GPU-enabled Functional-as-a-Service

by

Sungho Hong

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2022 by the
Graduate Supervisory Committee:

Ming Zhao, Chair
Zhichao Cao
Mohamed Sarwat

ARIZONA STATE UNIVERSITY

August 2022

ABSTRACT

Function-as-a-Service (FaaS) is emerging as an important cloud computing service model as it can improve scalability and usability for a wide range of applications, especially Machine-Learning (ML) inference tasks that require scalable computation resources and complicated configurations. Many applications, including ML inference, rely on Graphics-Processing-Unit (GPU) to achieve high performance; however, support for GPUs is currently lacking in existing FaaS solutions. The unique event-triggered and short-lived nature of functions poses new challenges to enabling GPUs on FaaS which must consider the overhead of transferring data (e.g., ML model parameters and inputs/outputs) between GPU and host memory.

This thesis presents a new GPU-enabled FaaS solution that enables functions to efficiently utilize GPUs to accelerate computations such as model inference. First, the work extends existing open-source FaaS frameworks such as OpenFaaS to support the scheduling and execution of functions across GPUs in a FaaS cluster. Second, it provides caching of ML models in GPU memory to improve the performance of model inference functions and global management of GPU memories to improve the cache utilization. Third, it offers co-designed GPU function scheduling and cache management to optimize the performance of ML inference functions. Specifically, the thesis proposes locality-aware scheduling which maximizes the utilization of both GPU memory for cache hits and GPU cores for parallel processing.

A thorough evaluation based on real-world traces and ML models shows that the proposed GPU-enabled FaaS works well for ML inference tasks, and the proposed locality-aware scheduler achieves a speedup of 34x compared to the default, load-balancing only scheduler.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

The popularity of Function-as-a-Service (FaaS) increases as there is a growing demand for building websites, real-time file processing, and machine learning (ML) inferences without worrying about scalability and resource management Jonas et al. 2019. However, running ML inference with FaaS functions is limited as the major public cloud providers do not provide or directly provide FaaS platforms to access GPU resources. For example, AWS Lambda Chand 2021, a popular FaaS from Amazon, does not provide GPU to FaaS, while Azure functions Garg 2020, FaaS from Microsoft, can indirectly access GPU via GPU-enabled Kubernetes. However, the support for sharing GPU in the Azure function is limited as Kubernetes avoids addressing the challenges of GPU resource management by preventing multiple pods from sharing a single GPU. Therefore, GPU-enabled-FaaS is a growing research area where both industry and the research community work to uncover and address the unique challenges of managing GPU resources in the FaaS platform.

Applying GPUs to the FaaS platform is imperative as ML inference running on FaaS requires low latency to meet the Service-Level-Agreement (SLA). The survey of FaaS Scheuner and Leitner 2020 indicates the increasing deployment of ML inference on FaaS platform. ML inference applications in production have stringent latency requirements; for example, providing auto-suggestions in the search bar requires returning the inference results in real-time while users browse for keywords Garg 2020. Using GPU for running ML inference can significantly reduce the latency when input data can be grouped into a large batch and models are designed for parallel

computation. The larger batch size allows ML inference to further benefit from GPU acceleration Gunny 2019 because a bigger batch gives the GPU a higher volume of work to run in parallel. ML inference models such as Transformers Wolf et al. 2020 translate the sequential computation of recurrent neural networks (RNN) into the independent calculation to benefit from GPU parallelization.

Managing the GPU resources in the FaaS platform is challenging as sharing GPUs differs from sharing conventional resources such as CPU and memory. The GPU is designed to maximize a single application's throughput performance by allocating the entire resource to a single GPU process. Since the single process has full access to the GPU resource, the GPU expects the application to be programmed to avoid exceeding the available GPU memory. Unless the GPU is configured with vendor-specific features such as Nvidia Unified Memory Li et al. 2015, the applications using the GPU are constantly vulnerable to an out-of-memory (OOM) error Landaverde et al. 2014. The limited sharing capabilities of GPU are problematic for the FaaS platform because functions require sharing the limited GPU resources to maximize GPU utilization and performance.

Estimating the latency of data transfer and execution of the GPU is a significant challenge for scheduling FaaS functions that have access to GPUs. The initialization phase of the GPU application takes longer in GPU memory than in host memory because the GPU has the extra overhead of transferring the data from host memory to device memory. Additionally, determining the size of the input data the GPU will process is critical as the benefit of GPU acceleration must outweigh the cost of the data uploading overhead. Estimating the latency of FaaS functions using GPU is imperative, as accurate estimation allows the FaaS platform to improve the performance of FaaS functions using GPU resources.

The major challenge of GPU-enabled FaaS is to address the above limitations by finding the balance between locality and load-balancing. From a locality perspective, the GPU-enabled-FaaS can reduce the function latency by serving requests on the same cached GPU that already has uploaded the model. However, the extreme locality may increase the average latency of requests because all the requests are forwarded to the cached GPU while other uncached GPUs stay idle. From a load-balancing perspective, the GPU-enabled-FaaS can increase GPU utilization by distributing the requests evenly to available GPUs. However, load-balancing may increase the cache miss ratio when handling a workload with a larger working set size. The more extensive working set increases the chance of evicting popular models in limited memory space.

The thesis introduces complementary components that allow the existing open-source FaaS platforms to utilize GPU resources and improve the performance of ML inference running as FaaS functions. The GPU Manager decouples the GPU resource management from the FaaS platform by handling the GPU resources on behalf of FaaS functions and estimates each GPU's finish time of its queued requests. The Cache Manager treats the uploaded inference models in GPU memory as cache items and follows the LRU cache eviction policy to evict the unpopular models when GPU memory exceeds. The Scheduler reads the estimated finish times and LRU lists from the GPU Manager and Cache Manager and follows a scheduling policy to dispatch the FaaS Function to a GPU.

The proposed locality-aware scheduler improves the GPU utilization by balancing the workload to GPUs and increasing the hit ratio of cached items in GPUs. The scheduler reads the LRU list from the Cache Manager to prioritize the requests to be dispatched to the idle GPU with the cached items. However, suppose the GPU with the cached item is busy. In that case, the scheduler uses the estimated finish

time of the busy GPU to determine whether the cache hit in the busy GPU has a lower estimated finish time than the cache miss in the idle GPU. The scheduler only forwards the cache miss request to idle GPUs when the busy GPUs do not provide a lower finish time with cache hits.

The performance of the GPU-enabled FaaS is evaluated using the real-world trace disclosed by the public cloud provider and inference models widely used in production. The results indicate that the proposed locality-aware scheduler improves the performance of FaaS functions with a limited GPU resource. The locality-aware scheduler reduces the average latency and cache miss ratio of the baseline (load-balancing) scheduler by 80% and 65%. With the out-of-order dispatch, the scheduler further reduces the average latency and cache miss ratio of the baseline scheduler by 97% and 81%.

The rest of the paper is organized as follows: Section 2 introduces the background and related works; Section 3 describes the design of the GPU-enabled-FaaS; Section 4 discusses our performance evaluation of the five scheduling policies, and Section 5 concludes the paper.

Chapter 2

BACKGROUND

2.1 Function-as-a-Service

Function-as-a-Service (FaaS) is emerging as a popular cloud computing service that can provide scalability and cost-efficiency to event-driven applications Jonas et al. 2019. Amazon Web Services Miller, Vandome, and McBrewster 2010, one of the major public cloud providers, markets the FaaS as part of AWS Lambda Chand 2021 to provide scalability and cost-efficiency while running short event-driven jobs. Unlike traditional cloud services that provide the entire VM instance to the user, FaaS introduces function as a service unit by deploying each user's function code in lightweight containers. Using lightweight virtualization takes less space and time to run the functions, allowing FaaS significantly improve scalability and cost-efficiency for event-driven workloads.

To deploy the FaaS function in Lambda, the user must first select the type of function and then write the function code to upload to the API-Gateway. Lambda provides options for deploying functions with specific dependencies such as Programming languages: Java, Python and popular 3rd party libraries such as Tensorflow or Pytorch. The registered function responds to specific events such as an HTTP request or a trigger from other AWS services such as S3 and DynamoDB.

Figure 1 explains the components of the FaaS architecture. The general FaaS architectures contain three major components: API Gateway, Watchdog, and Datastore that run on top of the container orchestration, such as Docker Swarm Soppelsa

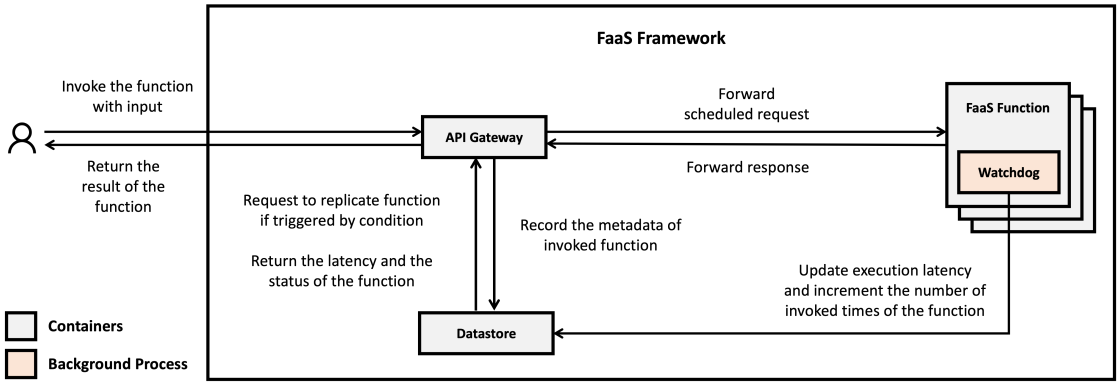

Figure 1. General FaaS architecture

and Kaewkasi 2017 or Kubernetes Hightower, Burns, and Beda 2017. The grey boxes represent the applications that run on containers. The red box represents the background process that runs inside the container and terminates with the container.

API Gateway component is the public route that interacts with the end-user by handling the create, read, update, delete (CRUD) of functions and invoking the registered functions. The Watchdog runs in the background along with the function code to serve as the communication bridge between API Gateway and Datastore. The Watchdog receives a request and returns a response to API Gateway and stores history status such as execution latency to Datastore. Datastore stores the history log of the invoked functions. Datastore can also be configured to trigger the API Gateway to scale up the function containers when the same function is repeatedly invoked for a specific time interval.

The end-users of the FaaS service can write function without configuring any resources and installing dependencies required to run the function. The end-users can use the code template provided by the specific FaaS platform to deploy the FaaS function. Once the end-user forwards the function code and the template to the FaaS platform, the platform builds the function by creating a running container that installs

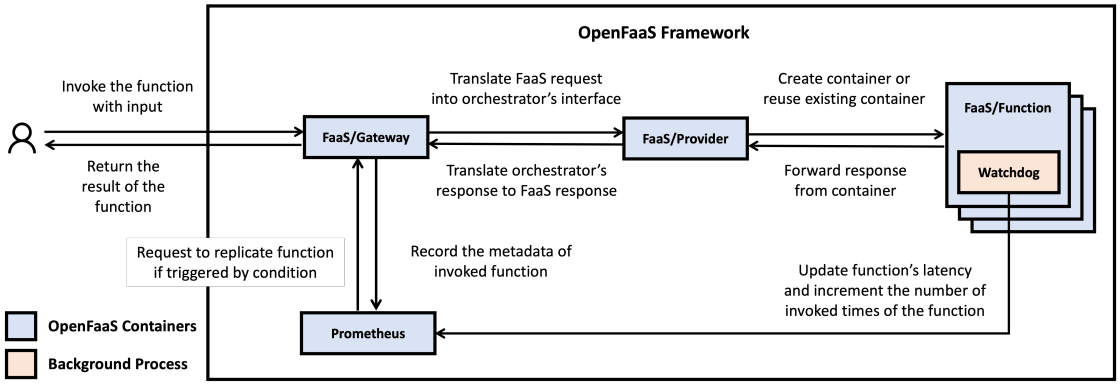

Figure 2. OpenFaaS architecture

the required resources written in the template. The deployed function is registered as a RESTFUL API, and the end-user can invoke the function by creating an HTTP request or implementing a trigger in other applications.

Netflix is one of the avid users of AWS cloud services that shows an exemplary use case of AWS lambda Retter 2020. Netflix uploads thousands of video files submitted by the producers in S3, which must be encoded and correctly reordered before the customers can stream the file as a movie. The S3 is configured to trigger multiple Lambda functions when the producer finishes uploading a video file. The Lambda functions split the updated files into 5-minute blocks and encode them in parallel. After the last block of the video is processed by the final Lambda function, the S3 triggers another Lambda function that aggregates the encoded files following the correct order.

2.2 OpenFaaS

We now explain the fundamental components and the lifecycle of the OpenFaaS, an open-source FaaS framework, in Figure 2. OpenFaaS simplifies the management

of containers by running on top of container orchestrators such as Docker Swarm or Kubernetes. The FaaS/Gateway (API-Gateway) works as an intermediator between the end-user and the internal components of OpenFaaS by forwarding create, delete, and modify requests to FaaS/Provider. FaaS/Gateway records the unique function ID and name to Prometheus when the user registers the function so that Watchdog can record the logs of the running function.

Prometheus Rabenstein and Volz 2015 is an open-source time-series database that collects metrics from FaaS/Function and can be configured to trigger alerts to OpenFaaS/Gateway. The FaaS/Provider translates the request forwarded by FaaS/Gateway into acceptable request that the interface of the container orchestrator can understand. FaaS/Function is the container that runs the function code of the end-user, and the Watchdog is a background process that communicates with the FaaS/Provider and stores the metrics such as function results, and latency to Prometheus.

The lifecycle of deploying and invoking functions in the default OpenFaaS is straightforward. The end-user registers the function by building a container with the function code and OpenFaaS template and invokes the function using HTTP requests to the FaaS/Gateway. The FaaS/Gateway either creates a new function container or runs the requested function by forwarding the request to the FaaS/Provider.

FaaS/Provider translates the request received from API/Gateway into a message that the container orchestrator, such as Docker swarm or Kubernetes, can understand. The container orchestrator finds the registered container that matches the description of the requested function and deploys the container that represents the FaaS/Function. The Watchdog that runs as a background process inside the container receives the request, returns the response to FaaS/Provider, and updates metrics such as latency

as a history log in Prometheus. Finally, the FaaS/Gateway receives the return message from the FaaS/Provider and returns the result to the end-user.

It is important to note that the feature of scheduling decisions such as replicating functions or load-balancing is not implemented in the OpenFaaS framework. The absence of the scheduling feature in OpenFaaS is to maintain the OpenFaaS framework as simple as possible by bequeathing all the matured scheduling decisions to 3rd party components. For example, Prometheus is in charge of making decisions for replicating the number of containers by triggering the FaaS/Gateway when Prometheus detects a burst of duplicated requests. Container orchestrators are responsible for making fundamental scheduling decisions such as load-balancing for deploying the containers to the available nodes.

2.3 GPU Computing

Graphics Processing Unit (GPU) Agbaje, Ohwo, and Adekunle 2018 introduces a different design than conventional multi-core processors by offering thousands of simple cores and a high bandwidth memory architecture. A GPU has multiple streaming multiprocessors (SM) that contain computing cores, shared cache, and shared memory. Each SM holds local registers, an integer arithmetic logic unit, and a floating-point unit for computing cores to execute instructions.

A GPU memory management unit (MMU) provides virtual address spaces for GPU applications by mapping the application context with the unique GPU page table. The GPU is an external device that communicates via PCI Express (PCIe) with the host, and the data transfer between the host and GPU is required for the application to utilize the GPU. Memory-mapped input/output (MMIO) allows the CPU in the

host to access the GPU registers and memory. GPU applications communicate with the GPU by sending operations using the MMIO interface.

GPUs are traditionally used for graphics acceleration, but the popularity of general-purpose computing such as machine learning has been increasing dramatically. The graphics acceleration focuses on rendering 2D, and 3D graphics using GPU rendering features such as tessellation, compute shaders, and multi-threading. The applications that benefit from GPU rendering capabilities range from video encoding programs to visually intensive 2D and 3D graphic games. General-purpose computing focuses on improving the performance of throughput-sensitive and computation-intensive applications. Machine learning models widely use the GPU features of general-purpose computing using API frameworks such as CUDA Cook 2012 and OpenCL Khronos OpenCL Working Group 2011.

The growing demand for accelerating ML applications with GPUs gave birth to popular ML framework APIs such as Tensorflow Martín Abadi and Ashish Agarwal 2015 and Pytorch Paszke et al. 2019 that support tools and pre-trained models that are tailored for GPUs. For example, TensorFlow provides a GPU-enabled version that allows programmers to deploy the vectors and matrices (tensors) and the functionality without the knowledge of CUDA. Furthermore, a growing number of research works focus on applying GPU acceleration to model serving platforms such as TF-Serving Olston et al. 2017 that host the pre-trained models on cloud or on-premise for ML applications to share. For example, Hu et al.Hu et al. 2018 integrate GPU scheduling with the existing model serving system to further improve application performance and GPU utilization.

A GPU disassembles the application into many threads operating on different data spaces, maximizing execution throughput with high data parallelism. For example,

CUDA API allows the programmer to define functions called kernels run by the computing cores of GPU. CUDA threads can execute the kernels, and they have access to the local memory shared among the threads defined in the same thread block and the global memory. The applications using GPU need the functionality of the GPU memory allocation and deallocation and data transfer between host and GPU memory.

GPUs have three limitations that make it unsuitable for multiple applications to share the same GPU. First, the capacity of GPU memory is significantly lower than host memory. Without using the special features such as Unified Memory available in devices equal to or higher than Nvidia's Pascal architecture, the GPU memory does not automatically take care of oversubscribed memory. If the processing data size is larger than the available GPU memory capacity, GPU programmers are responsible for managing the active working set in GPU memory. The oversubscription of GPU memory ultimately leads to an out-of-memory error (OOM), terminating the entire GPU application.

Second, the multiple computing cores of the GPU are not designed for multiple applications to effectively share. In other words, running multiple inference applications that process a small number of images concurrently provides sub-optimal performance compared to a single inference application that processes a large number of images. Third, although GPU provides fast computation for inference models, it incurs extra overhead while transferring data from host memory to GPU memory. The data transfer overhead of GPU arises in the PCIe interface as the maximum bandwidth of the current PCIe is low (i.e., 16GB/s) compared to the internal memory bandwidth of GPU (i.e., hundreds GB/s). To address the three limitations, it is essential to build a GPU scheduler that can both balance requests across GPUs and utilize the models already loaded in GPU memory to service the requests.

2.4 Deep Learning

Deep learning applications based on deep neural networks (DNN) are the key solution to many important tasks, such as voice recognition, natural language processing, image classification, and object detection Hu et al. 2018. DNN is an advanced version of artificial neural network (ANN), which is a subfield of machine learning where the architecture follows the brain's neural networks. The architecture of DNN is represented as the weighted directed graphs where the neurons are grouped as multiple layers, and each connection between neurons communicates with each other. The Convolutional Neural Network (CNN) O'Shea and Nash 2015, one of the major classifications of DNN Liu et al. 2017, is used as the primary workload in this paper's evaluation.

The main tasks of deep learning comprise training a model and using the model for inference. Training updates the weights of each layer iteratively towards a target by running the forward and backward propagation. Inference uses the updated weights of each layer to make a prediction based on the input by running the forward propagation. For image classification, training reduces the difference between the result of the forward propagation and the ground truth label by updating the weights of a model during backpropagation. The inference predicts the image class of the image input by translating the input data into a numeric result representing the classes.

GPUs offer excellent parallelism for both model training and inference. For example, the convolution operation in CNN requires configuring a fixed-sized filter to generate a feature table from the input and determine the batch size in training and inference. The filter performs repeated calculations by traversing through the whole image file,

and the repeated calculations provide opportunities for GPU to exploit parallelism. The larger batch size allows more data to be propagated through the neural network, thus allowing more input to be processed in parallel. Therefore, GPUs can significantly reduce the latency than CPUs.

The thesis focuses on FaaS functions running ML inference as the characteristics of inference applications are ideal for the FaaS platform to maximize scalability. The training applications are predictable long-running tasks that generate the pre-trained model by processing many training samples. The inference applications are short-running tasks triggered on-demand and process a small amount of input data using the pre-trained model. Unlike training, the inference can significantly benefit from the scalability of FaaS as the workload is unpredictable and resource occupation is small.

The stateless nature of ML inference greatly benefits inference functions from FaaS architecture. The training application requires the computed result from forward propagation to be stateful, as the computed result is used again during the backward propagation. However, the inference application performs stateless computation as it returns the output by running the forward propagation. As the inference can run stateless, the FaaS platform has less restriction to scale in and out when provisioning resources to multiple ML inferences.

Apart from scalability, the FaaS architecture improves the productivity of software development. FaaS platform maintains the functions as containers that can be installed with the required dependencies of ML libraries. ML inference running on the FaaS function no longer needs to address the compatibility issues, as the functions run on the predefined containers. Also, FaaS improves the reusability of the functions as the ML inference models can be treated as a function and be accessed by multiple applications such as smart-car and traffic surveillance applications.

2.5 GPU-enabled FaaS

Enabling GPUs in FaaS can significantly reduce cost and improve the performance of computation-intensive applications Baldini et al. 2017. Specifically, the use cases of ML inference in FaaS are becoming popular throughout the industry Zaharia et al. 2018. FaaS functions that use ML inference require significant computation resources and parallelization; therefore, introducing GPU to FaaS is crucial to reduce latency. However, addressing the unique challenges of GPU FaaS like GPU sharing and GPU scheduling make optimizing the inference functions difficult.

There are three challenges to enabling GPUs on FaaS for ML inference. Firstly, the non-preemptive characteristic of SMs degrades the performance of multiple processes sharing the same GPU. The thread block is the schedulable unit consisting of a number of threads translated into a computation core residing in one of the SMs in the GPU. The end-user can allocate thread blocks to the computation task called kernels which can perform computations on the SM of the GPU. Once a thread block is dispatched to the SM, the execution of the SM cannot be preempted by another thread block.

Secondly, the accelerators such as GPU and FPGA are limited resources compared to traditional resources such as CPU and memory. The scarcity of GPU resources could be a bottleneck if the FaaS functions are dependent on GPU to finish the tasks. The latency of the FaaS function may increase if multiple FaaS functions are throttled by waiting for the release of the GPU resources from other Faas functions. Therefore it is vital to share the limited GPU resources efficiently without hurting the scalability of FaaS.

Thirdly, the overhead of GPU data transfer creates a challenge for inference applications. The latency of the training application is long as the application needs

to process sample data to create the pre-trained model. The data transfer overhead is obsolete in a training application, as the training application requires running forward and backward propagation on a large number of sample data to create the pre-trained model. On the other hand, the data transfer overhead takes a large portion of the inference latency because the inference application runs a forward propagation of a small number of input data. Thus, depending on the input data size, the inference application can perform better on the CPU than the GPU due to the data transfer overhead.

2.5.1 GPU Sharing

Kim et al. Kim et al. 2018 introduce a FaaS platform with GPU support by enabling containers used in the FaaS platform to access GPU directly. Although the research shows the benefits of GPU-enabled functions, the functions use the NVIDIA Container Toolkit "Nvidia Docker Container Toolkit" 2022 that restricts multiple containers from sharing a single GPU. The function that occupies the GPU may run non-GPU tasks such as preprocessing the input images while preventing other waiting functions from using the GPU. Our solution solves the issue of GPU monopolization by enabling the functions to share the GPUs and provide optimized GPU resource management for the functions. For example, our functions in GPU-enabled FaaS occupy the GPU resource only when uploading the inference model or running the inference.

Naranjo et al. Naranjo et al. 2020 overcome the GPU monopolization by introducing rCUDA, a GPU virtualization framework, to FaaS Duato et al. 2010. The solution prevents the FaaS functions from directly managing GPUs by intercepting the

GPU operations from FaaS functions to the rCUDA interface. The research entirely decouples the FaaS from GPU resource management by relying on rCUDA. The FaaS platform and rCUDA fail to coordinate to improve performance because they do not share the information on pending FaaS requests and GPU utilization. Our solution decouples the GPU resource from FaaS functions but also allows coordination between global GPU cache management and FaaS scheduler to improve the performance of GPU-enabled FaaS.

Satzke et al. Satzke et al. 2020 implements features for GPU sharing on top of Knative, an open-source framework that provides tailored features for managing FaaS functions in Kubernetes. The proposed solution translates the memory of a single GPU into multiple vGPUs and provides a constraint policy that prevents FaaS functions from oversubscribing the GPU memory. However, the solution focuses on avoiding out-of-memory (OOM) errors caused by GPU memory oversubscription while failing to address the increased overhead caused by multiple functions sharing the same GPU. Our solution prevents over-subscription of GPU memory and reduces the average runtime of functions by introducing GPU cache management compatible with GPU scheduling.

Dakkak et al. Dakkak et al. 2019 introduce GPU caching by implementing a daemon process that provisions the GPU memory to functions by intercepting their CUDA requests. The solution uses GPU memory virtualization services such as Unified Memory and CUDA IPC to share uploaded models among GPU memory and processes. The solution prevents OOM in GPU by limiting the memory usage of GPU and using basic cache mechanisms such as LRU to reduce the latency. However, the solution fails to address locality-aware scheduling that can further increase the cache hit and uses load-balance scheduling by forwarding the requests to the GPU with

the lowest utilization. Our solution uses LRU as the GPU cache replacement policy while managing the GPU resources globally and a locality-aware scheduler tailored to improve performance on the real-world trace.

2.5.2 GPU Scheduling

Cox et al. Cox et al. 2020 introduce KFServing, a lightweight auto-scaling technique that scales out GPU resources according to the number of pending requests waiting to be scheduled. KFServing promotes batching by waiting for the requests that use the same model within a fixed period and batching them for processing on the same GPU. The solution fails to address the optimal waiting time to accumulate enough requests to create batch requests. The paper warns that the fixed waiting time may degrade the performance on a low-intensive workload because the scheduler may waste time waiting for requests to be batched. Furthermore, the paper fails to provide the ideal batch size that guarantees both locality and load balance in GPU scheduling. Our solution uses the scheduling policy that improves locality and load balance by estimating the optimal size of batched requests without specifying the waiting period.

Both Romero et al. Romero et al. 2021 and Zhang et al. Zhang, Krintz, and Wolski 2020 introduce a FaaS scheduler that reduces the latency by allocating the FaaS functions on the resources such as CPU, GPU, and network with the lowest estimation overhead. The scheduler improves both locality and load balance by assigning computation-intensive functions to GPU and prioritizes batching as long as there is no violation of the target latency. However, both schedulers fail to provide on-demand estimations as they are unaware of how the GPU resources are managed, as they rely on GPU virtualization provided by public or private cloud providers.

For example, periodical profiling assumes that a function recently deployed on the specific GPU may perform faster as the GPU memory still has the required data. The GPU may evict the uploaded data as multiple functions share the GPU, and the scheduler needs to wait for the following periodical profiling to have the correct estimation for the GPU overhead. The problem intensifies as the network and CPU may encounter unforeseen issues and affect the GPU overhead while coordinating with the GPUs. Our scheduler makes on-demand estimations free from profiling by allowing the scheduler to coordinate with the global GPU cache management.

Prakash et al. Prakash et al. 2021, Garg et al. Garg et al. 2021, and Yang et al. Yang et al. 2020 use function code of GPU (kernel) as a schedulable unit to support the schedulers with time-share and space-share of GPU resources. The solutions provide space-sharing of GPU by dynamically changing the allocated compute cores of the kernels and control time-sharing of GPU by slicing the kernel input into small batches. The solutions customize a specific version of the CUDA API (CUDA version 9) so that the CUDA API framework can allow kernels to be compatible with different thread block sizes. However, the solutions are unrealistic for inference applications as pre-trained models require occupying a fixed number of thread blocks, and inputs must be processed in large batches to benefit from GPU acceleration.

The kernel-slice scheduler expects each FaaS function to request for GPU resourced as kernel with undefined size, which is not compatible with ML inference as each pre-trained model requires a bundle of kernels with a specific resource size. Our solution is highly compatible with any ML inference and any version of the CUDA library because we provide Pytorch's generic actions, such as data transfer and execution of GPUs as a schedulable unit. First, our solution includes GPU Manager that intercepts requests from ML libraries such as Pytorch API, and no complicated changes in the CUDA

library are required. Second, we provide coarse-grained but lightweight scheduling decisions that reduce average latency and control starvation with a parameter.

Chapter 3

DESIGN

3.1 Architecture

The objective of our proposed GPU-enabled FaaS is to improve the performance of GPU functions, especially for model inference functions, by optimizing GPU scheduling and resource management. Figure 3 shows our framework's complete architecture that includes four additional containers (Scheduler, ETCD, Cache Manager, GPU Manager) which enables and optimizes GPU functions upon an existing FaaS framework such as OpenFaaS. The blue color boxes and the solid arrow lines represent the default components (running as containers) and the data flow of the default OpenFaaS. The yellow color boxes and the dotted arrow lines represent the components and data flow of the new solution.

The architecture explains the components and the life-cycle of our new solution built on top of the OpenFaaS with minor changes in FaaS/Gateway. The end-user can include a GPU-enable flag in the Dockerfile of FaaS/Function when registering the function using the FaaS/Gateway. The FaaS/Gateway checks for the GPU-enable flag in the Dockerfile and replaces the Pytorch interfaces with the customized Pytorch interface. The change of Pytorch API is not visible to the end-user as the customized interface receives the same parameters as the original interface. The custom interface replaces the generic Pytorch interface that accesses the GPU resources, such as data transfer and execution of GPU, with the interface that redirects those requests to the GPU Manager.

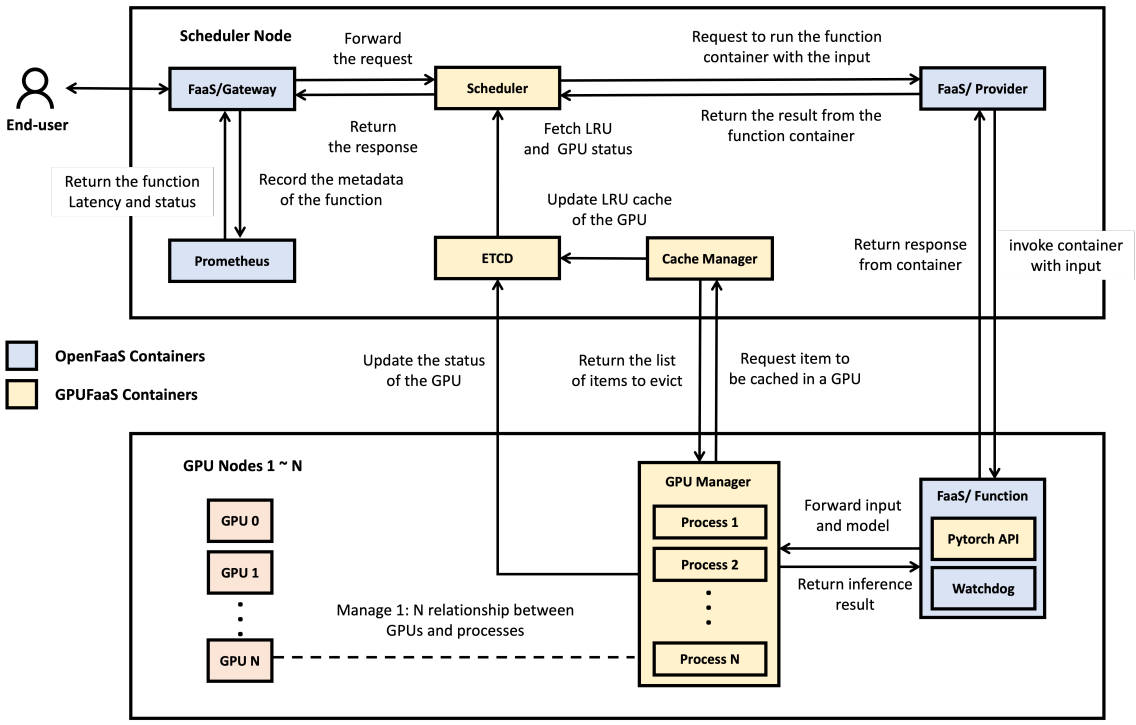

Figure 3. GPU cache in OpenFaaS Architecture

3.2 Scheduler

The role of the Scheduler component is to schedule the function request to a GPU before forwarding the request to the FaaS/Provider. The Scheduler follows a specific scheduling policy that can be enabled when the Scheduler component is first initiated. Once the Scheduler decides the requests that need to be dispatched, it groups the function information with the GPU address and forwards them to the FaaS/Provider. The function request contains the input data and the registered function's ID that uses the pre-trained model for inference. The GPU address contains the IP address of the server where the GPU is installed and the device name that is used to access the GPU on that server.

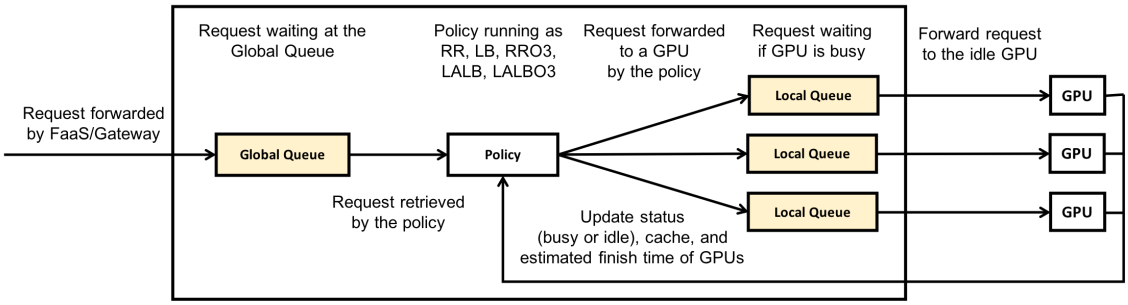

Figure 4. Logical representation of the Scheduler

The diagram of Figure 4 explains how to dispatch the forwarded request to a GPU according to scheduling policies. Two queues (global queue and local queue) are used for the request to wait before being dispatched to the GPU. Once the request is forwarded from the FaaS/Gateway to Scheduler, Scheduler puts the request into the global queue. The pending requests in the global queue are sorted by the earliest arrival time. Scheduler dispatches the pending requests to GPU following the scheduling policy. If the GPU selected by the policy is in busy state, Scheduler puts the requests in the local queue of the selected GPU.

3.3 GPU Manager

GPU Manager exists in each GPU node and manages the GPU processes running on the GPU node. GPU Manager runs the Pytorch process on behalf of the function by receiving the inference request from the customized Pytorch API and returning the results to the Pytorch API. Each GPU process uploads an inference model when initiating, and the GPU process reports the latency reports to ETCD. When the GPU process is uploading or processing the inference request, GPU Manager reports to

ETCD that the GPU status is busy and then updates the status back to idle when the GPU process finishes the task.

The Algorithm 1 and the Algorithm 2 explain how GPU Manager works once receiving the request from the FaaS/Provider. GPU Manager enforces GPU to run one request at a time and sets the status of GPU to busy when GPU is processing the request. GPU Manager manages the local GPU processes on the same node GPU Manager is running on. GPU Manager communicates with Cache Manager to maintain the running GPU processes as cache items.

First, GPU Manager requests Cache Manager to get the cache hit or miss result and whether there are victim models to evict if there is a cache miss. If there is a victim, GPU Manager first kills the processes reported as victims to guarantee that the new GPU process has enough GPU memory to upload the inference model. After the GPU process uploads and runs the inference model, GPU Manager forwards the input for the GPU process to run inference. Once the GPU process finishes the inference, the GPU process returns the result, and GPU Manager returns the result back to FaaS/Provider when the inference function finishes.

3.4 Cache Manager

Cache Manager exists along with the Scheduler as a global component and follows the LRU cache policy to manage the models for each GPU in its memory. The cache item is a running GPU process that uploaded the inference model to GPU by GPU Manager. When the function is ready to use GPU, the function requests Cache Manager to return the GPU process that uses the required inference model. If the required GPU process exists in GPU Manager, this is equivalent to a cache hit as

Algorithm 1: GPU Manager

Input:

The specific GPU that the request uses

The inference model that the request uses

The input (image files) that request uses

```
1  Set the requested GPU as busy
2  Get victim models and cache result (hit or miss) from Cache Manager
   /* Each GPU process uploads a model to GPU and is represented as a cache
      item to Cache Manager                                                 */
3  If victim models exist {
4      Send Kill message to the GPU processes with victim model
5  }
6  If cache result is a miss {
7      Create the new GPU Process with model and input
8  Send input as a message to the GPU Process
9  Return the inference result as the response
10 Set the GPU as idle
```

the function can skip the model transfer and use the existing GPU process. If the required GPU process does not exist in the GPU Manager, this is equivalent to a cache miss as the Cache Manager requests the GPU Manager to create a new GPU process to have the new inference model uploaded to GPU memory.

Algorithm 3 explains how Cache Manager manages the GPU memory cache. Cache Manager receives a message that contains the available memory space of GPU that needs to run the inference, the name, and the required memory size for the model from GPU Manager. The purpose of Cache Manager is to provide the list of victim models that need to be evicted from GPU according to the LRU eviction policy. If there is a cache miss, Cache Manager provides a list of cache models for eviction to

Algorithm 2: Function GPU Process

Input:

The specific GPU that the request uses

The inference model that the request uses

The input (image files) that request uses

```
/* Cache miss scenario where model is uploaded to the GPU                    */
FinishTime = (uploading time + inference time)
Upload the model to the GPU
FinishTime -= uploading time
While True {
    Receive message from the GPU Manager
    /* Cache eviction scenario where the process is killed                   */
    If received an exit message {
        Break
    }
    If received an input from the message {
        /* Cache hit scenario where function skip model uploading and starts
           from here                                                         */
        If this is a not the first inference {
            FinishTime += inference time
        }
        Run the inference
        FinishTime -= inference time
        Return result to GPU Manager
    }
}
```

make enough space for the new model. Cache Manager updates the LRU list of GPU for every request.

Algorithm 3: Cache Manager

Input:

The selected GPU's name, used, and total memory

The inference model that is used by the request

```
1  Get the latest version of LRU list of GPU
2  If the model is a hit {
3      Update the model to the head of LRU list
4      Return model
5  }
6  If new model exceeds the GPU memory {
7      Collect the victims from the LRU list
8  }
9  Update the model to the head of the LRU list
10 Return victims (if exists) and model
```

3.5 Datastore

Etcd "Etcd: A distributed, reliable key-value store for the most critical data of a distributed system" 2021 stores the estimated latency of each inference model, the LRU list of each GPU, and the status of each GPU. Etcd is a distributed key-value store that guarantees a high level of consistency appliable in a distributed environment. The estimated latency is used by the Scheduler to decide the optimal GPU to dispatch the request during scheduling. The LRU list is used to prioritize the cache hits for the waiting requests in the local queue. The status of GPU is used by all policies to identify which GPU is available for the function to immediate dispatch.

Chapter 4

SCHEDULING POLICIES

By default, the OpenFaaS platform relies on the load-balancing scheduling policy that container orchestrators perform on dispatching the containers. The container orchestrators such as Kubernetes use CFS quota Bovet and Cesati 2005 as a schedulable unit to perform load-balancing. The load-balancing scheduler aims to maximize the overall CPU utilization by predefining the containers with the upper bound of the CFS quota and dispatching the containers to the CPUs while not exceeding the available CFS quota of each CPU.

We introduce a load-balancing scheduler that aims to maximize GPU utilization by forwarding the request to the idle GPUs that are least frequently used. However, the GPU load-balancing scheduler is limited compared to the CPU load-balancing scheduler in terms of performance and utilization. The CPU load-balancing scheduler provides better utilization by forwarding the requests to CPUs until the CPU reaches the limit because the CPUs are well designed for multiple processes to share the same CPU without degrading performance. However, the GPU resources are not designed for sharing; thus, the GPU load-balancing scheduler is limited to dispatching one request to one GPU at a time.

We propose a locality-aware and load-balancing (LALB) scheduler that addresses the GPU's limitation. Besides using the load balancing feature to improve GPU utilization, the scheduler treats the data uploaded to the GPU as cache items to combine the locality-aware feature to enhance the performance and utilization of the GPU. Locality-aware feature in the LALB scheduler reduces the latency of each

function by forwarding the request to the cached GPU to avoid the upload time of the inference model. However, the LALB scheduler also considers load balancing by allowing cache miss when the estimated finish time of cache GPUs is higher than uncached GPUs.

4.1 Round Robin

The Algorithm 4 explains the Round-Robin scheduler (RR), which serves as the baseline. The RR scheduler invokes when at least one request arrives at the global queue. The RR scheduler dispatches the waiting request in the global queue to the least frequently used GPU. If the GPU state is busy, the request will be forwarded from the global queue to the local queue of the selected GPU. Scheduler will dispatch the request at the head of the local queue once the GPU status becomes idle.

The RR scheduler is expected to have an imbalance as the it assumes that each GPU will process identical requests that require the same amount of GPU resource and runtime. Although the requests are evenly distributed to the GPUs, each request uses inference models with different sizes. The GPUs processing small pre-trained models will quickly become idle, while some GPUs processing large pre-trained models will become throttled. The limitation of the RR scheduler reveals that the scheduler requires to consider that each function uses a different amount of resources to avoid the load imbalance.

Algorithm 4: RR: Round Robin

Input:

The list of GPUs sorted by frequency

The list of pending requests sorted by earliest arrival in global queue

```
/* Scheduler starts when at least one request exists in the global queue  */
Foreach GPUs {
    Foreach request in global queue
        Move the request to local queue of GPU
        Break
    }
    If GPU status is idle {
        Dispatch the request at the head of the local queue to the idle GPU
    }
}
```

4.2 Load Balance

The Algorithm 5 explains the Load-balance scheduler (LB), which is another baseline scheduler. The LB scheduler reduces the latency of the function more than the RR scheduler by considering the GPU's status (busy or idle). Identifying the GPU status helps the LB scheduler prioritize idle GPUs available to process the request immediately, preventing the request from waiting in the local queue of the GPU. The LB scheduler starts when there is at least one waiting request in the global queue and one idle GPU and forwards the earliest arrived request to the idle GPU. The LB scheduler evenly distributes the requests with different GPU resource requirements by prioritizing the idle GPUs to process the requests first.

Algorithm 5: LB: Load Balance

Input:

The list of idle GPUs sorted by frequency

The list of pending requests sorted by earliest arrival in global queue

```
/* Scheduler starts when at least one waiting request and idle GPU       */
1 Foreach idle GPU {
2     Dispatch the request at the head of the global queue
3 }
```

4.3 Round Robin Out-of-Order

The Algorithm 6 explains the Round Robin Out-of-Order scheduler (RRO3) that increases the cache hit by allowing requests in the local queue to be dispatched out of order. The RRO3 scheduler is invoked as same as the RR scheduler, but the scheduling decision changes as the waiting requests in the local queue follow the out-of-order (O3) dispatch. When the GPU is idle and more than one waiting request in the local queue, the O3 dispatch prioritizes the waiting requests that use the models already cached in that idle GPU.

4.4 Locality-aware Load-balancing

The locality-aware and load-balance scheduler (LALB) is invoked only when at least one request is waiting in the global queue and at least one GPU is idle. The Algorithm 7 and the Algorithm 8 explains how the scheduler considers both the GPUs' load balance and the models' locality in the GPU memory.

First, the LALB scheduler gets the request from the head of the queue and checks for available idle GPUs that can generate a cache hit, i.e., have the requested model

Algorithm 6: RRO3: Round Robin Out of Order

Input:
The list of GPUs sorted by frequency
The list of pending requests sorted by earliest arrival in global queue

```
/* Scheduler starts when at least one requests exists in the global queue
   */
Foreach GPUs {
    Foreach request in global queue
        Move the request to local queue of GPU
        Break
    }
    If GPU status is idle {
        Foreach request in the local queue {
            If the request's model is cached in the idle GPU {
                Dispatch the request to the idle GPU;
                Break
            }
        } else {
            Dispatch the request at the head of the local queue to the idle GPU
        }
    }
}
```

already stored in their memory. If the request is found not to be cached in any idle GPUs, then the LALB scheduler immediately dispatches the request to the least frequently used idle GPU. If the request can be a cache hit, the request is dispatched to the one of the idle GPUs with the cached item. After the LALB scheduler finds out that there are no available cache hits in idle GPUs, the LALB scheduler searches for the cache hits in busy GPUs.

If there is a cache hit in a busy GPU, the LALB scheduler compares the estimated

finish time between cache miss in the idle GPU and cache hit in the busy GPU. If the cache hit in the busy GPU provides a lower estimated finish time than the idle GPU, the request is scheduled in the local queue of the busy GPU. The requests waiting in the local queue are used to calculate the estimated finish time of the busy GPU when the scheduler again finds the cache hit in the busy GPU. Finally, when no cached GPUs can produce a lower finish time than the idle GPUs, the LALB scheduler dispatches the request to one of the idle GPUs, creating a cache miss.

The estimated finish time is calculated by using the Table 1. The latencies of uploading the model and running the inference are collected by profiling each unique function on the GPUs in the system. The inference time represents combined latency of both processing the input and uploading the input data to the GPUs. For our experiments, all the models is expected to use the batch size of 32, and process the total size of the images is 24.4 MB. As long as the number of inputs is larger than the batch size, the larger batch size reduces the inference time as GPU can process more inputs in parallel. However, the configurable batch size is limited as a larger batch size requires more GPU memory, and the GPU memory is limited.

4.5 Out of Order Dispatch

The out-of-order (O3) dispatch prioritizes the waiting requests that can be a cache hit in one of the idle GPUs and can be applied in the RR and LALB scheduler. For example, applying O3 dispatch in the locality-aware load-balance out-of-order scheduler (LALBO3) changes the behavior of the LALB scheduler as the waiting requests in the global queue are no longer first come first served. The request waiting in the global queue may revisit the LALBO3 scheduler because the LALBO3 may

Algorithm 7: LALB: Locality-aware Load-balancing

Input:

The list of GPUs sorted by frequency

The list of pending requests sorted by earliest arrival in global queue

The map that records the number of visits made by the policy

```
/* Scheduler without out-of-order rule has a specified limit of 0          */
/* Scheduler with out-of-order rule has a specified limit of 25            */
```

1 Foreach idle GPU {

```
    /* Prioritize the waiting requests already dispatched to busy GPUs     */
```

2 If the local queue is not empty {

3 Dispatch request at the head of the local queue to idle GPU

4 **Continue**

5 }

```
    /* Following LALB scheduling policy                                    */
```

6 Foreach request in the waiting queue (starting from the head) {

7 If request's model is cached in the idle GPU {

8 Dispatch the request to the idle GPU

9 **Break**

10 }

```
        /* Enforcing out-of-order rule                                     */
```

11 If the number of visits of the request is higher than the specified limit {

12 Flag = **localityLoadBalance**(idle GPU, idle GPUs, busy GPUs, request)

13 **Break** if flag is True else **Continue**

14 } else {

15 Increment the number of visits of the request;

16 }

17 } Else {

18 Foreach request in the waiting queue (starting from the head) {

19 Flag = **localityLoadBalance**(idle GPU, idle GPUs, busy GPUs, request);

20 **Break** if flag is True else **Continue**

21 }

22 }

23 }

Algorithm 8: Function **localityLoadBalance**

Input:
The selected of idle GPU
The list of idle GPUs excluding the selected idle GPU
The list of busy GPUs
The selected request by the scheduling policy

```
/* Allow cache miss as there are no cached GPUs                    */
If the request is not cached in any other GPU {
    Dispatch the request to the idle GPU
    Return True
/* Dispatch this request to one of the idle GPUs                   */
} Else if the request is cached in another idle GPU {
    Dispatch the request to that idle GPU
    Return False
} Else {
    /* Considering cache hit in busy GPUs                          */
    Foreach busy GPU {
        Estimate the finish time of the request on the busy GPU
        If the finish time is less than the model loading time then {
            Move the request to the local queue of the busy GPU
            Return False
        }
    /* Allow cache miss as cache hits can no longer benefit        */
    } Else {
        Dispatch the request to the idle GPU
        Return True
    }
}
```

prioritize the latest requests that generate cache hits. A specified limit value is set to 25 by default in the O3 dispatch to prevent the waiting requests from starvation. If the number of revisits of the request exceeds the O3 limit value, then the request is immediately dispatched by the LALB scheduler regardless of a cache hit or miss.

Chapter 5

EVALUATION

5.1 Methodology

The experiment focuses on evaluating the performance of ML inference workloads. We have explored five aspects of GPU performance in GPU-enabled FaaS: average latency, cache miss ratio, GPU (SM) utilization, false-miss ratio, and average number of duplicated cache items of popular working set. We introduce our unique metric called the false-miss ratio. The false miss is a type of cache miss that happens when the scheduler does not forward the request to the cached GPU since the busy cached GPU is estimated to have a higher latency than the idle uncached GPU. The false miss ratio correlates with the number of duplicated cache items because the false miss decision forces idle uncached GPUs to store the cache item already held in busy cached GPUs.

5.1.1 Workloads

The Table 1 shows the 22 popular CNN models considered in our workload. In the table, we have checked the models' actual size and the occupation size in GPU memory when the model inference runs with the fixed batch size of 32. The Cache Manager uses this peak memory occupation size for cache replacement decision, as the GPU results in OOM if it exceeds the available memory. Based on the GPU

occupation size of the models, we have categorized the models into small, medium, and large groups.

We have used the Microsoft Trace Shahrad et al. 2020 to evaluate the performance of the schedulers. Our evaluation uses this trace because the trace represents the actual workload of FaaS functions provisioned by Microsoft Azure. Microsoft Trace contains 14 files that represents 14 days of invocations of each unique function. Each file provides a column representing each minute, a row representing each unique function, and a value containing the total invocations of the unique function per minute. We have extracted the first 6 minutes of the trace and normalized the maximum number of requests for each minute to 325 requests to prevent overburdening the 12 GPUs we are using for our experiment.

The total number of unique functions (working set) in Microsoft trace is 46413 which is too large for our testbed to handle. Therefore, we consider only the most frequently used functions as the working set in our workload. We use three different working set sizes: 15, 25, and 35. The larger working set size introduces more unique functions while maintaining the maximum number of requests per minute to be 325 requests. We map each unique working set to a model that belongs to a specific category (grouped by model size) and ensure models with different sizes (small, medium, and large) are distributed evenly in the workload.

The Microsoft Trace provides a workload with a highly skewed working set, as the top 15 popular working set represents 56% of the total invocations per minute. We have further identified that the working set below the top 15 represents less than 0.01% of the total invocations per minute. For this reason, we set the minimum working set size as 15 because we believe the top 15 represents the skewness of the working set

for each minute. The working sets are randomly distributed to each minute of our normalized workload while preserving the ratio of the actual invocations per minute.

Size	Name	Size(mb)	Model(sec)	Input,Inference(sec)
small	squeezenet1.1	1269	2.41	1.28
	resnet18	1313	2.52	1.25
	resnet34	1357	2.60	1.25
	squeezenet1.0	1435	2.32	1.33
	alexnet	1437	2.81	1.25
	resnext50.32x4d	1555	2.64	1.29
medium	densenet121	1601	2.49	1.28
	densenet169	1631	2.56	1.30
	densenet201	1665	2.67	1.40
	resnet50	1701	2.67	1.28
	resnet101	1757	2.95	1.30
	resnet152	1827	3.10	1.31
	densenet161	1919	2.75	1.32
	inception.v3	2157	4.42	1.63
	resnext101.32x8d	2191	3.51	1.33
large	vgg11	2903	3.94	1.29
	wide_resnet50_2	3611	3.16	1.31
	wide_resnet101_2	3831	3.91	1.32
	vgg13	3887	3.98	1.30
	vgg16	3907	4.04	1.27
	vgg16.bn	3907	4.03	1.26
	vgg19	3947	4.07	1.33

Table 1. Occupation size in GPU, uploading latency, and inference (fixed batch size of 32) latency of models

5.1.2 Dataset

For the input image used for inference, we have provided a small group of 150 image files which comprise standard datasets such as CIFAR10 Krizhevsky, Hinton, et al. 2009, Modified National Institute of Standards, and Technology (MNIST) Kaziha

and Bonny 2019, and Hymenoptera Paszke, n.d. The MNIST dataset provides a fixed 28x28 grayscale image file that splits into 60,000 training images and 10,000 validation images. CIFAR-10 includes fixed 32x32 RGB images that have 50,000 training images and 10,000 validation images. Hymenoptera provides RGB images with different sizes ranging from 50KB to 2MB that must be compressed before being used in model inference. The dataset consists of 245 training images and 153 testing images.

5.1.3 Testbed.

We conducted all the experiments on three GPU servers equipped with four GeForce RTX 2080 GPUs and deployed GPU Managers as Nvidia Docker containers that have access to GPU resources. A separate server is used to run Docker containers for Scheduler, ETCD, Cache Manager, and the components required for the OpenFaaS platform. Once OpenFaaS receives a request, OpenFaaS will deploy the FaaS function as the Docker container installed with our custom Pytorch API in one of the GPU servers. Pytorch API communicates to the GPU Manager, which manages the GPU resources in the server.

5.2 Latency Results

Figure 5 shows the total latency of the five scheduling scheduler. First, RR and LB shows the performance improvement by only using load-balancing. Both RR and LB schedulers show a similar cache miss ratio in Figure 6, but the LB reduces the average latency of the RR scheduler by 44% in a working set size of 35. The RR scheduler distributes the requests evenly to all the GPUs without considering that each

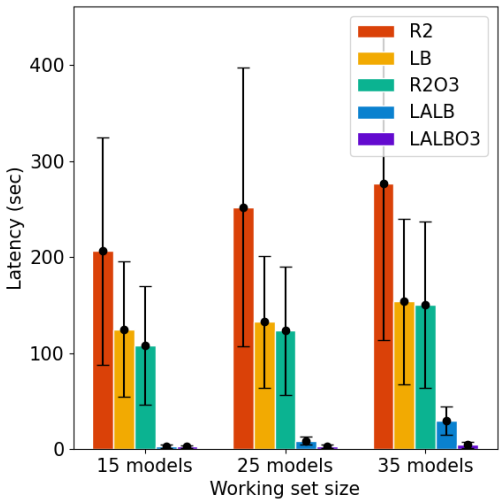

Figure 5. Average total latency

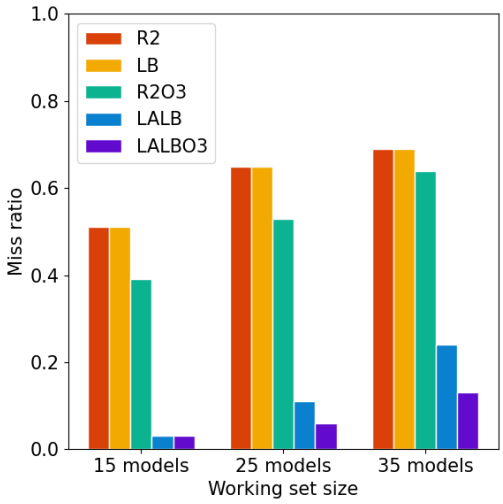

Figure 6. Cache miss ratio

request has different runtime. The requests with longer runtime may be skewed to a single GPU, causing load imbalance. The LB scheduler reduces the load imbalance by prioritizing the idle GPUs that are readily available to process the request.

The performance improvement of O3 dispatch is shown by comparing the RR and RRO3 schedulers in Figure 5. The O3 dispatch prioritizes the requests in the local queue of the GPU that can be a cache hit. RRO3 scheduler reduces the average latency and cache miss ratio of RR by 47% and 23% in a working set size of 15 because the O3 dispatch improves the locality of the RR scheduler. However, the average latency and cache miss ratio of RR reduces meagerly by 13% and 7% in a working set size of 35 because the more extensive working set size increases the chance of cache eviction.

One interesting factor is the comparison between LB and RRO3 because both schedulers have a similar cache miss ratio. LB scheduler reduces the latency with better load balance, whereas the RRO3 reduces the latency by increasing cache hits using O3 dispatch. When the working set size is 15, RRO3 reduces the average latency

of RR by 5% more than the LB because the working set size is small enough for local queues of a GPU to increase the cache hits. However, as the working set size increases, the performance difference between RRO3 and LB becomes negligible because the working set size is too big for each GPU to maintain without a considerable number of cache eviction.

The further performance improvement of locality-aware is shown by comparing LALB with RR and LB in Figure 5 and 6. The LALB scheduler reduces the average latency of RR by 98% and LB by 97% in working set size of 15 and 25. However, the average latency and the cache miss ratio of the LALB scheduler degrades as the working set size increases to 35. The result indicates that the cache miss ratio reduces by 94% in the working set size of 15 but reduces by 65% in the working set size of 35. The degrading performance is the same as the RRO3 because the improving locality becomes challenging when the working set size becomes more extensive.

Applying the O3 dispatch to the LALB scheduler further improves the performance in the working set size of 25 and 35. The O3 dispatch further prioritizes the cache hit by allowing requests in the global queue to be dispatched out of order. As the working set size increases, reducing the cache miss ratio becomes essential as the working set size overwhelms the limited GPU memory size. The LALB scheduler reduces the cache miss ratio of LB by 65%, and the LALBO3 scheduler reduces the cache miss ratio of LB by 81% in working set size of 35.

5.3 Utilization

Figure 7 shows SM utilization of the five schedulers. The SM utilization of the five schedulers remain consistent across all three working sets, as the maximum number

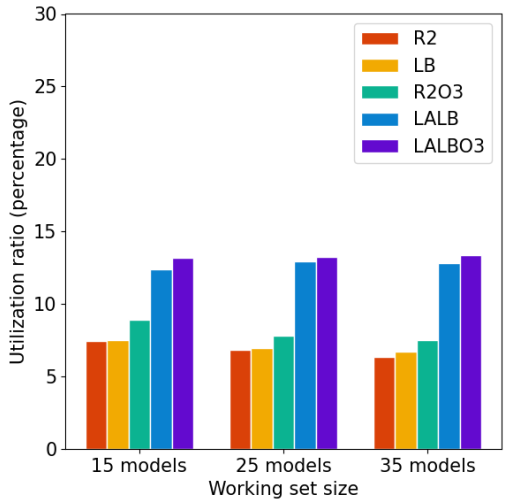

Figure 7. GPU (SM) utilization

of requests per minute is 325 for all three working sets. The result indicates that RR, LB, LALB, LALBO3 scheduler utilizes 7.48%, 7.51%, 12.41%, 13.17% of the computing cores of the GPU. Reaching the SM utilization of 100% is impossible as the GPUs accommodate multiple inference models and cannot risk exceeding memory by allocating a large batch size.

The LALBO3 scheduler has the highest SM utilization due to the lowest cache miss ratio. The SM utilization negatively correlates with the cache miss ratio because GPUs cannot use the SM to run the inference until the inference models are uploaded to the GPU memory. When there is a cache-miss, the SM utilization remains zero until the victim model becomes evicted and the new model is uploaded to the GPU. As a result, the LALBO3 scheduler shows the highest SM utilization as it has the lowest cache miss ratio.

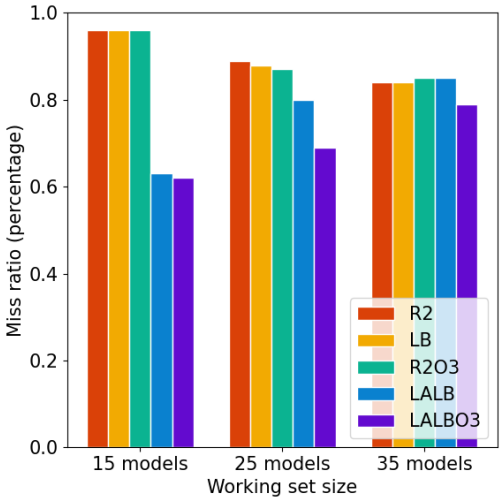

Figure 8. False miss ratio

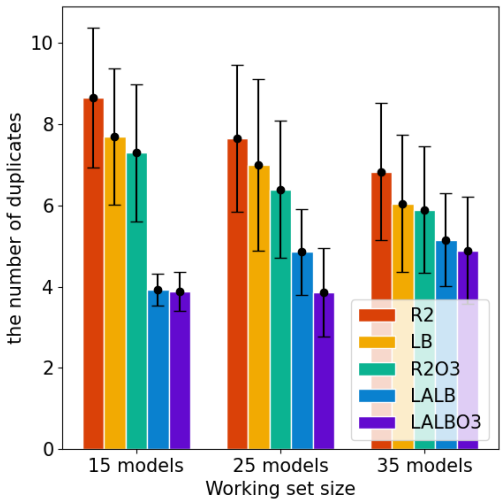

Figure 9. Average number of the top one duplicated model

5.4 Efficiency

This section explains the efficiency of the five schedulers determined by the false-miss ratio and the number of duplicated hot items. The ideal scheduler should maintain a minimal number of duplicated items while not degrading the number of cache hits. The false-miss ratio is a cache-miss scenario in the scheduling decision where the request is forwarded to the idle GPU as a cache miss even though a cached item exists in the busy GPU. The number of duplicated cache items is collected by tracking the total number of GPUs that cached the most popular working set (vgg19) in Table 2 per scheduling decision.

Figure 8 shows that both LALB and LALBO3 schedulers reduce the false miss ratio in working set 15 and 25. The LALB and LALBO3 reduce the false-miss ratio of RR by 34% and 35% while LB and RRO3 schedulers fail to minimize the false-miss ratio of RR in a working set size of 15. For the small working set size, the available GPU memory can find the optimal number of duplicated cache items to promote locality

Working Set	Scheduler	vgg19	densenet161	resnet152	resnet18	alexnet
15	RR	8.66	8.64	7.52	4.83	3.26
	LB	7.70	8.57	7.73	4.24	2.75
	RRO3	7.30	7.82	6.69	5.00	3.24
	LALB	3.93	4.91	3.95	3.92	1.94
	LALBO3	3.89	4.37	3.96	2.72	1.90
25	RR	7.65	7.29	6.94	4.14	2.92
	LB	7.58	7.22	6.53	4.00	2.63
	RRO3	6.40	6.68	6.13	4.00	3.19
	LALB	4.86	4.78	5.75	4.24	2.67
	LALBO3	3.46	5.41	3.86	4.36	2.64
35	RR	7.05	7.24	6.44	3.94	2.62
	LB	6.83	6.75	6.49	3.59	2.63
	RRO3	5.90	6.76	5.90	4.03	2.86
	LALB	4.56	5.20	4.08	4.19	2.86
	LALBO3	4.21	4.48	4.01	3.31	2.47

Table 2. Average number of duplicates for top 5 inference models

without the O3 dispatch. As the working set size increases to 35, only LALBO3 scheduler is able to reduce the false-miss ratio of the RR scheduler by 6%, as the LALBO3 has the O3 dispatch to exploit locality further by prioritizing the waiting requests to the cached GPUs.

Figure 9 shows the total number of duplicated models for the most popular function. As the GPU-enabled-FaaS uses 12 GPUs, the highest number of duplicated models cannot exceed the number 12. We further explain the average number of duplicated popular items for the top 5 popular items in Table 2. The vgg19, desnenet161, resnet152, resnet18, and alexnet are the top 5 popular working sets representing 13%, 11%, 9%, 4%, and 3% of the total workload.

The LB scheduler reduces the average number of duplicates of the RR scheduler by 11%. Both RR and LB scheduler does not consider locality are subjected to the situation where the duplicated cache items continuously evict each other. The LB

scheduler slightly reduces the number of duplicates compared to the RR scheduler because the scheduler reduces the scope of spreading duplicates to only idle GPUs.

The LALB scheduler reduces the average number of duplicates of the RR and LB scheduler by 54%, 43% in working set size of 15. The LALB scheduler improves locality by prioritizing the request to the cached idle GPU. The increased cache hits reduces the number of duplicated cache items per scheduling decision. Increasing the working set size degrades the ability of the LALB scheduler to maintain the optimal number of duplicates, as it meagerly reduces the duplicates of RR and LB scheduler by 24%, 13% in working set size of 35. The working set size of 25 and 35 degrades the locality performance of the LALB scheduler as it increases the chance of cache misses in the limited GPU memory.

The LALBO3 scheduler does not significantly reduce the number of duplicates compared to LALB as it reduces the average number of duplicates of the RR and LB scheduler by 55%, 44% in a working set size of 15. The working set size of 15 shows negligible performance improvement for the O3 dispatch because the available GPU memory is enough to cover the working set size. The LALBO3 scheduler performs better than the LALB scheduler in working set size of 35 by reducing the average number of duplicates of the RR and LB scheduler by 28%, 17%. The results indicate that by applying the O3 dispatch, the performance of locality can be further improved to reduce the average number of duplicates.

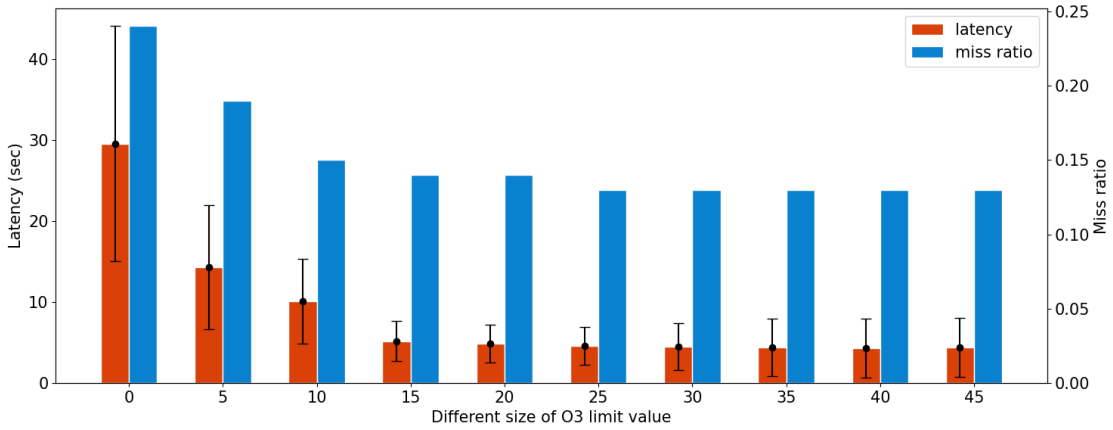

Figure 10. Latency and cache miss ratio of different O3 limit value

Limit	Avg lat (sec)	Lat var (sec)	Miss ratio	False miss ratio
0	29.54	210.15	0.24	0.85
5	14.30	8.22	0.19	0.81
10	10.10	7.71	0.15	0.81
15	5.20	6.02	0.14	0.81
20	4.88	5.51	0.14	0.81
25	4.61	5.49	0.13	0.79
30	4.53	8.33	0.13	0.79
35	4.40	12.54	0.13	0.79
40	4.32	13.14	0.13	0.79
45	4.30	13.15	0.13	0.79

Table 3. Performance under different O3 value in working set 35

5.5 O3 Sensitivity Test

Figure 10 focuses on the sensitive study of the specified limit value of the O3 dispatch in LALBO3 scheduler. We have experimented on the workload with the working set size 35, and changed the specified limit of the O3 dispatch from zero to 35 (x-axis). The total latency (left y-axis) and the cache miss ratio (right y-axis) are used as the evaluate the performance changes created by the different limit value. The

result indicates that the both latency and cache miss ratio reduce as we increase the specified limit value of O3. Note that we do not provide the results above the limit 45 as the latency, cache miss ratio, and variance do not change significantly.

The O3 limit value of 25 reduces the average latency and cache-miss ratio of the O3 limit value of 0 by 84% and 45%. The more significant O3 value increases the locality performance as it increases the number of times requests with cache hit can prioritize the earliest arrived requests. Furthermore, the O3 limit value of 25 reduces the variance of the average latency of the O3 value of 0 by 97%. The larger O3 value reduces the latency variance as the variance is contributed significantly by the busy GPUs uploading a new inference model. To check the performance changes in detail, we have provided Table 3.

Chapter 6

CONCLUSION

The demand for GPU-enabled FaaS is growing as the use-cases of ML inference tasks that can benefit from GPU acceleration increase in the FaaS platform. Our solution focuses on improving the FaaS functions running ML inference tasks such as CNN that heavily benefit from GPU acceleration. However, the existing GPU design provided limited resource sharing capabilities among multiple FaaS functions, and the short-lived nature of the FaaS function makes the GPU difficult to outweigh the cost of data transfer overhead with the benefit of parallelization.

Our approach is applicable to different FaaS frameworks, as it requires additional complementary components to introduce the GPU scheduling and management. Our GPU-enabled FaaS provides global management of GPU memory and treats the uploaded inference models in GPU as cache items to reduce the data transfer overhead of inference models. Furthermore, the GPU-enabled FaaS includes the LALB scheduler that considers both locality and load balance to improve the GPU performance of the FaaS functions.

We have used real-world trace and inference models widely used in production to evaluate the performance of our GPU-enabled FaaS. The LALB scheduler reduces the baseline (LB) scheduler's average latency and cache miss ratio by 80% and 65%. Additionally, the out-of-order (O3) dispatch can work with the LALB scheduler to improve the locality performance further. The LALBO3 scheduler reduces the LB scheduler's average latency and cache miss ratio by 97% and 81%. We believe that

the LALBO3 scheduler can provide optimal performance for GPU-enabled FaaS that handles model inference tasks.

Future work will improve the cache replacement algorithm by converting from local to global. Currently, the cache replacement follows the local LRU policy by evicting cached items within the GPU. The local LRU may evict the hot items if the local LRU list contains only hot items and misses the opportunity to evict the cold items cached in other GPUs. The plan is to improve GPU-enabled FaaS performance by converting the local LRU to a global LRU.

REFERENCES

Agbaje, Michael, Onome Ohwo, and Bammeke Adekunle. 2018. "Heterogeneous System Architecture (HSA)." *International Journal of Scientific Research in Computer Science Engineering and Information Technology* 3 (March): 2456–3307.

Baldini, Ioana, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, et al. 2017. "Serverless Computing: Current Trends and Open Problems." In *Research Advances in Cloud Computing,* 1–20. Singapore: Springer Singapore. https://doi.org/10.1007/978-981-10-5026-8_1.

Bovet, Daniel P, and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc."

Chand, Poornima. 2021. "Machine learning inference at scale using AWS serverless." *Amazon blog* (November). https://aws.amazon.com/blogs/machine-learning/machine-learning-inference-at-scale-using-aws-serverless/.

Cook, Shane. 2012. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Cox, Clive, Dan Sun, Ellis Tarn, Animesh Singh, Rakesh Kelkar, and David Goodwin. 2020. "Serverless inferencing on Kubernetes." *arXiv preprint arXiv:2007.07366.*

Dakkak, Abdul, Cheng Li, Simon Garcia de Gonzalo, Jinjun Xiong, and Wen-mei Hwu. 2019. "TrIMS: Transparent and Isolated Model Sharing for Low Latency Deep Learning Inference in Function-as-a-Service." In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD),* 372–382. https://doi.org/10.1109/CLOUD.2019.00067.

Duato, José, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique 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.

"Etcd: A distributed, reliable key-value store for the most critical data of a distributed system." 2021, https://etcd.io/.

Garg, Anirudh. 2020. "Why use Azure Functions for ML inference?" *Microsoft blog* (May). https://techcommunity.microsoft.com/t5/apps-on-azure-blog/why-use-azure-functions-for-ml-inference/ba-p/1416728.

Garg, Anshuj, Purushottam Kulkarni, Umesh Bellur, and Sriram Yenamandra. 2021. "FaaSter: Accelerated Functions-as-a-Service with Heterogeneous GPUs." In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC),* 406–411. IEEE.

Gunny, Alec. 2019. "Accelerating Wide and Deep Recommender Inference on GPUs." *TECHNICAL BLOG* (December). https://developer.nvidia.com/blog/accelerating-wide-deep-recommender-inference-on-gpus/.

Hightower, Kelsey, Brendan Burns, and Joe Beda. 2017. *Kubernetes: Up and Running Dive into the Future of Infrastructure.* 1st. O'Reilly Media, Inc.

Hu, Yitao, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. 2018. "Olympian: Scheduling GPU Usage in a Deep Neural Network Model Serving System." In *Proceedings of the 19th International Middleware Conference,* 53–65. Middleware '18. Rennes, France: Association for Computing Machinery. https://doi.org/10.1145/3274808.3274813.

Jonas, Eric, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. "Cloud programming simplified: A berkeley view on serverless computing." *arXiv preprint arXiv:1902.03383.*

Kaziha, Omar, and Talal Bonny. 2019. "A comparison of quantized convolutional and LSTM recurrent neural network models using MNIST." In *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA),* 1–5. IEEE.

Khronos OpenCL Working Group. 2011. *The OpenCL Specification, Version 1.1.* https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

Kim, Jaewook, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung 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.

Krizhevsky, Alex, Geoffrey Hinton, et al. 2009. "Learning multiple layers of features from tiny images."

Landaverde, Raphael, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. "An investigation of unified memory access performance in CUDA." In *2014 IEEE High Performance Extreme Computing Conference (HPEC),* 1–6. IEEE.

Li, Wenqiang, Guanghao Jin, Xuewen Cui, and Simon See. 2015. "An Evaluation of Unified Memory Technology on NVIDIA GPUs." In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing,* 1092–1098. https://doi.org/10.1109/CCGrid.2015.105.

Liu, Weibo, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. 2017. "A survey of deep neural network architectures and their applications." *Neurocomputing* 234:11–26.

Martín Abadi and Ashish Agarwal. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. https://www.tensorflow.org/.

Miller, Frederic P., Agnes F. Vandome, and John McBrewster. 2010. *Amazon Web Services.* Alpha Press.

Naranjo, Diana M., Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. "Accelerated Serverless Computing Based on GPU Virtualization." *J. Parallel Distrib. Comput.* (USA) 139, no. C (May): 32–42. https://doi.org/10.1016/j.jpdc.2020.01.004.

O'Shea, Keiron, and Ryan Nash. 2015. "An introduction to convolutional neural networks." *arXiv preprint arXiv:1511.08458.*

"Nvidia Docker Container Toolkit." 2022. *official documentation* (January). https://docs.nvidia.com/ai-enterprise/deployment-guide/dg-docker.html#enabling-the-docker-repository-and-installing-the-nvidia-container-toolkit.

Olston, Christopher, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. "TensorFlow-Serving: Flexible, High-Performance ML Serving." In *Workshop on ML Systems at NIPS 2017.*

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural Information Processing Systems 32,* 8024–8035. Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Paszke, Pytorch. n.d. "An imperative style, high-performance deep learning library." *Adv. Neural Inf. Process. Syst,* no. 32, 8026.

Prakash, Chandra, Anshuj Garg, Umesh Bellur, Purushottam Kulkarni, Uday Kurkure, Hari Sivaraman, and Lan Vu. 2021. "Optimizing Goodput of Real-time Serverless Functions using Dynamic Slicing with vGPUs." In *2021 IEEE International Conference on Cloud Engineering (IC2E),* 60–70. IEEE.

Rabenstein, Bjorn, and Julius Volz. 2015. "Prometheus: A Next-Generation Monitoring System (Talk)." Dublin: USENIX Association, May.

Retter, Mariliis. 2020. "Serverless Case Study - Netflix." *dashbird* (July). https://dashbird.io/blog/serverless-case-study-netflix/.

Romero, Francisco, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. "Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines." *arXiv preprint arXiv:2102.01887.*

Satzke, Klaus, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. 2020. "Efficient gpu sharing for serverless workflows." In *Proceedings of the 1st Workshop on High Performance Serverless Computing,* 17–24.

Scheuner, Joel, and Philipp Leitner. 2020. "The State of Research on Function-as-a-Service Performance Evaluation: A Multivocal Literature Review" (April).

Shahrad, Mohammad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider." In *2020 USENIX Annual Technical Conference (USENIX ATC 20),* 205–218.

Soppelsa, Fabrizio, and Chanwit Kaewkasi. 2017. *Native Docker Clustering with Swarm.* Packt Publishing.

Wolf, Thomas, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. "Transformers: State-of-the-art natural language processing." In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations,* 38–45.

Yang, Ryan, Nathan Pemberton, Jichan Chung, Randy H Katz, and Joseph Gonzalez. 2020. *Pyplover: A system for gpu-enabled serverless instances.* Technical report. Technical report, University of California, Berkeley.

Zaharia, Matei, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. "Accelerating the machine learning lifecycle with MLflow." *IEEE Data Eng. Bull.* 41 (4): 39–45.

Zhang, Michael, Chandra Krintz, and Rich Wolski. 2020. "STOIC: Serverless Teleoperable Hybrid Cloud for Machine Learning Applications on Edge Device." In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops),* 1–6. https://doi.org/10.1109/PerComWorkshops48775.2020.9156239.