SLA-Driven ML Inference Framework for Clouds with Heterogeneous Accelerators

Junguk Cho [1] **Diman Zad Tootaghaj** [2] **Lianjie Cao** [2] **Puneet Sharma** [2]

Abstract

The current design of Serverless computing frameworks assumes that all the requests and underlying compute hardware are homogeneous. This homogeneity assumption causes two challenges in running ML workloads like Deep Neural Network (DNN) inference services on these frameworks. Such workloads can have various request types and might require heterogeneous accelerators. First, existing serverless frameworks are threshold-based and use simple query per second or CPU utilization as autoscaling rules, thus ignoring heterogeneous requests and accelerators, resulting in sub-optimal performance. Second, ignoring infrastructure heterogeneity for workload scheduling and inference request distribution can lead to further performance inefficiencies. To address these challenges, we propose SLA-aware ML Inference Framework, which is a novel application and hardware-aware serverless computing framework to manage ML (*e.g.*, DNN) inference applications in a heterogeneous infrastructure. Our framework designs an intelligent autoscaling strategy by leveraging rich, precise workload-specific metrics and heterogeneous GPU compute capability. We schedule functions on the suitable GPU accelerators and proportionally distribute inference requests to the deployed functions based on the autoscaling decision. In addition, our framework enables efficient shares of GPU accelerators with multiple functions to increase resource efficiency with minimal overhead. Unlike prior works, we use application-specific SLA metrics to make scheduling/autoscaling decisions. We implement a prototype of our framework based on the *Knative* serverless framework and evaluate its performance with various DNN models.

1 Introduction

Serverless computing has emerged as a new and compelling paradigm for the deployment of cloud-native applications and services due to simple programming abstraction, minimal infrastructure maintenance, and flexible cost management with *pay-as-you-go* billing model. In addition, it provides new features that make building scalable microservices easier and more cost-effective. Further, the popularity of ML applications, particularly Deep Neural Network (DNN) inference serving applications on different domains, have lured programmers to host these models on serverless platforms (Ishakian et al., 2018; Bhattacharjee et al., 2019b; Feng et al., 2018; Wang et al., 2019; Bhattacharjee et al., 2019a). Since the requests are generated by different users with different computing and memory demands, a heterogeneous system with diverse computing power and memory capacity helps to improve QoS and cost savings (Baldini et al., 2017; Yan et al., 2016). Especially in many DNN-based use cases (*e.g.*, deep learning-based vision applications), hardware accelerators are required to meet stringent latency requirements/SLAs. This makes the support for heterogeneous hardware on serverless platforms essential.

However, the inference services leveraging heterogeneous hardware (*e.g.*, GPU, FPGA, and TPU) in serverless computing have not been extensively explored yet. This is mainly because the current design of serverless computing framework is based on the assumption that all requests and computing hardware are homogeneous. This assumption causes several problems in heterogeneous environments. First, existing serverless frameworks do not consider the application and hardware-specific metrics to make autoscaling decisions. Most of them mainly use simple query per second (QPS) or CPU utilization as autoscaling metrics without considering the heterogeneity of incoming inference requests and hardware accelerators, leading to sub-optimal performance, as we will see in the evaluation section.

Since autoscaling is a crucial feature of serverless computing to meet performance requirements and cost in real-world use cases, it is essential to design an autoscaling approach that utilizes the resources efficiently and responds quickly to the dynamic workload changes and hence application

[1] During this project, some of the contributors have changed their affiliated institutions. All of the work was done by them during their employment at Hewlett Packard Enterprise. [2]Hewlett Packard Labs. Correspondence to: Diman Zad Tootaghaj <diman.zad-tootaghaj@hpe.com>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

resource requirements. Second, existing serverless frameworks are designed to place and schedule the functions on homogeneous infrastructure[2]. In addition, the incoming requests can also be heterogeneous with respect to the task type and the data size within a specific task, e.g., batch sizes for image inferencing services. Existing serverless frameworks use simple round robin-based policies for workload scheduling and request distribution without considering this heterogeneity.

In this paper, we focus on designing a serverless framework for such inference services on heterogeneous hardware accelerators and try to answer the following questions: (i) what are the key metrics to capture inference performance and how to efficiently collect them, (ii) how to make autoscaling and placement decisions for inference applications on heterogeneous hardware accelerators, and (iii) how to distribute heterogeneous inference requests (*e.g.*, with different batch sizes) to applications with heterogeneous hardware accelerators. The **major contributions** are as follows.

(i) We propose SMIF (short for SLA-driven ML inference framework), a heterogeneity-aware serverless framework for ML inference services on heterogeneous infrastructure to address those challenges. The core component of our framework is the intelligent scheduler that, firstly, leverages the knowledge of heterogeneous GPUs (*e.g.*, GPU compute capability, memory, and NVIDIA Multi-Process Service (MPS) capability) and specific information of ML inference applications (*e.g.*, inference-specific metrics such as inference request per second, request execution time, and request type such as batch size). Secondly, unlike existing serverless frameworks that assume the homogeneous functions running on homogeneous infrastructure, our framework takes function and compute heterogeneity into account. Finally, our framework manages the homogeneous functions as a group. When our framework needs to make orchestration decisions (*e.g.*, autoscaling, workload placement, and incoming request distribution), it leverages a global view by aggregating information from heterogeneous groups.

(ii) Our system allows users to specify the application SLA (*e.g.*, target latency and inference per second) and specific application contexts (*e.g.*, application metrics and inference interface or protocol) in the function description. Furthermore, based on the function description, our framework provides a generic monitoring system to collect both application-specific metrics exposed by the ML inference applications (if available) and application-agnostic metrics from the underlying platform (*e.g.*, Kubernetes and Knative). The autoscaler component of our framework configures the

[2]In our framework, each function runs on one container, and each application with multiple functions runs on one or multiple containers. In the remainder of the paper, we use the terms “function” and “container” interchangeably

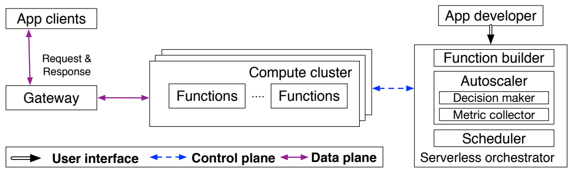


Figure 1. Serverless framework architecture.

target SLA from the application description and classifies functions into multiple groups based on the assigned hardware types. With the metric information collected from different groups and the application SLA, the autoscaler makes decisions to scale up/down the application on heterogeneous hardware and distribute inference requests accordingly.

(iii) In addition, our framework leverages the service mesh (*e.g.*, Istio (ist, a)) to distribute inference requests based on the collected metric information and the inference request type. More specifically, we support two inference workload routing policies: *hardware-aware* and *inference request-aware* for homogeneous and heterogeneous inference requests respectively in order to meet *computation* fairness unlike *inference request* fairness with round-robin routing policy.

(iv) To the best of our knowledge, this paper presents the first system with a GPU scheduler and device manager that enable heterogeneous GPU managements and application placement based on the compute capabilities of the GPU (*e.g.*, memory and MPS).

(v) We have built a prototype of our framework using Knative Serving (kna, a) and container orchestration platform Kubernetes (k8s, a). We create new components and extend existing components (*e.g.*, traffic manager, GPU scheduler and device manager, autoscaler) by leveraging the pluggable/extensible interfaces in Knative and Kubernetes.

(vi) We evaluate our prototype with TensorRT Inference Server (TRTIS) (trt, a) as an image recognition application with various DNN models on heterogeneous GPUs. To the best of our knowledge, our framework is the first holistic serverless framework designed for ML inference applications that leverage heterogeneity of inference requests and infrastructure.

2 Background

This section describes the background behind serverless frameworks and ML Inference services and the associated challenges in this area.

2.1 Serverless Frameworks

Figure 1 shows a generic serverless framework. It consists of (i) a *serverless orchestrator* which receives function descriptions from application developers, schedules the functions, and dynamically scales-up/down the deployed functions based on the status of workload in functions measured in the metric collector, (ii) *functions* which perform application-specific computation running on a cluster of compute hosts when they receive requests. When there are no requests for a certain period of time for a specific function, the *autoscaler* scales down the function to zero, which is called *zero scale*, and (iii) a *gateway* is a proxy between functions and application clients, which uniformly distributes traffic from application clients to multiple functions. In addition, *the gateway* holds the requests before the function is ready in case of *zero scale*, and sends a notification to the *autoscaler* to launch new functions for the requests.

Autosclaer is one of the critical components of our serveless framework, which decides to dynamically scale-up/down functions based on the current loads of running functions. Various metrics (*e.g.*, request per second (rps), the number of in-flight request (called *concurrency*), CPU utilization, etc.) with threshold-based autoscaling with sliding windows algorithm are used to make the auto-scaling decisions. For example, with *concurrency*, the autoscaler evaluates the windows and decides scale-up/down the number of replicas (*i.e.*, the number of replicas $= \frac{\text{concurrent requests}}{\text{specified threshold}}$).

To quickly react to dynamics of requests (*e.g.*, bursty request), the autoscaler operates on two modes (*i.e.*, *stable* and *panic* modes) based on two long and short sliding windows. In *stable* mode, the autoscaler calculates the average concurrency over a long time window (*e.g.*, 60s) to evaluate stable scale-up/down decisions. However, since the *stable* mode is not responsive to handle bursty and sudden requests, the auto-scaling also operates short time windows (*e.g.*, 6s) called *panic* mode. The auto-scaling mode transitions from *stable* to *panic* when the average window reaches two times more than the specified concurrency threshold from users.

2.2 Deploying ML Inference Services

ML inference services with trained ML models usually run on ML accelerators (*e.g.*, GPU, FPGA, TPU, etc.) due to their intensive computation requirements and the requirements of inference services (*e.g.*, low latency). So, before deploying the trained models with inference services on ML accelerators, the trained ML models are optimized for specific ML accelerators to achieve high performance based on their hardware characteristics (*e.g.*, computational capabilities). Manual and automated approaches are currently used for ML model optimizations. In a manual optimization approach, each accelerator vendor provides specific libraries optimized by human experts who understand the hardware-specific knowledge well. (*e.g.*, TensorRT (ten), cuDNN (nvi, a) for NVIDIA GPUs and Intel Math Kernel Library (MKL) (int) for Intel CPUs). The automated performance optimization (Chen et al., 2018; Liu et al., 2019; Vasilache et al., 2018) finds optimized low-level implementations for specific hardware without the human experts in a vendor-agnostic way. While the two approaches to optimize the ML model are different, they generate various optimized models based on accelerators, precision, and batch size from the same trained model. The optimized models show different performance and resource requirements (*e.g.*, GPU memory) based on accelerators, precision, and batch size. We show the detailed analysis in Section 3.

2.3 GPU Sharing Strategies

The NVIDIA Multi-Process Service (MPS) (cud, b) and Multi-Instance GPU (MIG) (cud, a) are designed to enable co-operative multi-process CUDA applications to utilize the latest NVIDIA GPUs. Without MPS, only one process can use the GPU at a given time (i.e., time multiplexing among multiple processes), and GPU resources can be underutilized. To address this problem, NVIDIA proposes GPU sharing features (*e.g.*, MPS and MIG) to enable multi processes to use the Hyper-Q capability on NVIDIA GPUs. Hyper-Q can simultaneously process the CUDA kernel of NVIDIA GPUs on the same GPU. Since GPU compute capacity is usually underutilized by a single application process, MPS can improve resource efficiency. The NVIDIA Volta GPU architecture introduces several new MPS capabilities (cud, b): 1) Volta MPS clients can submit tasks directly to the GPU without passing through the MPS server, 2) Each Volta MPS client owns its own GPU memory address space instead of sharing GPU memory address space with other MPS clients, and 3) Volta MPS supports limited execution resource provisioning for Quality of Service (QoS). However, existing container platforms (*e.g.*, Kubernetes) only support exclusive GPU allocation that assigns the entire GPU to one application (kub, b) or a time multiplexing approach to share GPUs (Dee; Ali) among multiple applications. This may cause resource inefficiency and performance interference.

3 MOTIVATION

This section motivates the need for heterogeneity-aware scheduling of ML inference applications by benchmarking the performance of TensorRT Inference Server (TRTIS) (trt, a) with several DNN models (*e.g.*, InceptionV3, MobileNetV1, MobileNetV2, and ResNetV2) on different GPUs (*e.g.*, NVIDIA Tesla V100, P100, and K40). This work focuses on three types of heterogeneity: GPU compute capability, inference request, and GPU resource allocation.

Heterogeneous GPU Compute Capability

Different GPU architectures provide different computational capacities and performance for each application. One of the limitations of current serverless frameworks is the homogeneity assumption that does not consider each cluster

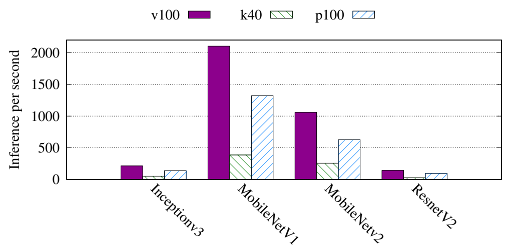


Figure 2. DNN inference performance.

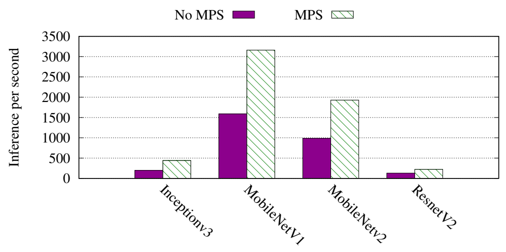


Figure 3. Shared GPU performance.

Batch Size	Throughput (inf/s)	Response Time (ms)
1	68.6	168
16	164.267	1,226
64	170.667	4,242
128	183.467	4,598

Table 1. Throughput and response time of different batch sizes.

node's different computational capacity and hardware features. The workload is usually distributed uniformly across all cluster nodes, while we show that this assumption leads to performance degradation. This section shows how compute capability differs among different GPU models.

We conduct experiments to measure inference performance, requests per second, and GPU resource usage (i.e., GPU memory) of different ML models running on different GPUs with batch size 1. Figure 2 shows that the inference performance varies significantly when running the same ML model on different GPUs. Therefore, when deploying ML inference services as serverless functions on different GPUs, the GPU heterogeneity needs to be considered to meet the target SLA (*e.g.*, throughput and latency). For example, if a less powerful GPU meets the application's target SLA, the serverless framework should run functions on that GPU to save cost.

Heterogeneous Inference Requests

Existing serverless frameworks mainly use simple metrics such as query per second (QPS) or CPU utilization to make autoscaling decisions (*e.g.*, scale up/down functions) (aut, a;c;b). *However, not all requests are the same.* To understand the impact of the inference requests with different batch sizes (*i.e.*, number of images in the same request) and how existing autoscalers react to such heterogeneous inference requests, we generate multiple consecutive inference requests in 15 seconds with different batch sizes.

As shown in Table 1, a larger batch size yields higher throughput (inferences per second) and higher response time. It is a common practice to "batch" multiple images in the same inference request to improve the overall throughput of inference services and reduce the average processing time of each image (Kochura et al., 2019; Kosaian et al., 2021). However, the end-to-end response time is also increased because the server has to complete the inference tasks for all images in the same batch before sending the response back to the client. Moreover, since the default Knative autoscaler cannot differentiate the inference requests with batch sizes, it fails to scale the functions to process inference requests with larger batch sizes. Since one of the critical features of serverless computing platforms is to meet the performance requirement of each application, the autoscaler needs to consider application-specific metrics such as batch sizes to make autoscaling decisions.

We get similar results for utilization-based autoscaler. Since performing ML inference (especially for DNN models) mainly runs on accelerators (*e.g.*, GPUs), the CPU utilization cannot correctly capture the accurate status of ML inference applications. Thus, the default autoscaler fails to scale up due to low CPU utilization even though more replicas are expected to achieve the target SLA.

GPU Sharing and Allocation

With NVIDIA MPS and MIG technologies, a GPU accelerator can be separated and shared by multiple ML inference applications at the same time without introducing performance interference. Figure 3 shows the throughput of the 8 ML inference applications sharing the same NVIDIA Tesla V100 GPU. We compare the performance of different DNN models with and without MPS enabled. With MPS enabled, the inference applications of all ML models show significantly better performance because the resource isolation of MPS can efficiently prevent performance interference among the applications sharing the same V100 GPU. This technology is also used in other research work for ML model training and hyperparameter tunning (Yu et al., 2021). With GPU sharing technologies (*e.g.*, NVIDIA MPS and MIG), we can achieve more fine-grained GPU resource allocation to achieve better GPU resource efficiency for cloud/infrastructure providers and lower operation costs for service operators. However, existing container orchestration platforms and serverless frameworks (k8s, d; nuc; blu) are only able to assign dedicated GPU(s) to one container or function. Ideally, the serverless framework should be able to automatically 1) identify whether GPU sharing technologies are available on each cluster node and 2) allocate the appropriate amount of GPU resources based on the GPU compute capability and the target SLA to an ML inference application. This work focuses on enabling fine-grained GPU resource allocation through NVIDIA MPS for serverless

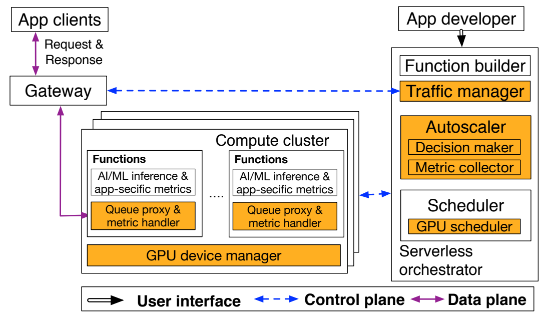


Figure 4. SLA-driven ML Inference Framework Architecture

applications.

4 Framework Design

4.1 Design Goals

We identify several key goals to design a heterogeneity-aware serverless platform for DNN inference service. First, the platform should offer a better abstraction for policy in Serverless to express the user's Service-Level Agreement (SLA). For example, required latency or inference per second is more intuitive than query per second and request per second for DNN inference service. Second, the platform should support various application-specific and agnostic metrics to make better orchestration decisions. Finally, the platform should orchestrate autoscaling and traffic management for the application to meet target SLA with the collected metrics and target SLA. Notably, the framework should leverage heterogeneous application requests (e.g., different batch sizes) and hardware capabilities (e.g., computation power, memory, sharing efficiency, etc.) to make intelligent workload distribution and deployment.

4.2 Architecture Overview & Workflow

Figure 4 shows an overview of the proposed serverless platform. Our framework has the same essential components described in a generic serverless framework. It introduces new components depicted in color to enable application and hardware-aware DNN inference services on heterogeneous clouds. Specifically, it has *GPU scheduler* and *GPU device manager* to schedule inference applications and manage the heterogeneous GPU devices. GPU device manager is deployed in every GPU server and is responsible for reporting the GPU information to the GPU scheduler and health check of GPU devices and specific GPU configurations (e.g., MPS). GPU scheduler stores the GPU information (e.g., GPU models, GPU memory, MPS capability, computation capability) and leverages the information while scheduling inference applications. While scheduling the inference application, the *queue-proxy* is deployed with every function and configured to collect application-specific/agnostic metrics. *Autoscaler* collects various and rich metric information from the queue-proxy and makes the decision for scale-up/down and workload distribution in autoscaler. Finally, *Traffic manager* configures the gateway for workload distribution based on hardware capabilities and inference requests.

Using the new components in our framework, the following describes a step-by-step procedure for deploying and managing inference applications. (i) *Submit application:* application developers submits the description of inference application with required SLA (e.g., target latency or inference per second) to Function builder. The Function builder converts the description to the platform-specific function description. (ii) *Schedule the application:* All scheduling requests in our framework are first forwarded to a default scheduler, and only scheduling requests for applications (e.g., inference application) requiring GPU are further forwarded to the GPU scheduler. Since the inference application requires GPU resources, GPU scheduler receives the function description and schedules the inference application on GPU. When the inference application is scheduled, one queue-proxy server is injected with the inference application. (iii) *Configure autosclaer and traffic manager:* the autoscaler and traffic manager get notification of launching inference application with its descriptions (e.g., target SLA and application-specific metrics). The autoscaler starts scraping metrics from the queue proxy server and makes scale-up/down decisions based on the collected metrics. The traffic manager configures the gateway to forward application requests to the inference application. (iv) a *Serving inference requests:* The deployed application is ready and starts serving the inference request, and the metric handler in a queue proxy collects application-specific metrics and application-agnostic metrics and exposes them to autoscaler. Queue-proxy scrapes the application-specific information provided by inference application if available, and the availability of application-specific information is based on the function description.

After the initial deployment, our framework manages the inference application to meet the required SLA. Specifically, the autoscaler makes scaling-up/down decisions and traffic distributions to functions based on collected metrics and the required SLA. The autoscaler sends requested information, such as application-specific and SLA metrics, to the scheduler and traffic manager if needed, and then the steps (ii) to (iv) are repeated.

4.3 Application-specific/agnostic metrics

Our framework collects application-specific/agnostic metrics since precise and rich metrics of deployed applications are important for autoscaling decisions. The queue proxy is responsible for forwarding the traffic and collecting metrics that are exposed to the autoscaler.

Since recent application developers advocate including logic to expose application-specific metrics via monitoring software (e.g., prometheus (pro)) for better observability and analysis (clo, a), application-specific metric information is collected in queue-proxy. For example, popular inference serving frameworks (e.g., TRTIS (trt, a), tensorflow serving (tfs, a)) support common metrics (e.g., request count, inference execution count, inference execution time, queue time, etc) (tri; tfs, c).

When the application exposes its specific metrics, the application-specific metrics information is provided by application developers in their application specification, and it is automatically set up in the queue-proxy. Then, the queue proxy collects its metrics through the local network interface. These application-specific metrics are precise and explicit since they capture the status of the application. For example, since many inference applications support various batch sizes in one inference request to get high throughput, shown in Figure 2, simply counting the number of inference requests does not capture a specific workload.

If the application does not expose its metrics, queue-proxy collects application-agnostic metrics. Since the queue-proxy is a layer seven proxy (e.g., supporting HTTP and GRPC), it can collect various application-agnostic information. It can collect the number of requests and the number of concurrent requests by counting inference requests (e.g., HTTP, GRPC request). Further, it can measure histogram of inference execution time by observing the time between sending a request to an application to receiving its corresponding response from the application and histogram *content-length* field in HTTP header, which indirectly exposes the number of images in one inference request (i.e., batch size).

4.4 Autoscaler

One of the core components of our framework is the *Autoscaler* which consists of a metric collector and decision-maker for scaling up/down functions based on the collected metrics and the target SLA of the application. The target SLA is set up in autoscaler when the application description is submitted. The critical design decision of autoscaler is to consider the same application running on the same GPUs (*homogeneous* autoscaling) and running on different GPUs (*heterogeneous* autoscaling).

The autoscaler manages the homogeneous application as a group. This decision provides multiple advantages with respect to metric collections, traffic distributions, and scaling decisions. While prior serverless frameworks use CPU utilization or query per second as a metric to scale up/down the number of pods (ope; k8s, a), our framework is SLA-driven and uses application-specific metrics to make such decisions.

4.4.1 Metric collector

Metric collector in the autoscaler scrapes metrics from only one of the queue-proxies deployed with the application in each group simultaneously. Since the inference requests are forwarded to multiple functions belonging to the same group in a uniform distribution fashion, the metric collector can estimate overall load by multiplying the metric from only one application in one group to the number of functions on the same accelerator. This approach helps to reduce the overhead of collecting metrics in autoscaler. In addition, the metric collector has a global view of load by scraping metrics from each group simultaneously.

4.4.2 Provision the inference applications

In our framework, we have three scaling decisions and workload distribution decisions.

• Homogeneous scale-up/down: scale-up/down the application on the same hardware.

• Heterogeneous scale-up/down : scale-up/down the application on the different hardware. It enables the application to gracefully scale down to zero using low-end GPUs or limiting available CUDA thread allocation with MPS.

• Workload distribution: the ratio of the inference request distribution to heterogeneous groups when the applications are deployed in heterogeneous GPUs.

The critical question here is how to make homogeneous or heterogeneous scale-up/down decisions. The decision is based on various application-specific/agnostic metrics (e.g., absolute inference execution time, queueing delay, etc.).

Workload distribution: An application can get deployed in heterogeneous GPUs based on the availability of GPUs when the GPUs can meet the SLA of the application. Since the workload distribution is initially uniform, each GPU approximately receives the same workload, but the execution time is different according to GPU computation power. The autoscaler can recognize relatively sluggish groups since the metric collector scrapes histogram of the execution time of inference requests and queueing delay in each group. Based on the information, autoscaler dynamically adjusts the workload distribution ratio and sends the information to the traffic manager.

4.5 Traffic manager

The traffic manager is responsible for managing inference workload distribution by controlling the gateway in our framework. It supports two inference workload routing policies: *hardware-aware* and *inference request-aware* routing. The hardware-aware routing policy is based on the information from the service rate monitoring module that periodically specifies each application's service rate on each node. It adjusts the workload distribution ratio on heterogeneous groups. The traffic manager updates traffic weight rules for each group in the gateway if the inference request is homogeneous (i.e., the batch size is uniform).

When the inference request is heterogeneous (i.e., each inference request has a different batch size. In other words, one inference request has a different number of images to be inferred), the heterogeneous inference request is distributed to different GPUs based on their computation capabilities. In this case, the traffic manager updates the gateway to apply inference request-aware routing policy. Basically, inference request-aware routing policy decides the destination of functions based on a specific field (e.g., batch size) in inference request if available or *content-length* field which approximately estimates batch size in request header (e.g., HTTP or GRPC header). As batching request has performance advantage (i.e., increase throughput), popular inference serving frameworks (e.g., triton (trt, a), tensorflow serving (tfs, a)) support the batch requests with specific header field (trt, b; tfs, b). Since the batch size is an application-specific request header field, the application developers should provide this information when submitting the application description. The traffic manager notifies this header information and configures it to gateway using StringMatch capability (ist, b) based on GPU computation capability. Assuming D/D/1 queues, we use the following equilibrium property to distribute the traffic on different nodes:

$$\frac{\mu_1}{\lambda_1} = \frac{\mu_2}{\lambda_2} = ... = \frac{\mu_N}{\lambda_N} \tag{1}$$

Where, μ_i is the service rate of node i in the cluster, and λ_i is the arrival rate of the job that should be distributed to node i in the cluster. Intuitively, this equilibrium property says that if the service rate of a job on node i is X times the service rate of the job on node j, then we need to distribute X times more requests to node i than node j. We currently rely on the developer's input or *content-length* fields in the inference request header for inference request-aware routing policy. Since there are extensive efforts for standardizing inference request (e.g., Predict Protocol in KFServing (kfs, b) and CloudEvents (clo, b)), the request-aware routing policy in our framework can be applied to all inference applications following the generic request standard.

4.6 GPU Scheduler and Device manager

We introduce two components to manage GPU resources and assign them to inference applications.

4.6.1 GPU Device manager

GPU device manager in our framework is deployed in every GPU node and responsible for reporting GPU hardware specifications (e.g., GPU models, GPU memory, MPS capability, computation capability) to our GPU scheduler, health check of GPU and NVIDIA MPS daemon management if GPU supports MPS. In addition, at runtime, it inserts GPU-related environment variables (e.g., CUDA_VISIBLE_DEVICES or CUDA_MPS_ACTIVE_THREAD_PERCENTAGE) for the application before deploying the application container on the GPU node.

Annotations	Description
`latency`	Target latency
`user-metrics`	Enable user-specific metrics
`initialDesiredScale`	The number of initial functions
`volume-mounts`	Persistent volume name
`gpu-model`	GPU model (e.g., k40, v100)
`dnn-model`	DNN model name (inceptionv3)
`dnn-quantization`	DNN quantization (fp32, fp16)
`infer-batch-size`	Inference batch size
`gpu-mem-limit`	GPU Memory
`mps-on`	MPS enabler
`gpu-core-percentage`	GPU core percentage
`gpu-count`	The number of GPUs

Table 2. Used Annotations.

4.6.2 GPU-aware Scheduler

Our GPU scheduler takes care of the placement of inference applications submitted from users or autoscaler on GPUs. GPU scheduler gets notifications of detailed GPU information from GPU device manager whenever new GPU servers are added or removed and maintains the heterogeneous GPUs. It continuously keeps track of available GPU resources (e.g., GPU memory, thread percentage on GPU) and GPU capability (e.g., MPS) while assigning and releasing the GPU resource to and from the application. The scheduler in our framework does not allow GPU memory overcommitment since applications using GPU can easily crash when they do not have enough GPU memory.

Our GPU scheduler takes a best-fit scheduling policy that looks at the number of logical GPUs requested by the application and assigns the physical GPU with the smallest free partition that meets the requirements.

5 IMPLEMENTATION

We implemented the SMIF architecture using KNative Serving (kna, a) which are built on service mesh (e.g., Istio (ist, a)) and container orchestration platform Kubernetes (k8s, a). Since Knative Serving is built on Istio and Kubernetes, it already provides critical components (e.g., autoscaler, metric collector, queue-proxy), and programming APIs to develop and deploy serverless applications. Especially we leverage well-defined pluggable/extensible interfaces or frameworks for our extensions (e.g., traffic manager, GPU scheduler and device manager, autoscaler) in Kubernetes, which allows us to replace or extend them relatively quickly.

5.1 System Implementation

Job description: We enable several parameters explained in Table 2 to specify the job description. These parameters are used in several components (e.g., target latency in autoscaler, GPU model information in GPU scheduler, DNN model information in webhook, etc.).

Application metric collection: In KNative serving, one queue-proxy container is injected as a sidecar container when the serverless application starts. This queue-proxy collects various metrics and exposes them to the autoscaler using Prometheus server (pro). We extend the queue-proxy to collect application-specific metrics (e.g., inference request count, inference execution count, inference request duration, inference computation duration, and inference queue duration) offered by the prometheus metric server in the application and application-agnostic metrics (e.g., HTTP request count, concurrent request count, and latency histogram, etc). The application-agnostic metrics are collected whenever an HTTP request event happens, and the application-specific metrics are collected every second. With the configuration of queue-proxy of the application, queue-proxy decides to collect the application-specific metrics if the application has a Prometheus server.

Autoscaler: We implement our autoscaler by refactoring autoscaler in Knative Serving. Our autoscaler collects the application-specific and agnostic metrics exposed by the queue-proxy every second. Based on these metrics, our autoscaler leverages application-specific information and application-agnostic latency histogram by comparing latency specified by users to make the decision to scale-up/down the application instead of Query Per Second (QPS), Request Per Second (RPS) in Knative Serving, The autoscaler updates its scaling decision to GPU scheduler. The autoscaling decisions are written as annotations in the job description, used in our GPU scheduler.

GPU scheduler: We implement GPU scheduler by using scheduler extender (k8s, e), which is Kubernetes scheduler plugin interface. Based on the decision in our autoscaler, the GPU scheduler decides GPU nodes to deploy the application in case of scale-up events. We also developed MPS capability to provide GPU sharing for the GPUs that provide MPS capability (V100 and T4 in our testbed).

GPU device manager: We implement GPU device manager using device plugin framework (k8s, c) in Kubernetes and deploy it as Kubernetes DaemonSet (k8s, b). Thus, it is deployed in every GPU node gets GPU hardware specifications (e.g., GPU models, GPU memory, MPS capability, computation capability) by using NVIDIA Management Library (NVML) (nvi, b) and reports them to our GPU scheduler. The GPU device manager monitors the health of GPUs and failure events from GPU and various OS signal events (e.g., SIGINT, SIGTERM, etc.). When failure happens, it reports the events to our GPU scheduler. For supporting MPS, we leverage NVIDIA MPS daemon docker container (cud, c) and if an inference application is scheduled on the MPS-enabled GPU, *hostIPC* and *hostPath* fields in the application description are added to enable the application to talk to the MPS daemon running on the MPS daemon docker.

Traffic manager: To configure inference request routing in *Gateway* which is envoy proxy (env), we leverage Istio VirtualService (ist, c). VirtualService is an abstraction to express routing rules with match and route specifications (e.g., different traffic routing ratios to various applications and HTTP header matching-based routing). These routing rules are applied to Gateway. Our Traffic manager controls the Gateway by leveraging VirtualService. It supports proportional traffic distribution based on the computation capability of GPUs when the applications are deployed in different GPU models. It expresses the ratio with *Route* (kna, b) resource in KNative, which is a higher abstraction of VirtualService. In addition to proportional traffic distribution, the Traffic manager enables inference request-aware routing based on HTTP headers (e.g., batch size or content-length in HTTP headers) with StringMatch (ist, b). Based on the GPU computation capability, it forwards inference requests with larger batch size or large content length to the application running on high-end GPU.

Webhook: Since KNative serving supports limited Kubernetes capability (e.g., volumes, GPU resource expression), to fully utilize this capability in our framework, we implement mutating admission webhook (Kubernetes documentation) Kubernetes plugin, which allows modification of job description before scheduling the application. In our implementation, webhook modifies the job description to support volumes, GPU resource expression based on defined parameters shown in Table 2.

Model repository: Since the size of the trained DNN model is large and the inference server requires a specific trained DNN model according to the GPU model, its quantization type (e.g., fp32, fp16, int8) and batch size, we cannot put them into a docker container. Instead, we leverage Network File System (NFS) to store various DNN model files stored according to the GPU model, its quantization type, and batch size in the NFS. The required DNN model is loaded from the NFS when an inference application starts. We leverage volumes and persistent volumes capability in Kubernetes. Thus, the inference application does not need to be modified or specify how DNN model should be loaded.

Conatiner registry: We set up a local registry in our testbed to pull container images of inference applications to avoid unpredictable delays (e.g., network delay).

Workload generator: We deploy TensorRT Inference Server

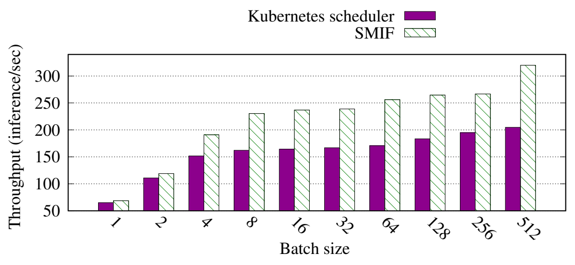
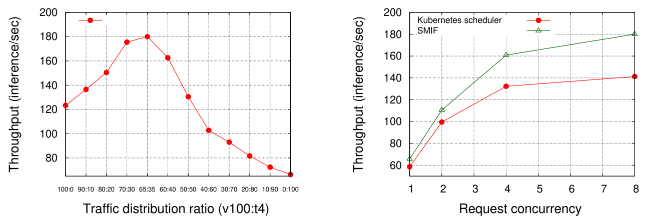


Figure 5. Throughput as we increase the batch size of inference application on the Kubernetes scheduler and SMIF.



a. Traffic Distribution. b. Comparison.

Figure 6. (a) Throughput of SMIF for different traffic distribution ratios, and (b) Comparison of SMIF with respect to the default Kubernetes scheduler.

(TRTIS) (trt, a) on our framework to serve various ImageNet inference requests. Since we do not require application modifications, we can easily deploy the original TRTIS on our implementation.

6 Evaluation

To validate our proposed framework, we have implemented its prototype based on the Kubernetes container orchestrator platform (k8s, a) and service mesh (ist, a). We evaluate our implementation on a heterogeneous cluster consisting of two GPUs (V100 and T4) with MPS capability on the AWS cluster. We use m5.large for the master node and a p3.2xlarge for the V100 worker GPU and a g4dn.2xlarge for the T4 worker GPU in the cluster. We compare the performance with respect to the default Kubernetes scheduler for two scenarios: (i) workload-aware heterogeneity and (ii) hardware-aware heterogeneity. We used TRTIS to serve various ImageNet DNN inference requests and used perf_client (per) as an inference request generator to send inference requests to TRTIS.

6.1 Workload-aware heterogeneity

While batch processing improves the average performance of each inference image, current serverless platforms are not application-aware and process a request with one image the same as a request with a batch size of 512. Figure 5, shows the throughput of the inference application as we increase the batch size for the performance comparison with baseline (Kubernetes scheduler). While the baseline uniformly distributes the traffic across different nodes and does not increase the number of replicas for larger batch sizes, our implementation gets application specific metrics and distributes the load according to the batch sizes and scales up the number of containers when the batch size is larger.

6.2 Hardware-aware heterogeneity

In this set of experiments, we evaluate the performance of our workload distributor, that is, application and hardware-aware, with respect to the Kubernetes baseline workload distributor. Figure 6-a shows the experimental results for different traffic distribution ratios on a three-node cluster consisting of a T4 and a V100 worker node. We run the trtis inference application on the two accelerators (V100, T4) that provide MPS capability. The default Kubernetes scheduler distributes the traffic uniformly on the two GPU nodes (50:50). However, our implementation has a monitoring module that continuously monitors the service rate of each node and schedules the traffic according to each node's service rate. In this example, the monitoring module observes that the optimal workload distribution ratio between the V100 and T4 GPU is when $x/y = 1.85$, where x is the percentage of the requests sent to the V100 GPU and y is the percentage of the requests sent to T4 GPU. Therefore, the optimal scheduler is when $x = 65, y = 35$ as shown in Figure 6. As shown, in our implementation, throughput improves from 130.5 to 180 inference/sec (38% improvement).

In the next set of experiments, we compare our implementation with the default Kubernetes scheduler. We run the TRTIS inference application with the resenet-v2-152 model, run it on the default Kubernetes scheduler that is not hardware-aware, and compare the throughput with our implementation. Figure 6-b shows the comparison between the two cases as we increase the number of concurrent requests. As shown the performance gap between our approach and the baseline increases as we increase the concurrency level.

6.3 GPU Sharing Capability

Kubernetes supports experimental AMD and NVIDIA GPU management (k8s, d). However, they do not allow sharing one GPU with multiple applications. Deepomatic (Dee) enables sharing one GPU with multiple applications on Kubernetes without any performance isolation. However, none of these approaches consider GPU heterogeneity in terms of computation capability or leverage the heterogeneity for workload distribution.

To evaluate our system, we compared our system against ex-

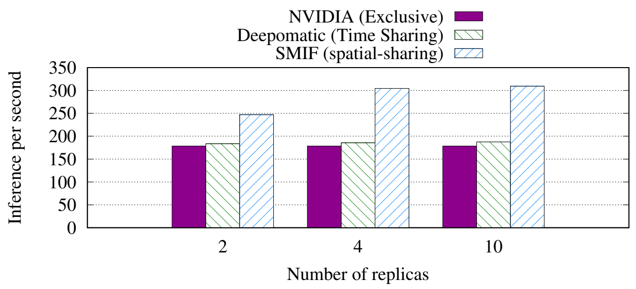


Figure 7. Throughput comparison between SMIF, Nvidia exclusive and Deepomatic frameworks.

perimental NVIDIA GPU management (k8s, d) and Deepomatic (Dee) which support exclusive and time-multiplexing GPU assignment respectively on Kubernetes. For the workloads, we deploy TRTIS to serve ImageNet DNN inference requests as Kubernetes Deployment and change *replica* (i.e., 2, 4, and 10) field to evaluate GPU sharing impact except for NVIDIA GPU management since it only allows exclusive GPU assignment to one Pod. For the inference request generator, we used perf_client tool (per) with 100 concurrent requests for 15 seconds. Figure 7 shows throughput (i.e., inference per second). As shown, there is 1.65× and 1.73× improvement compared to Deepomatic and Nvidia exclusive respectively when the number of replicas is ten since it enables spatial sharing approach by using logical GPU abstractions with GPU scheduler and device manager. While time-multiplexing also leverages parallelism with multiple replicas, it shows similar performance with exclusive assignment case due to its GPU resource contention.

7 Related Work

This section describes serverless computing frameworks, DNN inference systems, and GPU sharing efforts.

Serverless frameworks for DNN applications: In (Ishakian et al., 2018), the authors evaluate the performance of serving deep learning models on a serverless framework. In particular, they use the MxNet deep learning framework on AWS Lambda and show that while warm serverless function executions are within an acceptable latency range, cold start latency can cause SLA violations. A cluster-level centralized and core-granular scheduler for serverless functions is introduced in (Kaffes et al., 2019). In (Suresh & Gandhi, 2019) a function-level scheduler is proposed to minimize provider resource costs while meeting customer performance requirements. The proposed approach dynamically regulates the CPU shares of colocated functions at run time to mitigate resource contention. Kubeflow (kub, a) makes deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. KFserving (kfs, a) is a Serverless Inferencing framework on Kubernetes on top of Knative and Kubeflow. While both frameworks make deployments of ML applications simple, they do not have fine-grained GPU management (e.g., sharing GPUs for multiple containers) and traffic management. In addition, since KFServing relies on KNative functionality, it only supports QPS-based autoscaling. Existing container management and serverless frameworks (k8s, d; nuc; blu) also do not allow GPU sharing for multiple containers and functions. Barista (Bhattacharjee et al., 2019b) proposes a serverless system for machine learning workload. Spock (Gunasekaran et al., 2019) leverages serverless functions in combination with VM-based hosting to provide SLO guarantee at a given cost budget. The authors use cost predictions for serverless functions to make cost-aware decisions between the VM hosting and serverless functions.

DNN inference system: Yadwadkar et al. (Yadwadkar et al., 2019) propose a managed and model-less inference serving that addresses several challenges, including the heterogeneity for both hardware and models, designing user interfaces, and building SLO-driven systems. In (Romero et al., 2019) the authors propose INFaaS, which is a model-less inference-as-a-service system that allows users to define inference tasks and performance/accuracy requirements for queries, leaving it to the system to determine the model-variant, hardware, and scaling configuration.

GPU sharing for ML Inference: Recent works (Jain et al., 2018; Romero et al., 2019; Jain et al., 2019) have shown that sharing GPU resources for inference help to improve GPU utilization. There exists a large body of work to reduce deep learning inference latency (Crankshaw et al., 2017; Dakkak et al., 2019). Clipper proposes to combine multiple concurrent DL requests into batches to better utilize the GPU with higher latency costs (Crankshaw et al., 2017). Trims show that model loading is the primary source of *cold start* latency and propose to move the bottleneck of deep learning model inference to compute to mitigate the problem (Dakkak et al., 2019). However, all of these techniques suffer from their inability to model heterogeneity in the hardware architecture and applications' resource requirements. Various CPU/GPU sharing and virtualization techniques have been proposed to improve system throughput, and utilization by parallel sharing of CPU and GPU or time-sharing (Sengupta et al., 2013; Yeh et al., 2017). However, these models do not consider heterogeneous GPU/CPU models.

8 Conclusion

This paper proposes a novel application and hardware-aware Serverless computing framework. We design an intelligent traffic manager and autoscaling strategy that leverage the cluster's application-specific metric and heterogeneous compute capability to meet user defined application SLAs. Our experimental results from our prototype deployment show that our traffic distributor module can improve throughput by 38% compared to the default Kubernetes traffic distributor. Also, our proposed logical GPU abstractions enable spatial sharing of GPUs with GPU scheduler and device manager. Experimental results show 1.65× and 1.73× throughput improvement compared to Deepomatic and Nvidia exclusive, respectively.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their feedback on earlier drafts of this paper. We wish to thank Eric Wu in Hewlett Packard Labs for his support in setting up the testbed.

REFERENCES

Alibaba., GPU Sharing Device Plugin in Kubernetes. https://github.com/AliyunContainerService/gpushare-device-plugin.

Deepomatic., Support for shared GPUs by declaring GPUs multiple times. https://github.com/Deepomatic/shared-gpu-nvidia-k8s-device-plugin.

Autoscaling in Knative Serving. https://github.com/knative/serving/blob/master/docs/scaling/DEVELOPMENT.md, a.

kubeless. https://github.com/kubeless/kubeless/blob/master/docs/autoscaling.md, b.

Auto-scaling in OPENFAAS. https://docs.openfaas.com/architecture/autoscaling/, c.

Enabling a Cloud-Like Experience for On-Premises GPU Infrastructure. https://www.bluedata.com/blog/2019.

Observability and Analysis - Monitoring (62) in CNCF Cloud Native Interactive Landscape. https://landscape.cncf.io/category=observability-and-analysis&format=card-mode&grouping=category, a.

cloudevents: A specification for describing event data in a common way. "https://cloudevents.io/", b.

NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, a.

NVIDIA CUDA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, b.

MULTI-PROCESS SERVICE (experimental). https://github.com/NVIDIA/nvidia-docker/wiki/MPS-(EXPERIMENTAL), c.

Envoy. https://www.envoyproxy.io/.

Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

Istio. https://istio.io/, a.

StringMatch with Virtual Service. https://istio.io/latest/docs/reference/config/networking/virtual-service/#StringMatch, b.

Virtual Service. https://istio.io/latest/docs/reference/config/networking/virtual-service/, c.

Kubernetes. https://kubernetes.io/, a.

DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, b.

Device Plugins. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/, c.

Schedule GPUs. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/, d.

Scheduler extender. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md, e.

KFServing. Serverless Inferencing on Kubernetes. https://github.com/kubeflow/kfserving, a.

KFServing. Predict Protocol - Version 2. https://github.com/kubeflow/kfserving/tree/master/docs/predict-api/v2, b.

Knative Serving. https://github.com/knative/serving, a.

Route in Knative. https://knative.dev/docs/serving/spec/knative-api-specification-1.0/#route, b.

Kubeflow. The Machine Learning Toolkit for Kubernetes. "https://www.kubeflow.org/", a.

Kubernetes., Schedule GPUs. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/, b.

nuclio: Automate the Data Science Pipeline with Serverless Functions. https://nuclio.io/.

NVIDIA cuDNN. https://developer.nvidia.com/cudnn, a.

NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml, b.

OPENFAAS. https://www.openfaas.com/.

NVIDIA triton inference server tool. https://docs.nvidia.com/deeplearning/triton-inference-server/master-user-guide/docs/perf_client.html.

prometheus. https://prometheus.io/.

NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

Export server metrics in TensorFlow Serving. https://github.com/tensorflow/serving/issues/462, a.

Batch size in TensorFlow Serving. https://www.tensorflow.org/tfx/serving/performance, b.

Serving Models. https://www.tensorflow.org/tfx/guide/serving, c.

NVIDIA TensorRT Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/master-user-guide/docs/metrics.html.

NVIDIA TensorRT Inference Server. https://docs.nvidia.com/deeplearning/sdk/tensorrt-inference-server-guide/docs/, a.

Client Examples in Triton. https://docs.nvidia.com/deeplearning/triton-inference-server/master-user-guide/docs/client_example.html#section-simple-examples, b.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pp. 1–20. Springer, 2017.

Bhattacharjee, A., Barve, Y., Khare, S., Bao, S., Gokhale, A., and Damiano, T. Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pp. 59–61, 2019a.

Bhattacharjee, A., Chhokra, A. D., Kang, Z., Sun, H., Gokhale, A., and Karsai, G. Barista: Efficient and scalable serverless serving system for deep learning prediction services. *arXiv preprint arXiv:1904.01576*, 2019b.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 613–627, 2017.

Dakkak, A., Li, C., De Gonzalo, S. G., Xiong, J., and Hwu, W.-m. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *12th International Conference on Cloud Computing (CLOUD)*, pp. 372–382. IEEE, 2019.

Feng, L., Kudva, P., Da Silva, D., and Hu, J. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 334–341. IEEE, 2018.

Gunasekaran, J., Thinakaran, P., Kandemir, M. T., Urgaonkar, B., Kesidis, G., and Das, C. R. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud, 2019.

Ishakian, V., Muthusamy, V., and Slominski, A. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 257–262. IEEE, 2018.

Jain, P., Mo, X., Jain, A., Subbaraj, H., Durrani, R. S., Tumanov, A., Gonzalez, J., and Stoica, I. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

Jain, P., Mo, X., Jain, A., Tumanov, A., Gonzalez, J. E., and Stoica, I. The ooo vliw jit compiler for gpu inference. *arXiv preprint arXiv:1901.10008*, 2019.

Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 158–164, 2019.

Kochura, Y., Gordienko, Y., Taran, V., Gordienko, N., Rokovyi, A., Alienin, O., and Stirenko, S. Batch size influence on performance of graphic and tensor processing units during training and inference phases. In *International Conference on Computer Science, Engineering and Education Applications*, pp. 658–668. Springer, 2019.

Kosaian, J., Phanishayee, A., Philipose, M., Dey, D., and Vinayak, R. Boosting the throughput and accelerator utilization of specialized cnn inference beyond increasing batch size. In *International Conference on Machine Learning*, pp. 5731–5741. PMLR, 2021.

Kubernetes documentation. Dynamic Admission Control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing {CNN} model inference on cpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 1025–1040, 2019.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. Infaas: Managed & model-less inference serving. *arXiv preprint arXiv:1905.13348*, 2019.

Sengupta, D., Belapure, R., and Schwan, K. Multi-tenancy on gpgpu-based servers. In *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pp. 3–10, 2013.

Suresh, A. and Gandhi, A. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, pp. 19–24, 2019.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Wang, H., Niu, D., and Li, B. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 1288–1296. IEEE, 2019.

Yadwadkar, N. J., Romero, F., Li, Q., and Kozyrakis, C. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 184–191, 2019.

Yan, M., Castro, P., Cheng, P., and Ishakian, V. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, pp. 1–4, 2016.

Yeh, T. T., Sabne, A., Sakdhnagool, P., Eigenmann, R., and Rogers, T. G. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. *ACM SIGPLAN Notices*, 52(8):221–234, 2017.

Yu, P., Liu, J., and Chowdhury, M. Fluid: Resource-aware hyperparameter tuning engine. *Proceedings of Machine Learning and Systems*, 3, 2021.