage a high-performance FaaS scheme for highly-scalable function executions, and profit from GPU-based computing power for the execution of special compute-intensive functions. In summary our contributions are as follows:

- A multi-tenant FaaS framework that supports accelerators in ML workflows, in particular in inference workflows where GPUs often remain under-utilised.
- Automatic assignment of specialised accelerator resources to workflow functions, based on their individual requirements.
- Grouping of different FaaS into a single sandbox. This is advantageous in particular for DL inference due to fast and efficient data sharing among different DL functions. Our approach preserves workflow modularity as each GPU requiring function can still be maintained separately if required.
- We have open-sourced our framework.

2 BACKGROUND AND MOTIVATION

2.1 Serverless Computing

Serverless has become a buzzword within the systems community when discussing running large workloads at scale. Among the serverless providers, AWS Lambda is the most widely used framework for serverless computing. Since Amazon launched and serviced AWS Lambda in 2014, Microsoft has launched the Azure Functions service, and Google has followed releasing Google Cloud Functions (GCF, alpha) both in 2016, respectively. In terms of open source Serverless Framework, there are Apache OpenWhisk, IronFunctions, Fn from Oracle, OpenFaaS, Kubeless, Knative, Project Riff, etc. Serverless providers generally hide much of the complex set-up of dedicated instances behind a veil, allowing the user to directly send requests without having to worry much about setting up an instance for running these requests. In addition, having users share a single serverless provider allows a greater usage of the resource hidden behind the provider, since all aspects of the hardware can be used on user demand. Another advantage of serverless benefits both the client and the provider: by having requests done in more fine-grained chunks, the provider can more exactly match the demand given by its users.

However, AWS Lambda as well as other Functions as a Service (FaaS) public services also imposes strict computing requirements. AWS Lambda functions run on a constrained environment, where the function execution time and maximum RAM cannot exceed 15 min and 10,240 MB, respectively, the ephemeral disk storage is limited to 512 MB, and the CPU assignment to a function is controlled by the amount of RAM assigned to it. Also, it is not possible to assign any kind of accelerator (GPU, TPU,...) to a FaaS by with any of the providers mentioned in this section.

2.2 Accelerated Serverless Computing

Kubernetes [20] services of major container cluster service vendors around the world provide the capability to schedule NVidia GPU containers, but it is generally implemented by allocating an entire GPU card to a container. This allows for better isolation and ensures that applications using GPU are not affected by other applications. This is suitable for DL model training scenarios, but it would be a waste for applications like model development and prediction scenarios. Kubernetes already includes some experimental support for managing AMD and NVidia GPUs across several nodes. However, plain Kubernetes only allows to schedule entire GPUs to pods. Once a GPU has been allocated it remains unschedulable for other pods, until the prior pod execution terminates.

Alibaba Cloud Container Service for Kubernetes (ACK) Serverless Service [11] has added support for GPU-enabled container instances. These instances, based on existing Alibaba Cloud Elastic Container Instances (ECI), allow for running AI-computing tasks in a serverless mode more quickly. But as with Kubernetes, this service is limited to scheduling of entire GPUs to pods.

Similarly, Nuclio [21] is an open source and managed serverless platform used to minimize overhead and automate the deployment of data-science based applications. However, their implementation is based on the common NVidia device plugin, only allowing to assign full GPUs to serverless functions.

2.3 Motivation

The demand is to allow more inference tasks to share the same GPU card, thus improving the GPU utilisation in the cluster.

This calls for a suitable partitioning scheme of GPU resources being in place. Here, the dimension of GPU resource partitioning refers to the partitioning of both GPU core and memory.

For fine-grained GPU device scheduling, there is currently no good solution. This is because the extended resources such as GPU in Kubernetes restricts quantities of extended resources to whole numbers, cannot support the allocation of complex resources. For example, it's currently impossible for a user to ask for 0.5 GPU in a Kubernetes cluster.

In contrast to comparable approaches the user can configure any function to use only a fraction of an entire GPU core and memory employing the KNIX serverless platform. This increases flexibility for the user and scalability of the serverless platform because it enhances the GPU utility through sharing of the remaining GPU fraction with other FaaS (see Fig. 2).

With our design users does not need to possess knowledge in detailed GPU sandbox set-up for their function and cluster management because this is abstracted away from them. All these benefits come in addition to the finer-grained resource allocation for KNIX microfunctions and workflows as discussed in the previous section.

3 RELATED WORK

Traditional NVidia docker [22] environment simply assigns a GPU to a container, which causes program failure if multiple containers share the GPU and use GPU memory dynamically.

3.1 GPU attachement to CPU service

Prior work has been done on attaching a GPU to the conventional Lambda framework [18]. Our approach differs from this in that KNIX completely decouples the GPU from the CPU, allowing the user to define function requirements for GPU using device plugin and a GPU virtualisation scheme. A GPU statically attached to Lambda approach poses a few downsides. By doing this, a FaaS provider is essentially doing the same thing as the traditional serverful approach, but on a smaller scale. By decoupling the GPU requests from the CPU, the provider can rent out GPU usage as well in addition to the CPU. However, if the user wants to use both CPU and GPU simultaneously, this is not possible with the GPU attachments approach. Recently, cloud-hosted inference services (AWS Elastic Inference [6], Google Cloud Inference API) have emerged to attach the right amount of GPU-powered inference acceleration to selected CPU services, raising serious privacy concerns because this model calls for sending of private and privileged data over the network to remote servers. Also, these services cannot be used to extend the capabilities of serverless offers such as AWS Lambda or GCF.

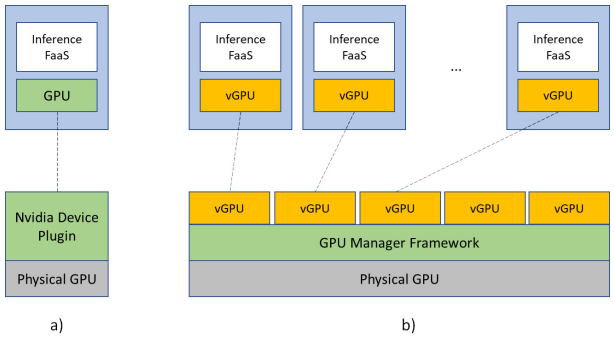


Figure 2: Illustration of a) traditional methods of processing GPU workloads on Kubernetes cluster versus b) KNIX method of dealing with this issue.

3.2 CUDA Without Environment Setup

Prior work has been done on remote GPU acceleration of GPU-enabled workloads [12], however without employing the serverless approach. The authors describe a framework to enable remote GPU-based code acceleration, thus permitting the reduction of accelerators in the cluster and consequently the global energy consumption. Using this approach, the user can make calls to CUDA through a driver. However, on the back end of the driver, CUDA requests are sent over the network. This has the inherent disadvantage of lacking fault tolerance in case of network outages.

3.3 GPU Sharing Frameworks

While GPU virtualisation has been extensively studied for VM, limited work has been done for containers. One of the key challenges is the lack of support for flexible and fine-grained GPU sharing between multiple concurrent containers. This limitation leads to low resource utilisation when a GPU device cannot be fully utilised by a single application, e.g. due to the burstiness of GPU workload and the limited GPU memory bandwidth. In the following we describe a number of solutions tackling these problems.

3.3.1 KubeShare. KubeShare is a framework to support GPU sharing in Kubernetes. In order to address the GPU utilisation and performance interference problems in shared resource environment, KubeShare manages GPUs as first-class resources, and provides a simple yet powerful resource specification to let users to request specific vGPU binding for their containers using the GPU identifier, and to control the location of their GPU by specifying the locality constraints. For this purpose, KubeShare introduces a new Kubernetes API called SharePod. However, this new API introduces compatibility problems in other parts such as the KNative service generation on Kubernetes.

3.3.2 GPUShare Scheduler Extender. Aliyun [16] is a container service open source project developed by Alibaba. This approach solves the GPU resource fragmentation problem incurred by other sharing approaches by developing a scheduler-extender in Kubernetes. However, the platform only limits the GPU memory usage of a container, not the computation resource usage. Additionally, GPUShare does not support resource isolation because all tasks share time slices on the same set of GPUs and may therefore affect each other.

3.3.3 GPU Manager Framework. The GPU manager framework [15] is a recent work that further extends Aliyun to support GPU usage isolation on kernel execution based on the *LD_LIBRARY_PATH* API interception technique. *LD_LIBRARY_PATH* is an environment variable for Linux systems that affects the runtime link of programs, allowing some directories to be loaded before the standard set of directories. The framework enables Kubernetes to not only run more than one Pod on the same GPU, but also gives QoS guarantees to each Pod. It allows to configure and limit both GPU and memory shares for each pod to be deployed on the cluster.

However, the authors did not test the overhead incurred in a serverless, cloud-native environment like KNIX, or the sharing of GPU resources between different sandboxes.

Therefore, we have used the GPU Manager framework as a starting point to extend the KNIX capabilities using fractional shared GPU resources for function and workflow deployments and to evaluate the incurred impairments with respect to platform overhead, application performance and resource isolation.

4 KNIX SERVERLESS PLATFORM

We used the serverless framework SAND, which recently has been renamed to KNIX, as a starting point for our test system implementation. KNIX MicroFunctions [19] is an open source serverless computing platform for KNative as well as bare metal or virtual machine-based environments. It combines container-based resource isolation with a light-weight process-based execution model that significantly improves resource efficiency and decreases the function startup overhead.

First the capability to execute Python KNIX functions in sandboxes using NVidia GPU resources for both ansible and helm deployments of KNIX has been added. GPU nodes are detected and configured automatically by the platform. The configuration details for KNIX deployments integrating NVidia GPU nodes are described in [19]. GPU Manager [15] has been used for managing the NVidia GPU devices in our Kubernetes test cluster. We have chosen GPU Manager because it allows flexible sharing of GPU memory and computing resources among multiple isolated containers by partitioning physical GPUs into multiple virtual GPUs (vGPUs). Each Kubernetes pod can be assigned with vGPUs as requested.

5 KNIX PLATFORM DESIGN

The target of KNIX GPU framework extension is to share both GPU memory and compute resources among containers with minimum cost and maximum performance enhancement. There are some challenges in achieving this goal:

- Transparency: It should not be required to modify any Kubernetes code or container images for GPU sharing. An application executed with shared vGPUs should behave as if it was executed on physical GPUs.
- Low overhead: The application performance with vGPUs should be as close as possible to the performance with physical GPUs.
- Isolation: KNIX should manage vGPU resource sharing and allocation/de-allocation for microfunctions and workflows and create the corresponding sandboxes so that they are completely isolated from each other.

6 KNIX PLATFORM IMPLEMENTATION

In the following we list the main architectural component of the KNIX serverless framework architecture and the modifications required to allow for GPU sharing, for a more detailed description of the KNIX architecture we refer to [5] and our github site on [19].

6.1 Management Service

The KNIX Management Service (MS) consists of functions that enables users to sign up, login and manage microfunctions as well as workflows. For this purpose the MS accesses its own storage to manage application data. The MS is also responsible for the calculation of KNative service configurations and parameters in case of a deployment on Kubernetes. We needed to extend the MS functions in several parts:

The functions responsible for adding and modifying microfunctions and workflows needed to be extended so that they can handle the requirements definitions for deployments on a sandbox running on a GPU host. These configurations are provided by either the KNIX GUI or by the KNIX SDK.

The MS is also responsible for configuration of KNative services representing microfunction workflows in case of a helm deployment, allowing to define resources in form of limits and requests. Requests and limits are the usual KNative mechanisms to control resources such as CPU and memory. In our case, these mechanisms are extended to additionally address vGPU resources for a microfunction.

The MS analyses the entire workflow description containing configuration information about all associated microfunctions, and searches for added definitions about GPU limits or requests. If it contains multiple microfunctions with GPU requirements, the MS sums up their requirements to configure the KNative service representing the corresponding workflow and the respective sandbox.

Finally, logic has been added to the MS to allow for queries on the allocatable vGPU capacities in the cluster prior to each workflow deployment. This enables the MS to detect if the vGPUs required by the workflow are indeed available in the cluster, so that pending workflows due to GPU capacity over-subscriptions are avoided.

6.2 Sandbox Image Flavours

We needed to create new KNIX sandbox GPU images flavour beyond the flavours already available for handling of the different languages supported by KNIX (Java, Python). Besides Python language support, the new sandbox flavours needed to provide the CUDA tools, math and deep neural network libraries and runtimes required to address the GPU from within this container. On the cluster GPU nodes, an installation of the NVidia Container Toolkit and Docker runtime was required to run this sandbox image flavour.

6.3 KNIX Workflow Description Extensions

The data structures describing a workflow has been extended to allow for each workflow function to configure the amount of virtualised GPU core and memory limits and requests required for function execution.

7 TESTBED EVALUATION

We build a cluster which is composed of five hosts created using vagrant and kubespray. One of the hosts acts as the Kubernetes master, with the remaining hosts acting as worker nodes. One of the worker node has a physical GPU. Detailed information of the testbed hardware can be found in Table 1. We use Kubernetes version 1.17.5 as the container orchestrator. For the GPU worker node, we use the driver version 450.102.04, CUDA version 10.1, cuDNN version 7.6.5. Docker version 19.03.14 is used as container provider. nvidia-docker [22] is installed on the GPU worker node, using the NVidia runc runtime for sandbox executions.

Table 1: Hardware configuration of the cluster

Kubernetes GPU Node	
CPU	16x Intel(R) Xeon(R) W-2245 CPU
RAM	128 GB
GPU	NVidia Quadro RTX 6000
Kubernetes Master/Worker (inside VM)	
CPU	8
RAM	32 GB

Table 2: Measured GPU virtualisation overhead

	$t_{baremetal}$ (s)	t_{vGPU} (s)	Overhead (%)
Tensorflow MNIST	298.2	300.0	0.67
Pytorch MNIST	189.1	222.3	14.9
MXNET MNIST	39.32	33.50	14.8

To examine the impact of GPU sharing on KNIX performance we use a helm deployment, following the description for helm deployments with GPUs support in [19]. We use helm v3.2.4 and KNative v0.16.0 for the KNIX platform deployment. Instead of deploying the NVidia plugin to enable usage of the GPU as cluster resource, we build and deploy the GPU Manager device plugin as a daemonset, following the description in [15].

7.1 Overhead

In order to identify the overhead of sharing GPUs incurred by the selected GPU Manager approach, we evaluate the performance of GPU applications deployed in KNIX sandboxes with different resource configurations. Today, most GPU applications are built with DL frameworks as they provide an easy way to construct deep networks. However, different DL frameworks have different libraries and resource allocation strategies, leading to different overhead when sharing GPUs among sandboxes. For our tests we have selected the popular Tensorflow [4], Torch [23] and MXNet [9] frameworks. We run the MNIST application on three different DL frameworks in both the bare metal and inside the KNIX sandbox, and measure its execution time. MNIST application is a program that detects handwritten digits with database MNIST [20] using Convolutional Neural Network. We choose this application for fairness as all evaluated frameworks provide it in their official examples.

The measurement results are shown in Table 2. $t_{baremetal}$ in the table refers to the execution time of the test applications running on the host with a full GPU assigned 100% to it, while the t_{vGPU} refers to the execution time of the application running in a sandbox with shared GPUs. Overhead refers to the difference between $t_{baremetal}$ and the t_{vGPU}, which is calculated using equation 1).

$$Overhead = \frac{t_{vGPU} - t_{baremetal}}{t_{baremetal}} * 100\% \quad (1)$$

All tests are repeated 10 times, and the average value is used. As shown in Table 2 the measured overhead of GPU Manager in case of the Tensorflow test application is still quite low and therefore can support performance to be achieved as native environment for the KNIX platform. Only in the case of Torch and MXNet frameworks, quite large overheads of almost 15% are measured. This overhead is related to large fluctuations observed in the GPU utility function during training phase of the MNIST application for both of these frameworks, while the GPU utility was flat during the measurements with TensorFlow. This apparently allowed for a more efficient allocation of vGPU resources for this framework by the GPU Manager.

7.2 Application Performance

Table 3: Configurations of vGPUs for application performance experiments

Number of vGPUs	1	2	4	6	8
Comp. resource per vGPU (%)	100	50	25	15	12.5
Memory per vGPU (GB)	2	2	2	2	2

This experiment evaluates the impact of the partitioning of GPU resources on application performance. We partition a physical GPU into 1,2,4,6, and 8 vGPUs, and assign them to KNIX sandboxes each executing the same DL test application (MNIST [3], Fashion-MNIST [3], Cifar10 [26], Matmul [25]), and measure the application execution time. Tensorflow framework is used to run these DL applications. The vGPUs configurations are detailed in Table 3, corresponding application execution times are shown in Fig. 3. The x-axis is the number of vGPUs and the y-axis represents the execution time of the application. Although the physical GPU has around 24GB memory, the actual memory available is less than that. Nevertheless it was possible to execute eight test functions in parallel on a shared GPU. As shown in Figure 3, the execution time is linear to the computing resources of the vGPU for large number of concurrently executing pods.

The execution time for small number of pods is non-linear, in particular for applications handling large model sizes. This is assigned to dynamic and elastic resource allocation effects through

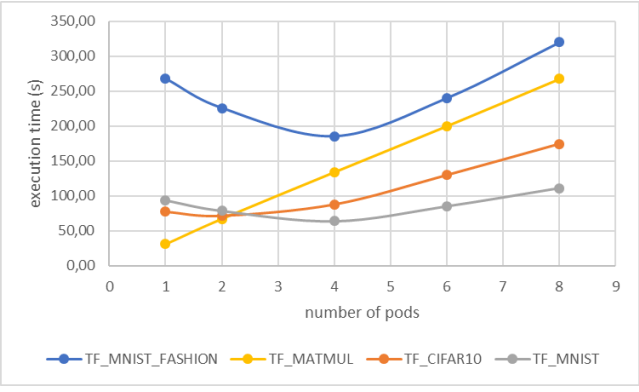


Figure 3: The performance of DL test applications under different partitioning of computing resources

the GPU Manager, which allows tasks to exceed the configured vGPU settings and dynamically re-calculates the upper limit of vGPU resources during sandbox execution, therefore increasing the measured GPU utility (see Section 7.4).

7.3 Resource Isolation

Resource isolation means that the allocation of resources to one sandbox should not have any unintended impacts on other sandboxes. To demonstrate if GPU Manager based approach can achieve GPU resource isolation among KNIX sandboxes, we launch 2, 4, and 8 sandboxes on one physical GPU and measure the memory and GPU utilisation of these sandboxes. Each KNIX sandbox is assigned to one vGPU.

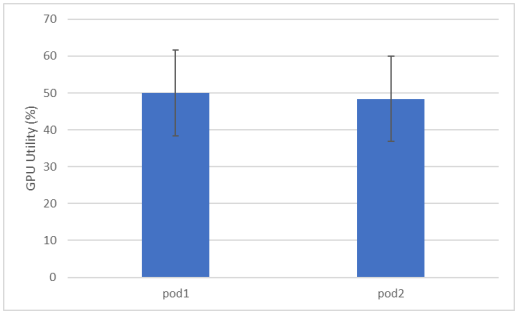
Table 4: Configurations of vGPUs for isolation experiments

Number of vGPUs	2	4	8
Computing Resource per vGPU	50%	25%	12.5%
GPU memory per vGPU (GB)	7.68	3.84	1.54

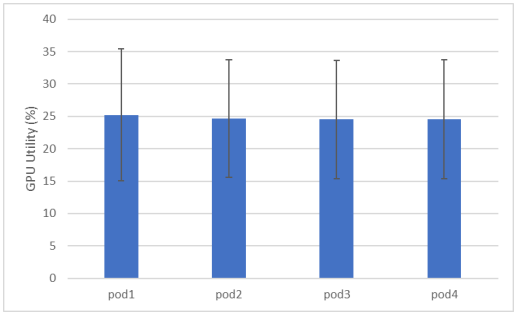
The KNIX GPU workload is performing random matrix multiplications using the Tensorflow framework. Table 4 shows the configurations of vGPUs for each measurement performed, and Fig. 4 and 5 illustrate the measurement results. In figure 5, the x-axis represents the elapsed time. The Tensorflow framework used in our experiments adopts a one-time resource allocation strategy on start-up, meaning that the entire available GPU memory is claimed when the Tensorflow session starts. This static vGPU memory scheduling is clearly visible in Fig. 5.

In Fig. 4, the y-axis refers to the measured GPU utilisation for different number of sandboxes executing in parallel. As expressed by the large standard deviation in this figure, the GPU utilisation shows fluctuations over time, mainly because the GPU is a non-preemptive device. Fig 4a illustrates that two sandboxes are running on one physical GPU. The average GPU utilisation of one sandbox is 49.99% and the other is 49,42%. The maximum GPU utilisation of each single sandbox configured to be 50%, but as the vGPU Library component monitors the resource utilisation of each process running on the GPU at a given time interval, and the process can temporarily use more resources than it has requested between monitoring intervals and therefore deviate from its configured target utilisation. In Fig. 4b four sandboxes execute concurrently sharing one physical GPU, and their respective GPU utilisations are 25,22%, 24,63%, 24,51% and 24,54%.

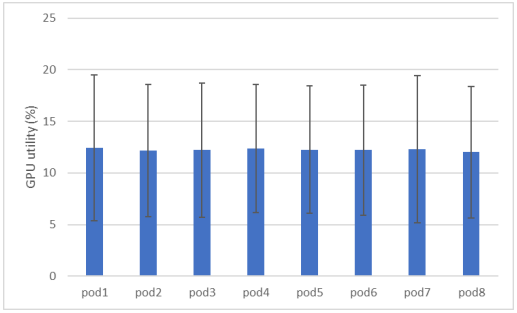
When a GPU is shared by eight sandboxes (Fig. 4c, the average GPU utilisation of sandboxes is around 12.4% and the standard deviation of the mean GPU utilisation value among sandboxes is less than 0.13% in our measurements. The experimental results reveal that the chosen approach effectively isolates the GPU resources of each sandbox when sharing GPUs among sandboxes. However, the monitoring strategy chosen for GPU virtualisation results in large fluctuations of 20-30% in the GPU utility function. However, the fluctuation are only pronounced in the GPU utility function, while the memory utility yields the expected value over the entire sandbox lifetime (see Figure 5).



(a) 2 vGPUs



(b) 4 vGPUs



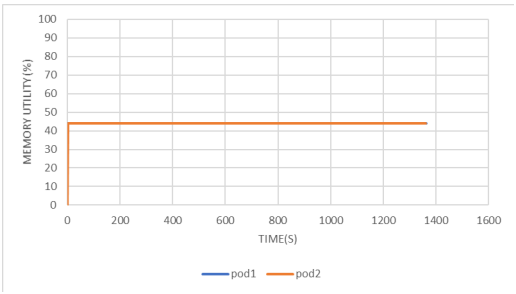
(c) 8 vGPUs

Figure 4: GPU utilisation with different number of vGPUs

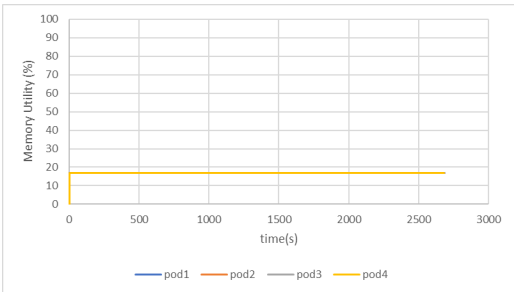
7.4 Dynamic Sandbox Resource Allocation

The GPU Manager approach offers the special feature of elastic GPU resource allocation. To investigate this property for KNIX sandboxes we launched two workflow sandboxes on one physical GPU, executing concurrently. One sandbox has been configured for vGPU usage of 0.1 GPUs and 4GB memory and the other requires 0.9 GPUs and 4 GB memory. The matrix multiplication workflow is executed with Tensorflow in two sandboxes. At first, one sandbox with 0.1 GPUs executes the Tensorflow application. After t=70s, another sandbox with 0.9 GPUs starts execution of the Tensorflow application, and terminates execution at t=890s. The sandbox with 0.1 continues execution until it terminates as well at t=1300s

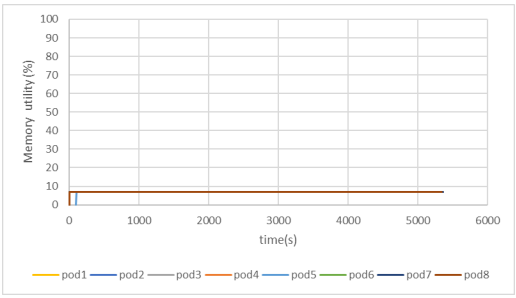
The experimental results are shown in Fig. 6. Compared to the hard resource limit case with static resource allocation, the execution time of the workflow with 0.1 GPUs is reduced by 80%, and the execution time with 0.9 GPUs is slightly increased by 15%. The average GPU utilisation of the physical GPU is increased by 73.5%



(a) 2 vGPUs



(b) 4 vGPUs



(c) 8 vGPUs

Figure 5: Memory utilisation with different number of vGPUs

during this experiment. The experimental results illustrate that elastic resource allocation can significantly improve the application performance and the GPU utilisation, in particular when using low values for resource allocations.

7.5 Fast Data Sharing

Machine learning uses statistical algorithms that learn from existing data, a process called training, in order to make decisions about new data, a process called inference. During training, patterns and relationships in the data are identified to build a model. This model allows a system to make intelligent decisions about data it has not encountered before. Optimising models compresses the model size so it runs quickly. Training and optimising machine learning models require massive computing resources, so it is a natural fit for the cloud. But, inference takes a lot less computing power and is often done in real-time when new data is available. Getting inference results with very low latency is important to ensure an IoT applications can respond quickly to local events. The inference functions themselves can run on shared resources

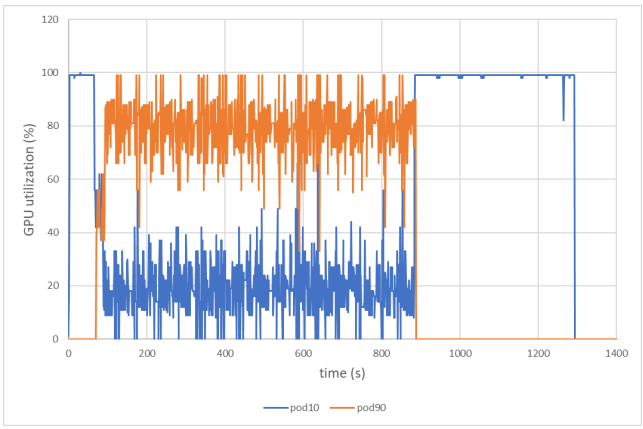


Figure 6: Dynamic GPU resource allocation dynamics

equipped with accelerators such as shared GPUs. Such a distributed scheme however assumes that a fast and efficient way to share data between multiple functions performing inference concurrently is in place. However, this is generally not the case. An obvious choice to share trained model data between a multitude of functions is to externalise the model state using a storage service such as Amazon S3, but the incurred latencies and for up- and downloading model data would be prohibitive for many applications.

As an alternative, Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) can be configured to allow sharing data among several pods running inference functions, with the following characteristics:

- If a Pod is destroyed, data is persisted.
- a PV can be used to share data between sandboxes, whether in the same pods or across pods.
- a PV does not allow to share data between pods running on different nodes.
- PVs can be accessed by many Pods at the same time.

Exchanged model data can be large: as an example, even the small, pre-trained VGG19 model used for neural style transfer measurements in section 1 already exceeds a size of 550MB [24]. KNIX offers an interesting alternative for sharing data among functions. Instead of running each GPU-using inference function in its own sandbox, microfunctions belonging to the same workflow could be configured to share GPU sandbox resources and additionally exchange model data via the local file system in a fast and effective way. We have prepared KNIX sandboxes and have evaluated Keras model data loading latencies using two different scenarios:

(1) data exchange via the GPU-using Sandbox filesystem, or
(2) via PV/PVC for sandboxes on the same GPU node

We performed measurements of model and weights loading, both via the intrinsic Tensorflow SavedModel and via the older HDFS loading scheme. Each experiment has been repeated 10x to build the measurement mean values. The results in Fig. 7 show the time required to share Keras VGG19 model and weights data via reading/writing to the local file system is slightly shorter than the time required to load the same VGG19 model data into GPU memory via the PV/PVC in all measurements, showing an small

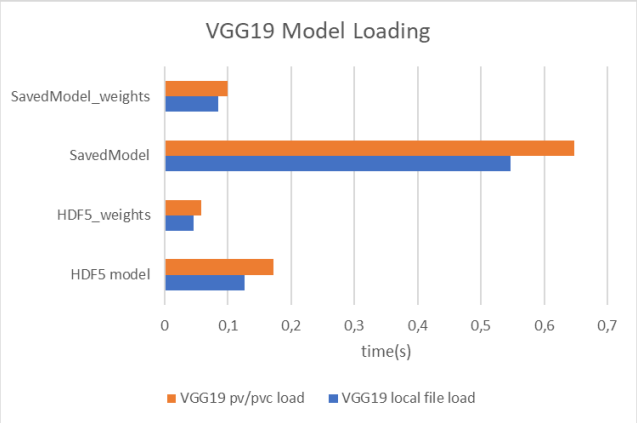


Figure 7: Comparison of read/write speeds of GPU-using KNIX sandboxes

advantage of 14-26%, depending on the data loading scheme. We therefore claim that KNIX offers additional benefit through the grouping of functions inside the same KNIX sandbox, because of the model sharing speed-up through the direct data exchange via the sandbox file system. However, in a distributed FaaS environment, if a model first needs to be unloaded from the GPU memory after a microfunction finishes, and before the follow-up function can execute, this unloading time would represent a challenge for the function start-up times because this would add additional overhead. More detailed evaluations of interactions between workflows with sequential or concurrent microfunction executions and the KNIX serverless platform are left for future work.

8 CONCLUSION

In this paper, we apply an approach for sharing GPU memory and computing resources to our high-performance serverless platform KNIX. Owing to its open design, only selected parts the framework required modifications. Employing installations of nvidia-docker, Kubernetes, KNative and the GPU-manager framework, we succeeded to partition a physical GPU into multiple vGPUs, and assigned these vGPUs to serverless KNIX microfunctions and workflows. The selected approach enables allocation elasticity by temporarily modifying container resources, therefore further improving GPU resource utilisation.

Experimental studies have been conducted to evaluate the performance impacts and overheads of GPU sharing using to different DL frameworks. The results reveal that the measured overhead introduced by the GPU sharing framework to DL frameworks is <15%, so that in general the GPU resources can still be considered as effectively managed. When measured over long execution times the fairness variations for concurrently executing functions remains low (<0.13%). However, our measurements still show potential for further optimisation as dynamic GPU resource allocation and deallocation results in strong fluctuations of assigned function's GPU resources. Finally, measurements on application performance and fast data sharing between different shared GPU-using KNIX microfunctions show potential for serverless application speed-up.

REFERENCES

[1] 2016. OpenLambda Dev Meeting July 5. http://open-lambda.org/resources/slides/july-5-16.pdf.

[2] 2021. AWS Lambda. https://aws.amazon.com/lambda/.

[3] A MNIST-like fashion product database. Benchmark 2021. A MNIST-like fashion product database. Benchmark. https://github.com/zalandoresearch/fashion-mnist.

[4] Martín 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. 2016. TensorFlow: A system for large-scale machine learning. arXiv:1605.08695 [cs.DC]

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[6] Amazon Elastic Inference 2021. Amazon Elastic Inference. https://aws.amazon.com/machine-learning/elastic-inference/.

[7] Autonomous Vehicle and ADAS development on AWS 2020. Autonomous Vehicle and ADAS development on AWS. https://aws.amazon.com/blogs/industries/autonomous-vehicle-and-adas-development-on-aws-part-1-achieving-scale/.

[8] Azure Functions—Serverless Architecture | Microsoft Azure 2021. Azure Functions—Serverless Architecture | Microsoft Azure. https://azure.microsoft.com/en-us/services/functions/.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[10] Cloud Functions - Serverless Environment to Build and Connect Cloud Services | Google Cloud Platform 2021. Cloud Functions - Serverless Environment to Build and Connect Cloud Services | Google Cloud Platform. https://cloud.google.com/functions/.

[11] Container Service for Kubernetes [n.d.]. Container Service for Kubernetes. https://www.alibabacloud.com/product/kubernetes.

[12] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation*. 224–231. https://doi.org/10.1109/HPCS.2010.5547126

[13] Alex Ellis. 2017. Functions as a Service (FaaS). https://blog.alexellis.io/functions-as-a-service/.

[14] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015. A Neural Algorithm of Artistic Style. *CoRR* abs/1508.06576 (2015). arXiv:1508.06576 http://arxiv.org/abs/1508.06576

[15] GPU Manager is used for managing the nvidia GPU devices in Kubernetes cluster. 2021. GPU Manager is used for managing the nvidia GPU devices in Kubernetes cluster. https://github.com/tkestack/gpu-manager.

[16] GPU Sharing Scheduler for Kubernetes Cluster [n.d.]. GPU Sharing Scheduler for Kubernetes Cluster. https://github.com/AliyunContainerService/gpushare-scheduler-extender.

[17] IBM Cloud Functions [n.d.]. Cloud Functions - Overview | IBM Cloud. https://www.ibm.com/cloud/functions.

[18] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim. 2018. GPU Enabled Serverless Computing Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 533–540. https://doi.org/10.1109/PDP2018.2018.00090

[19] KNIX 2021. KNIX. https://github.com/knix-microfunctions/knix.

[20] Kubernetes Schedule GPUs 2021. Kubernetes Schedule GPUs. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[21] Nuclio Serverless Functions [n.d.]. Nuclio Serverless Functions. https://nuclio.io/.

[22] NVIDIA-Docker [n.d.]. Build and run Docker containers leveraging NVIDIA GPUs. https://github.com/NVIDIA/nvidia-docker.

[23] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6 (2017), 3.

[24] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[25] tf.linalg.matmul API 2021. tf.linalg.matmul API. https://www.tensorflow.org/api_docs/python/tf/linalg/matmul.

[26] The CIFAR-10 dataset 2021. The CIFAR-10 dataset. https://www.tensorflow.org/datasets/catalog/cifar10.