Flashpoint:
A Low-latency Serverless Platform for Deep Learning Inference Serving

by

Justin David Quitalig San Juan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

Recent breakthroughs in Deep Learning (DL) have led to high demand for executing inferences in interactive services such as ChatGPT and GitHub Copilot. However, these interactive services require low-latency inferences, which can only be met with GPUs and result in exorbitant operating costs. For instance, ChatGPT reportedly requires millions of U.S. dollars in cloud GPUs to serve its 1+ million users. A potential solution to meet low-latency requirements with acceptable costs is to use serverless platforms. These platforms automatically scale resources to meet user demands. However, current serverless systems have long cold starts which worsen with larger DL models and lead to poor performance during bursts of requests. Meanwhile, the demand for larger and larger DL models make it more challenging to deliver an acceptable user experience cost-effectively. While current systems over-provision GPUs to address this issue, they incur high costs in idle resources which greatly reduces the benefit of using a serverless platform.

In this thesis, we introduce Flashpoint, a GPU-based serverless platform that serves DL inferences with low latencies. Flashpoint achieves this by reducing cold start durations, especially for large DL models, making serverless computing feasible for latency-sensitive DL workloads. To reduce cold start durations, Flashpoint reduces download times by sourcing the DL model data from within the compute cluster rather than slow cloud storage. Additionally, Flashpoint minimizes in-cluster network congestion from redundant packet transfers of the same DL model to multiple machines with multicasting. Finally, Flashpoint also reduces cold start durations by automatically partitioning models and deploying them in parallel on multiple machines. The reduced cold start durations achieved by Flashpoint enable the platform to scale resource allocations elastically and complete requests with low latencies without over-provisioning expensive GPU resources.

We perform large-scale data center simulations that were parameterized with measurements our prototype implementations. We evaluate the system using six state-of-the-art DL models ranging from 499 MB to 11 GB in size. We also measure the performance of the system in representative real-world traces from Twitter and Microsoft Azure. Our results in the full-scale simulations show that Flashpoint achieves a mean of 93.51% shorter average cold start durations, leading to 75.42% and 66.90% respective reductions in average and 99th percentile end-to-end request latencies across the DL models with the same amount of resources. These results show that Flashpoint boosts the performance of serving DL inferences on a serverless platform without increasing costs.

Acknowledgements

I thank my supervisor, Professor Bernard Wong, for his guidance and support throughout my graduate studies. Bernard has taught me to identify big-picture problems and seek creative solutions. He has also provided excellent feedback that have helped me improve my technical writing skills to bring a sensible order to the topics I discuss. My experiences under Bernard's supervision and the skills I have learned are undoubtedly valuable and lessons I will nurture throughout my life.

I thank the readers of my thesis, Professor Paul Ward and Professor Ali Mashtizadeh, for their actionable feedback. I also thank Sharon, Xinan, Rizwan, Dheeraj, Senyu, and Lucas for being there to attentively discuss the ideas in my thesis and help me seek out any weaknesses. In particular, I want to thank Rizwan for always being critical and whose book recommendations have also helped me grow as a person. I also thank other members in the Shoshin lab who have helped creating a great atmosphere to work and discuss interesting topics. Finally, I extend my gratitude to the CSCF staff and CS Graduate Office for their assistance throughout my studies. They have been essential to my completion of the program.

Dedication

This thesis is dedicated to the one I love, my partner, Joy, who has been my inspiration and source of encouragement throughout the challenges of graduate school and life. This work is also dedicated to my parents, Teodoro and Evelyn, and my family, who all have always loved me unconditionally and set excellent, virtuous, and hardworking examples in my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning (DL) products such as ChatGPT and GitHub Copilot have grown rapidly in demand in recent years. ChatGPT is a large language model with 175 billion parameters that provides an interactive and sophisticated chat experience on a wide range of subjects. Meanwhile, GitHub Copilot is a natural language model that autocompletes code, saving time for programmers. It is a challenge to operate these products cost-effectively with reasonable performance due to three important properties. First, these interactive products require low-latency responses; otherwise, users would not use them. For example, [106] indicates, as of 2022, that 90% of inferences across 600 machine learning models require response times below 200ms for services including chatbots, advertisements, and fraud detection. Second, these products experience dynamic demand due to being user-facing products, requiring the system to dynamically adapt to the load. Third, the operating costs to execute DL inferences is high. For ChatGPT, it has been reported to cost an estimated 3 million U.S. dollars per month [36] to perform the inferences for its 1+ million monthly active users.

Due to these properties, operating these services on a serverless platform is a promising approach. Serverless computing, as a paradigm, aims to automatically and cost-effectively meet the scaling needs of users while reducing costs compared to dedicated on-site hosting. To reduce costs and serve multiple customers, service providers use a shared compute infrastructure and allocate resources to a user's workload as needed. However, this dynamic allocation leads to the so-called "cold start" problem, which occurs when the demand for a program exceeds the service capacity of resources that are currently allocated. Requests must then wait until new replicas of the program are created on other nodes in the system.

With GPUs, the cold start problem is exacerbated and presents a barrier to the adoption of serverless for applications such as ChatGPT. This is because the cold starts cause long wait times that exceed the low-latency requirements in interactive DL products. The following factors amplify the problem. First, the size of DL models (in the order of megabytes to gigabytes) are often larger than general-purpose serverless code (within kilobytes to megabytes), causing longer cold starts. Second, short execution time requirements in DL inferences make the relative magnitude of cold starts larger and more undesirable compared to CPU-based systems. Third, while general-purpose systems can over-provision CPU resources at low cost to mask cold starts, GPUs are required to meet low-latency deadlines and are both limited and expensive. This makes over-provisioning costly for serving DL workloads.

While related works address parts of the cold start problem for general-purpose functions, they do not address the cold start problem in GPUs. For example, current systems such as [87] aim to reduce cold start frequencies through adaptive keep-alive windows. Other works that explore reducing cold start durations rely on memory sharing and creating new processes on the same machine [58, 25] which does not scale out service capacity for a DL model. Meanwhile, current commercial GPU-enabled serverless platforms such as AWS SageMaker [14] and state-of-the-art research [90] opt for over-provisioning despite its high cost. For instance, SageMaker recommends over-provisioning with a factor of 2 [111].

In this thesis, we introduce Flashpoint, serverless platform that achieves sub-second cold starts using model partitioning, locality-awareness, and network multicasting. These techniques enable cost-effective, low-latency elastic scaling of serverless DL serving applications. Flashpoint solves three keys problems regarding GPU cold starts:

1. Long process initialization durations from large DL model sizes.
2. Long cold start durations from downloading the DL model from cloud storage.
3. Network congestion during concurrent DL model downloads.

To tackle long process initialization durations, Flashpoint utilizes automated model partitioning to split the model into sequential parts and load them in parallel on separate GPUs. This reduces cold start durations through parallelism, thus achieving sub-second cold starts. Model partitioning trades off additional network latencies during inferences. Flashpoint's partitioning algorithm adaptively optimizes the partitioning scheme to minimize the average end-to-end request latencies of requests served by the partitioned model based on the expected number of requests. Without model partitioning, the duration of cold starts are bottlenecked by the duration loading the model after downloading it. These cold start durations are orders of magnitude greater than execution times, leading to wait times greater than acceptable interactive latencies [111, 77] (i.e. 1,000 milliseconds). While model partitioning improves the cold start durations by parallelizing the loading process on separate GPUs, the download process will be limited by the network bottleneck to cloud storage, often resulting in long wait times. This is the case when the compute cluster is separated from the cloud storage through the public internet. The network bottleneck from cloud storage to the compute cluster is an issue observed in our experiments and production systems [99, 113, 51]. Thus, another method on top of model partitioning is needed to address long cold start durations.

To reduce the duration of long cold start downloads from cloud storage, Flashpoint uses locality and remote memory pooling to create a compute-colocated distributed registry. This registry is then used to store and download the DL model across host machines, reducing the duration for downloads compared to cloud storage at no additional cost. To create this distributed registry, we first identify opportunities in unused memory space and network bandwidth in a serverless deployment. Then, by transparently changing the download source of a DL model, Flashpoint optimizes this critical path of downloading from cloud storage by prioritizing the download of the DL model from nearby host machines. In the best case, it completely negates the need for downloading the DL model for a GPU if it is available in the local host machine from previously being downloaded for another GPU. Flashpoint also maximizes the benefit of shorter paths using a Pareto-optimal greedy algorithm that fully captures the benefit of local downloads while maximizing the chance of future shortest-path cold starts. In comparison, current systems incur long cold start durations by downloading from AWS S3, PyTorch Hub, a remote NFS server, of an Elastic Container Registry [58, 21, 14].

In addition to reducing long cold start downloads from cloud storage, Flashpoint also tackles the problem of network congestion from concurrent DL model downloads. These downloads occur concurrently due to the autoscaler requesting multiple new replicas to be instantiated at the same time when reacting to changes in the workload. Flashpoint uses network-aware autoscaling where it utilizes multicasting protocols to minimize redundant packet transfers. In particular, concurrent cold starts cause congestion in network links when source host machines send the DL model to multiple recipient host machines. In Flashpoint, the network congestion problem is addressed through the application of reliable multicasting using the protocol of TCP chaining at the application level. This technique addresses redundant network transfers during simultaneous cold starts across host machines while guaranteeing completion of DL model transfers. In comparison, current multicasting protocols use UDP and is designed for broadcasting real-time messages using group addresses. Because of using the UDP as the underlying protocol, this form of multicasting requires additional logic for guaranteeing complete data transfers. The issue of network congestion becomes more prevalent when transferring DL model data from one host machine to multiple others. In this case, the bottleneck is in the uplink bandwidth of the source host machine. Reliable multicasting then minimizes network congestion and transfers data swiftly for multiple cold starts. Flashpoint also optimizes redundant DL model transfers to the same host machine through download sharing using a lightweight DL model manager in the host machines. These techniques, overall, bound tail cold start durations to the duration of host-to-host transfers without incurring additional costs as compared to cloud storage download times.

We evaluate the techniques in Flashpoint using experiments on our testbed, as well as a physical deployment on a Lambda Labs test bed. To further evaluate Flashpoint, we then use large-scale data center simulations parameterized with measurements from our testbed. In the experiments, we use six state-of-the-art DL models in the Natural Language Processing (NLP) domain with sizes ranging from 499 MB to 11 GB and real-world workload traces from Twitter [96] and Microsoft Azure [87]. We show that our approach achieves a 93.51% mean reduction in cold start durations and up to 75.42% and 66.90% respective reductions in average and 99th percentile end-to-end latencies compared to baselines while keeping the resource usage of techniques to be within 5% of the baseline for fairness.

To summarize, our contributions in this work are the following:

- We characterize the problem of cold starts in GPU-based DL inference serving systems.

- We design Flashpoint, a serverless framework that enables low-latency inference serving by reducing cold start durations. Flashpoint does this through the following. First, it divides cold start durations using automated model partitioning. Second, it exploits underutilized memory and network resources through remote memory pooling. Third, it minimizes GPU cold start durations through a compute-colocated distributed registry, multicasting, and download sharing.

- We evaluate Flashpoint through large-scale data center simulations parameterized with our physical testbed measurements. Our results show that, for the Twitter workload, Flashpoint achieves a 93.51% reduction in average cold start durations, 75.42% reduction in average end-to-end request latencies, and 66.90% reduction in 99th percentile end-to-end request latencies on average across DL models compared to baselines while keeping resource usage within 5% of the baseline.

Chapter 2

Background and Related Work

This chapter describes background and related work regarding serverless computing, deep learning inferences on serverless, and the three optimization aspects developed in Flashpoint: model partitioning, locality-aware autoscaling, and network multicasting.

2.1 Serverless Computing

Serverless computing is a computing paradigm that has emerged in recent years as a response to the growing demand for scalable, flexible, and cost-effective computing solutions. In a serverless environment, the cloud provider is responsible for managing the infrastructure and automatically allocating resources based on the needs of the application. This eliminates the need for businesses to invest in and maintain their own hardware, freeing them up to focus on delivering value to their customers.

In traditional computing models, the key challenge is the difficulty of allocating resources in a way that is both performant and cost-effective. In traditional environments, resources are statically provisioned in advance based on estimated needs, but this often leads to idle resource waste during low demand and missed requests during high demand. In most cases, due to the dynamic nature of user-facing services, spikes and troughs in demand lead to both over-provisioned resources and long delays from under-provisioning in static allocations [8].

Serverless computing addresses this challenge by providing a pay-as-you-go model that automatically scales resources based on the actual needs of the application. This eliminates the need for businesses to perform regular workload measurements and frequent hardware purchases or sales, freeing them up to focus on delivering value to their customers without having to worry about the underlying infrastructure. While serverless computing offers many benefits, it is not without its challenges. One of the main barriers to wider adoption of serverless computing is the issue of "cold starts," which refer to the latency introduced when a serverless function is invoked after a period of inactivity. Cold starts can result in slow response times, particularly for latency-sensitive workloads.

Several studies have shown that cold starts can be a significant barrier to the adoption of serverless computing, as they can negatively impact the performance of applications including user-facing chatbots and reduce the quality of user experience [45, 105]. To address this issue, current serverless systems trade off cost-effectiveness for performance, providing a balance between the two that is suitable for most workloads [66]. However, for low-latency workloads, such as those in user-facing chat applications, financial, gaming, or real-time data processing industries, the trade-off between cost-effectiveness and performance can be a challenge because of long cold starts, which lead to an inelastic system.

2.2 Deep Learning Inference Serving on Serverless

In recent years, deep learning has become an important area of research and development in the field of machine learning, with numerous applications in areas such as image and speech recognition, Natural Language Processing (NLP), and predictive modeling. As deep learning models continue to grow in complexity and size, serving these models in a production environment become a major challenge.

One of the biggest challenges in serving deep learning models is the requirement for GPU resources, which are essential for processing the large amounts of data and performing the complex computations required by these models. While GPUs have become increasingly powerful in recent years, serverless platforms using GPUs are still limited in their ability to support large-scale deep learning workloads, particularly when it comes to serving large numbers of requests in real-time.

While containerization [2] and memory sharing techniques [32, 60] have reduced cold start durations for general-purpose CPU functions, such methodologies are not applicable in GPUs due to lack of support for sequential control flow. These limitations have made it difficult to find a solution that is both performant and cost-efficient for GPUs in serverless computing. Current serverless platforms, such as AWS Lambda, do not support GPU processing. On the other hand, while AWS SageMaker supports GPU usage for inferences, it recommends an over-provisioning of 2, which leads to increased costs [62, 61].

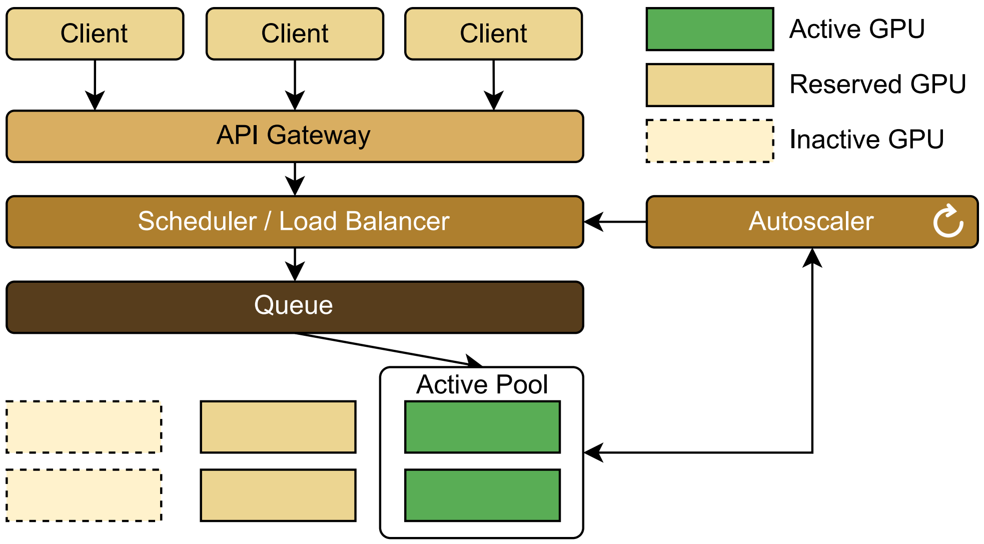


Figure 2.1: Overview of system architecture with autoscaling.

Recent studies on Deep Learning inference serving on GPUs [111, 81, 21] use CPUs in conjunction with GPUs to reduce costs while meeting Service Level Objective (SLO) requirements such as having 98% of requests below 200 milliseconds. Meanwhile, [106, 95, 76] adaptively batch requests to make GPU usage more efficient within SLA requirements. Despite the techniques, the serverless platforms still face the intrinsic trade-off of cost-effectiveness and performance.

In this thesis, we examine the use of serverless computing for deep learning inference serving, exploring the challenges and limitations of using serverless platforms for low-latency workloads and propose novel solutions that address the problem of cold starts in GPU-enabled serverless platforms.

2.3 System Architecture for Deep Learning Inference Serving on Serverless

Fig. 2.1 shows an overview of system architecture with autoscaling in open-source products such as Kubernetes and OpenWhisk [86, 1, 69]. Requests are sent by clients to the system's API gateway, which are then forwarded to the scheduler. Omitting authentication steps, the request is then sent to a message queue. Afterwards, active workers then pull messages from the queue whenever they are available. An active worker is a compute resource that has the requirements initialized (e.g. the DL model) and is available when it is not processing a request. While this occurs, workers update their status to the autoscaler. The autoscaler then runs an independent event loop to decide whether to scale up nodes based on metrics used by a configured autoscaling policy. Upon scaling up or down, the autoscaler triggers the initialization of inactive nodes and updates the scheduler when they are ready.

2.4 Model Partitioning

2.4.1 Automated Model Partitioning

Model partitioning is an active area of research in recent years with a focus on enabling the execution of large DL models on multiple machines. Previous studies such as [110, 65, 54, 47] have primarily focused on latency optimization and cost optimization with SLA-awareness for execution, but have not considered the impact of cold starts leading to long wait times. In contrast to existing works, we use model partitioning in Flashpoint to optimize cold start durations while minimizing degradation in execution latency due to intermediate data transfers. Flashpoint thus reduces long waiting times observed in state-of-the-art systems and adapts the partitioning scheme used based on the workload experienced by the system. Additionally, in comparison to the state-of-the-art, Flashpoint utilizes long-running compute instances rather than AWS Lambdas, as GPU execution is not supported in the latter.

This section was moved from the model partitioning chapter to here. Meanwhile, the idea of separating a deep learning model into sequential layers can seem trivial. Previous works in [110] and [54] have only been used in computer vision which primarily have simple types of convolutional layers, pooling, and fully connected layers. These have straightforward input to output mappings. However, with the Natural Language Processing (NLP) DL models researched in this thesis, the inputs and and outputs contain multiple strings, embedding tensors, and other tensors with non-trivial output to next-layer input mappings. In general Machine Learning frameworks such as PyTorch, ONNX, and other Machine Learning frameworks, DL models are constructed as a series of connected computation modules. These modules can have varying levels of modules inside them. This affects how DL models can be partitioned. In this thesis, we set a max module depth of 4 to break down high-level embedding, encoder, and decoder modules into smaller components. This level of depth results in a reasonable trade-off of complexity and cold start benefits in the tested models. Flashpoint then provides this interface which defines the model layers and transformations of outputs of a layer to the inputs of the following layer. This enables arbitrary DL models to work seamlessly with the automated partitioning process. However, acquiring the individual layers of a general DL model and input / output transformations is non-trivial and requires manual intervention in the state of the work. Once these layers and transitions are correctly defined, then the Flashpoint system can perform automated partitioning.

2.5 Locality

2.5.1 Centralized Registry

Replication in cloud computing is a technique used to horizontally scale the system's capacity by creating multiple copies of a service or application and distributing them across multiple nodes. However, current systems are limited in that replicas are treated as individual instances and unaware of DL model copies in other machines. To download the DL model in cold starts, centralized registries are used and have shown to result in network bottlenecks [99].

In [99], many clients have shown to request data from the registry, leading to congestion and delays in the network. This is a major challenge in cloud computing, as it can severely affect the performance and reliability of the system. [99] attempts to address the solution by structuring replica in a binary tree for container downloads. However, even with this approach, host nodes still suffer from bandwidth limitations in their network links. We have observed, for example, in our experiments that in the case of sending a DL model from a source host machine in the data center to multiple other host machines, the uplink bandwidth of the source host machine becomes congested with multiple unicast network streams.

2.5.2 Hierarchical Caching

In TrIMS [25], a cache-inspired mechanism is proposed to ship a DL model from CPU to GPU with fallback to cloud storage. TrIMS relies on CUDA Inter-Process Communication (IPC) to create new processes which reference shared GPU memory in the same GPU device. This is not used to scale the system's capacity for serving the model. GPU-enabled FaaS [117] proposes locality-awareness at the request scheduling level instead of replica autoscaling. It does this by comparing and selecting the best of the estimated completion time of requests if waiting for an active GPU to be available versus reloading the process on a GPU that has the DL model data in memory. Compared to TrIMS, GPU-enabled FaaS proposes selecting GPUs with the DL model data in the host machine instead of selecting the lowest utilization GPU, which may not have the DL model data in the host machine [117].

In contrast, Flashpoint reduces cold start durations at the replica autoscaling level by utilizing locality further in two ways. Firstly, under the common structure of data centers which have multiple GPUs per host machine, DL model data may not have been loaded in the memory of a GPU when the data is available in the host machine. By prioritizing these GPUs over those where the host machine does not have the data in memory, network data transfers are eliminated and cold start durations are reduced. Secondly, when no host machine has the data, Flashpoint leverages the stored memory of the DL model data from other host machines in the compute cluster, avoiding the slow download path of cloud storage. These approaches significantly improve system elasticity for more common scenarios in a bursty workload.

Meanwhile, FaaSNet [99] uses a binary tree to route function container downloads across virtual machines. In contrast, our proposed methodology uses a compute-colocated distributed registry that provides greater fan-outs than binary trees. Our proposed methodology also introduces locality-aware autoscaling which maximizes the benefit of hierarchical copies as well as increasing download efficiency in burst instantiation requests through download sharing.

2.5.3 Remote Memory Pooling

In Atoll [90] and Lin et al. [59] a proactive memory pool is used to pre-allocate memory of pre-initialized sandboxes in order to reduce sandbox initialization overhead. The memory footprint of this memory pool is configured by an administrator. The sandboxes are used within the worker as cores become available and requests are received. In comparison, we adaptively use excess resources in host machines to constitute the memory pool. Additionally, we use the DL models stored in the remote memory pool as a hierarchical source for downloading to avoid the slowest path of downloading from cloud storage instead of just executing inferences on the same machine.

2.6 Network

2.6.1 Multicasting

Multicasting, or the ability to simultaneously transmit data to multiple recipients, is not currently used in serverless environments due to the challenge of performing cold starts individually. Instead, individual unicast streams are used to transmit data to each recipient. However, the development of protocols such as one by Khooi et al. [52] is a promising direction using Remote Direct Memory Access (RDMA) with Software-Defined Networking P4 switches to perform multicasts in a fast and reliable manner. This work is inspired by this potential to be implemented in serverless environments to improve performance and scalability.

2.7 Autoscaling

2.7.1 Trade-offs of Modern Autoscaling

Current state-of-the-art systems such as [63, 111, 11, 26] use a mix of "serverless" or "burstable" Function-as-a-Service (FaaS) resources which have shorter startup times but are generally higher cost, and "serverful" or Infrastructure-as-a-Service (IaaS) resources with longer cold start durations and lower long-term costs, to reduce costs while serving a workload and meeting pre-defined end-to-end request latency targets such as Service Level Objectives (SLOs). Meanwhile, BATCH [4] and INFless [106] use adaptive batching to reduce resource costs within user-defined SLOs. On the other hand, SHEPHERD [112] aggregates multiple request streams for DL models to reduce workload unpredictability. Additionally, ENSURE [93] uses a square-root staffing policy to over-provision resources to meet high levels of performance, sacrificing cost-effectiveness. These systems achieve guarantees that trade off performance for cost-effectiveness [66] or optimizes resource sharing when multiple streams can use the same GPU resources. In contrast, we show that Flashpoint's model partitioning, locality-aware autoscaling, and network multicasting techniques improve the performance of the system at all levels of cost-effectiveness by tackling the underlying problem of long cold start durations and achieving greater system elasticity.

2.7.2 SLA-driven Scaling

Works such as [111, 81, 21] all adaptively use GPUs to meet SLA requirements, while other works such as [106, 95, 76] focus on adaptively batching requests to make GPU usage more efficient within SLA requirements. All of these works also use a variety of metrics including CPU and GPU utilization, profiled inference time, queue length and delays to inform scaling decisions. In comparison to the related works, our autoscaling placement decisions are informed by network structure, locality information, and profiling for model partitioning. Additionally, Flashpoint reduces cold start durations in any autoscaling policy with locality and network optimizations. These parts can provide benefit in conjunction with the aforementioned works.

Chapter 3

Measurement Study of Cold and Warm Starts on GPUs and CPUs

In this chapter, we perform a measurement study to quantify the magnitude of the cold start problem based on the duration of GPU and CPU cold starts and warm starts. The rapidly growing demand for interactive DL services makes efficient use of scarce GPU resources an important problem to solve. A key way to achieving elastic use of these resources is to reduce GPU cold start durations. Thus, our motivating research question for the overall thesis is "How can the duration of GPU cold starts in serverless DL inference serving systems be reduced?"

We then divide the analysis of the problem with the following research questions:

RQ1. What are the performance differences of CPUs and GPUs for DL inference serving?

RQ2. How much do long cold start durations impact system performance?

RQ3. What are the mathematical factors that impact system performance for serverless computing?

3.1 Experimental Configuration

The following experimental configurations were used in this measurement study:

Table 3.1 Overview of DL Models

Name	**Version**	**# Params**	**Size**
CodeBERT	Base	125M	499 MB
ALBERT	XXLarge V2	223M	890 MB
BART	Large	406M	1,626 MB
DialoGPT	Large	774M	3,135 MB
GPT-2	XLarge	1,558M	6,282 MB
T5	3B	∼2.8B	11,408 MB

Deep Learning Models. Table 3.1 shows the six popular DL language models used in this measurement study. These are representative of various model sizes in the domain of Natural Language Processing (NLP). The models used include CodeBERT [34], ALBERT [55], BART [56], DialoGPT [116], GPT-2 [72], and T5 [73], which range from 499 MB to 11 GB in size. By testing with these models, we show that our observations hold for various model sizes.

Workload. The simulation uses 1 hour long traces of realistic workloads provided by Twitter [96] and Microsoft Azure in 2019 [87]. These public workloads are used due to the lack of public DL inference serving traces. However, the user-facing nature of the related products are expected to result in similar access patterns to low-latency DL services such as ChatGPT and GitHub Copilot. The 1 hour Twitter trace has a median of 57 requests per second and peak-to-median ratio of 4.68, while the 10 hour trace has a median and peak-to-median ratio of 47 and 6.19 respectively. The Microsoft Azure trace has a median of 18 requests per second and a peak-to-median ratio of 2.54. For profiling, we use the inputs from the code to natural language test dataset from [34].

Baselines. We use PyTorch deployed using its inference serving framework, TorchServe, on Kubernetes (TSK) [1] as a baseline. We also compare this work with proactively-provisioned memory pooling techniques proposed in Atoll (ATL) [90], OpenWhisk on Kubernetes (OWK), and our implementation of an AWS SageMaker autoscaling policy (SM*) [29].

The TorchServe on Kubernetes (TSK) baseline uses queue latency as its autoscaling target metric with a default target of 7 seconds. Atoll (ATL) similarly uses queue latency as a metric for autoscaling as well as proactively provisioning resources based on the arrival rate of requests. In the experiments, Atoll is set to use the arrival rate of requests. OpenWhisk on Kubernetes (OWK) relies on the Kubernetes horizontal pod autoscaler, which uses the CPU compute utilization of host machines with a default target utilization of 60%. In this thesis, this is modified to use GPU compute utilization. Finally, AWS

SageMaker (SM*) uses compute utilization, invocations per instance, or a custom metric for its autoscaling policy. For this research, it is set to use invocations per instance, which is the number of requests served by each replica on average.

For model partitioning, we use baselines of Gillis (GLS) [110] and SerFer (SFR) [54] as the most related works. Both Gillis and SerFer limit the size of DL models to fit in a Lambda, which is 250MB [110, 54]. Gillis attempts to fit the memory limits while minimizing network transfers. Consequently, it also minimizes the number of partitions used, leading to longer cold start durations. Meanwhile, SerFer solves for partitions that fit these limitations with no consideration for the network transfer costs incurred and leads to more partitions than Gillis and Flashpoint.

Simulation. This thesis uses simulations to measure the impact of the work. The simulator is built using a customized version of faas-sim [75], which simulates network congestion through the framework, Ether [74], and is built on top of the Python simulation framework, SimPy [84].

It uses a fixed number of GPUs (1600) available within a data center with 8 GPUs per host machine. The structure of the network used is a spine-leaf architecture with 100 Gbit/s links from spine to leaf nodes and 50 Gbit/s links from leaf nodes to host machines with 20 host machines per leaf node. This work assumes homogeneous GPUs for simplicity and could be modified to incorporate the variety in compute capacity for heterogeneous GPUs. The simulation incorporates the time to perform autoscaling and scheduling decisions. This means that the actions do not take effect until time is incremented by the wall time it took to make the decisions.

Based on the Kubernetes baseline, each host machine is set to have the PyTorch base Docker image pre-loaded, meaning that a web server with PyTorch dependencies loaded is already running on each host machine.

Testbed. All characterizations in this chapter use the following setup unless otherwise specified. We use a cluster of 3 host machines that each have 40 Intel® Xeon® Silver 4114 CPUs and one additional dedicated workload generator machine with the same specifications. Each host machine has 2 Nvidia Tesla P40 GPUs attached for a total of 6 GPUs. The testbed uses 25 Gbit/s network links. For all characterizations of AWS S3 downloads, a p3.2xlarge EC2 instance with a single C++ S3 client is used in the us-east-2 (Ohio) region with a single AWS S3 bucket using in the same region. The EC2 instance has 1 Nvidia V100 GPU with 16 GiB of memory, 8 vCPUs with 61 GiB of memory, and up to 10 Gbit/s network link.

Confidence Intervals. All confidence intervals, wherever included, are 95% over 5 trials.

3.2 Performance differences of GPUs and CPUs

3.2.1 GPU Inferences are Faster than CPU Inferences

In Sections 3.2.1 and 3.2.2, we tackle **RQ1** "What are the performance differences of CPUs and GPUs for DL inference serving?" by measuring the difference in GPU and CPU inference and cold start durations. We show that cold start durations are several orders of magnitude larger than execution times and are made worse by longer GPU cold starts and faster GPU inferences.

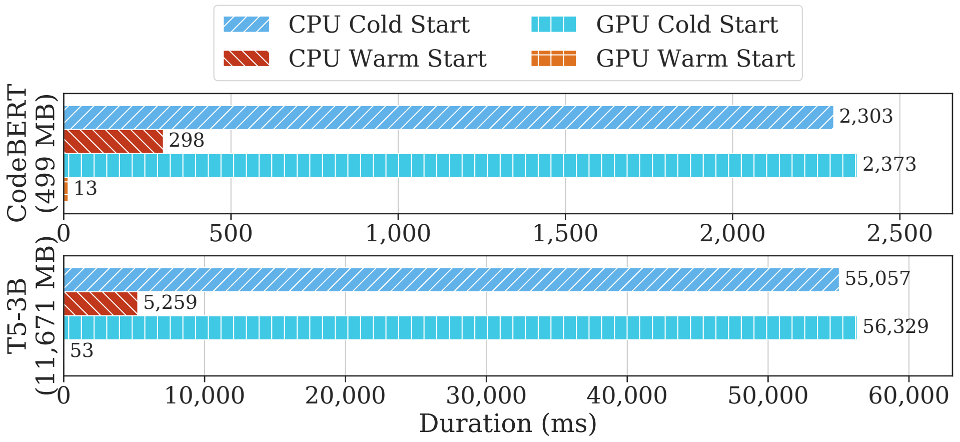


Figure 3.1: Duration of CPU and GPU warm starts compared to CPU and GPU cold starts.

As a natural consequence of computation parallelization in GPUs, inferences complete quicker in GPUs than with CPUs. Fig. 3.1 compares CPU and GPU inference times for different DL models. The data is collected by initiating cold starts for each model on a single host machine. The warm start excludes input and output transfer steps, while the cold start excludes the warm start. The results on average across DL models shows that CPU inferences take 53.15× longer than GPU inferences. The shorter GPU inferences motivates the use of GPUs in modern DL inference serving systems to meet low-latency deadlines. We also observe through the characterization that the execution latency of DL models are predictable, which is consistent with the findings in [39, 24].

3.2.2 GPU Cold Starts are Prohibitively Longer than Warm Starts

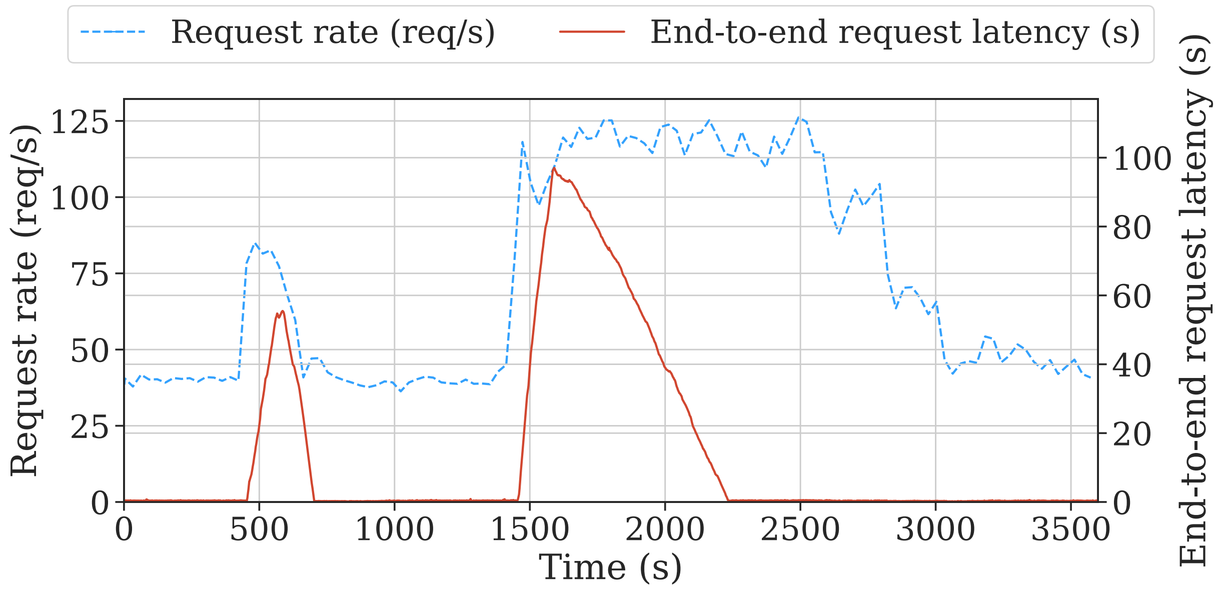
The benefits of faster GPU inferences are diminished by longer cold start durations in GPUs. Fig. 3.1 also compares the ratio of warm and cold start durations for GPUs and CPUs for the largest and smallest DL model. The GPU cold start is measured by downloading the model from the AWS S3 bucket in the testbed. GPU cold starts are longer than CPU cold starts due to the additional step of sending the model to the GPU. The characterization shows that, using the mean across DL models, GPU cold starts are 581.02× longer than GPU inferences while CPU cold starts are 10.71× longer than CPU inferences. This emphasizes the need for reducing cold start magnitudes in GPU-powered serverless systems.

3.3 Impact of Long Cold Starts on Performance

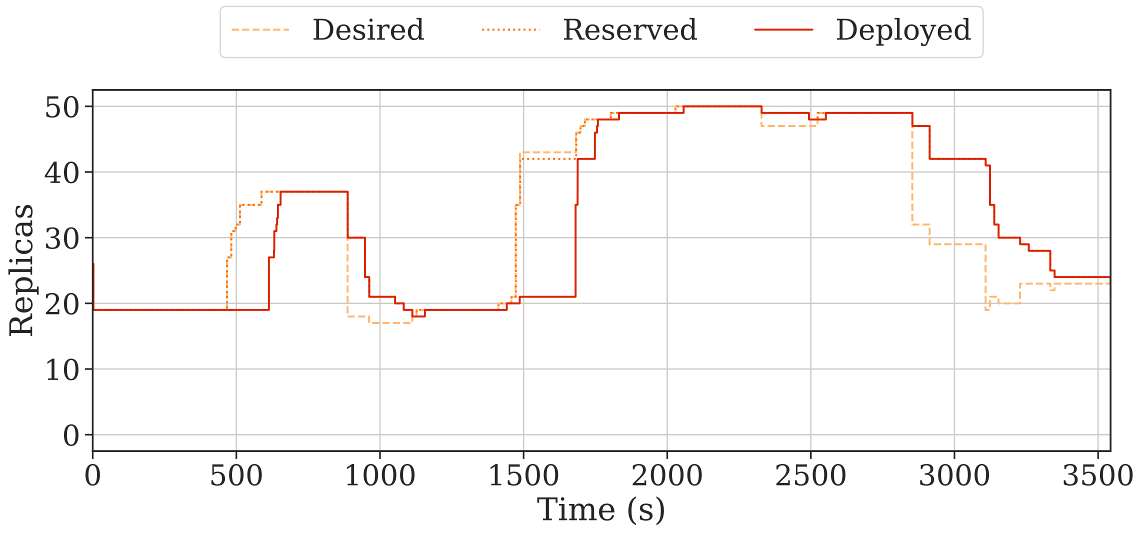
3.3.1 Workload Timeline Example

Fig. 3.2a shows the end-to-end latencies of requests at different request received times in 1 hour of the Twitter trace. The experiment shows results from the simulation of the full scale workload using the baseline Atoll autoscaler, which uses request rate. The practicality of the simulation is discussed and validated later in Chapter 4. Meanwhile, Fig. 3.2b shows the number of replicas desired, reserved, and deployed over the course of the workload. The figures, together, show that bursts of requests consistently lead to long waiting times due to new replicas not being instantiated quickly. The result shows that the burst of requests at 456 seconds causes an excess load that is not met by the current service capacity of 19 replicas, leading to the long tail of requests. In this overloaded state, later requests experience longer waiting times due to the buildup of the request queue. Then, during the burst, new replicas complete their initialization, leading to rapid consumption of requests in the queue and the drop in end-to-end request latencies after the peak. This example highlights the importance of fast replica initialization in order to reduce the number and magnitude requests in the long tail in terms of end-to-end request latencies.

In the example, a second burst occurs at 1,462 seconds. Prior to this, the relatively stable request rates before it has established a low number of deployed replicas. The burst then creates another local peak in end-to-end request latencies. While current systems opt to use scale down stabilization mechanisms which use a configurable window to prevent quickly scaling down, the optimal configuration for the window is dependent on the volatility of the workload and can lead to high amounts of idle resources. Instead, reducing



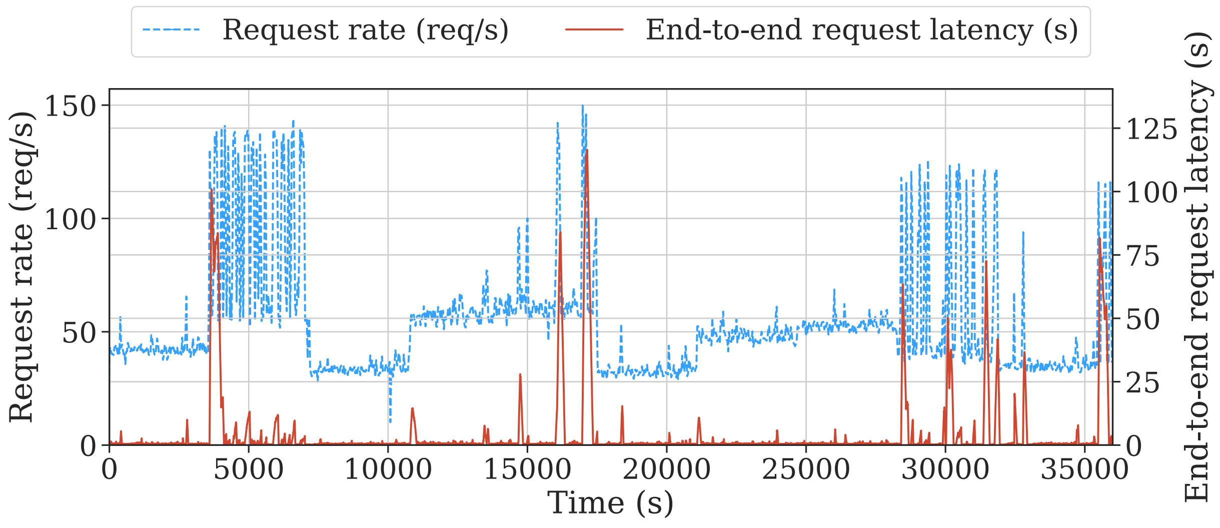
(a) End-to-end request latencies of requests compared to the arrival request rate.



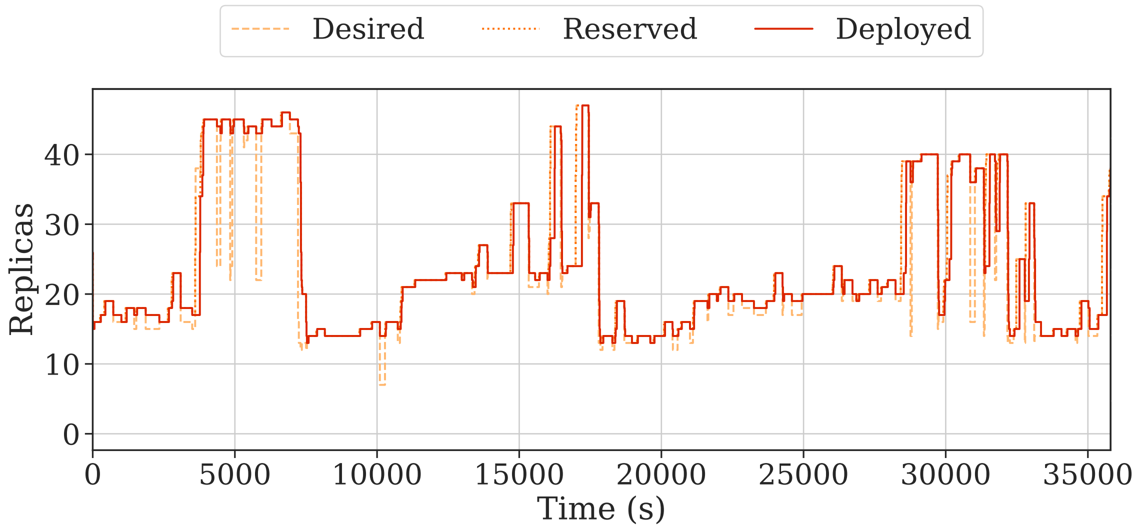
(b) Number of replicas desired, reserved, and deployed.

Figure 3.2: Simulated results for a 1 hour Twitter workload using the baseline system.

the duration of cold starts can make the system more elastic and adapt to any workload without relying on workload-specific configuration.



(a) End-to-end request latencies of requests compared to the arrival request rate.



(b) Number of replicas desired, reserved, and deployed.

Figure 3.3: Simulated results for a 10 hour Twitter workload using the baseline system.

Fig. 3.3 shows another example similar to Fig. 3.2 for a 10 hour trace of the Twitter workload. Fig. 3.3a and Fig. 3.3b respectively show the end-to-end request latency of requests and number of replicas desired, reserved, and deployed over the course of the workload. The result reinforces that bursts in request rates which occur after a period of stability lead to scenarios of under-provisioned resources where requests incur waiting time and longer end-to-end request latencies. This shows that while the baseline system may use stabilization mechanisms to prevent scaling down, sufficiently long periods of stability before a burst of requests will cause the system to evict enough replicas to cause long waiting times.

3.3.2 Cold Starts Manifest as Long Waiting Times

In this subsection, we address **RQ2** "How much do long cold start durations impact system performance?" and measure the impact of long cold start durations to the performance of the system in terms of end-to-end request latencies.

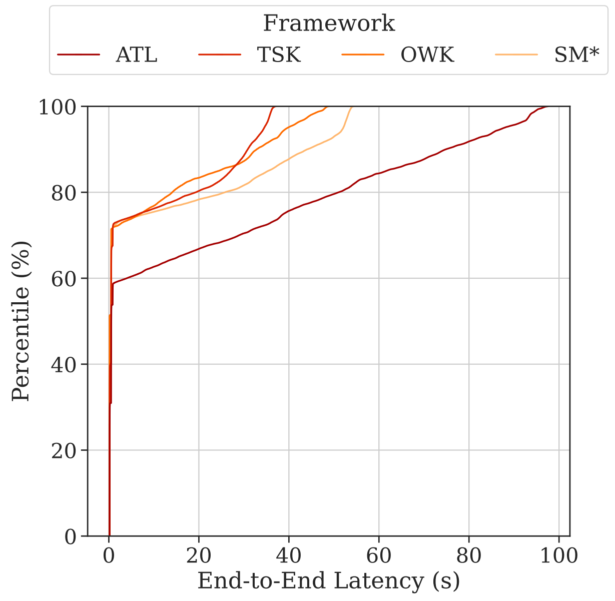


Figure 3.4: Simulated end-to-end request latency cumulative distribution function (CDF) of the baseline system with different autoscalers for the Twitter workload using the T5-3B model.

The time spent by requests waiting in the system during bursts in requests is largely determined by the speed that replicas can be instantiated on new machines. Fig. 3.4 shows the end-to-end request latency cumulative distribution function (CDF) for 1 hour of the Twitter workload for the T5-3B model with the baseline system using different autoscalers. The figure shows that the tail latency (99th percentile) of requests can be up to 674× longer than the inference time of the DL model. The characterization also shows that various autoscaling policies lead to an average of 147.48, 50.22, 51.80, and 69.91 seconds for the baseline frameworks ATL, TSK, OWK, and SM*, respectively. The mean of these values, 11.52 seconds is 82× as long as the average execution time. For each framework, their ratio of average end-to-end request latencies to execution times are 147×, 50×, 52×, and 70× respectively. These long tail latencies are attributed to both long cold starts causing delays in provisioning resources as well as long default periods in autoscaler metric evaluation. This validates the impact of long cold starts in real workloads and further motivates reducing cold start durations.

3.4 Mathematical Model for GPU Cold Starts and Performance

3.4.1 A Thorough Analysis of GPU Cold Starts

In this subsection, we approach **RQ3** "How can we mathematically represent system performance for serverless computing?" by first identifying the relative duration of different stages in the cold start process. We show that each stage in the cold start process must be reduced to achieve sub-second cold starts.

We identify three major steps in the cold start process:

1. Downloading the DL model
2. Loading the model on the host machine
3. Sending the model to the GPU

In this thesis, we define the startup time to be comprised of the latter two steps. Upon completion of all of the steps, a process has been initialized in the GPU to accept requests and perform inferences.

Fig. 3.5 shows the absolute duration of cold start steps for the smallest and largest DL model, CodeBERT and T5-3B respectively. In the process of the cold start, the DL model has already been trained. Its weights and layer information are serialized into a binary format readable by the tensor module. In our experiments, we use PyTorch's Just-In-Time (JIT) trace format. The steps in the cold start process from start to finish can be broken

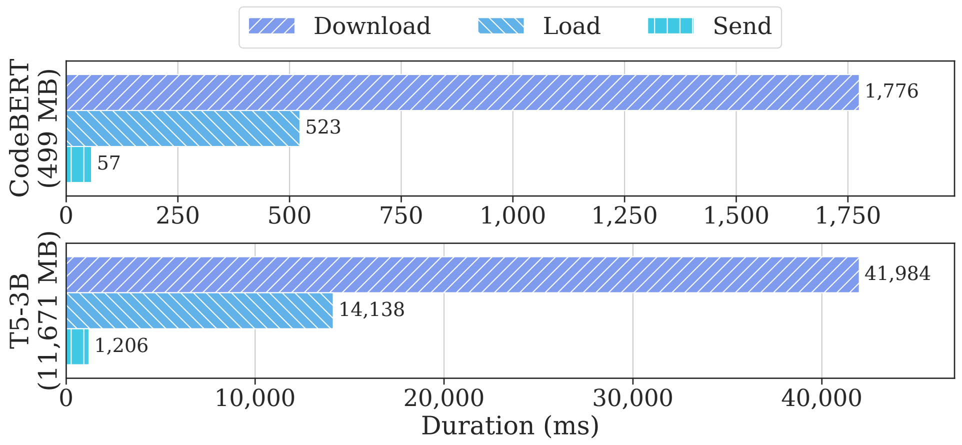


Figure 3.5: Detailed breakdown of GPU cold start durations.

down to the three steps. Using the example for the T5-3B model, the DL model binary data is first downloaded into host machine memory in approximately 41 seconds. Then, the binary data is loaded into usable tensor modules in the same host memory for about 14 seconds. Finally, the tensor modules are sent to the GPU in nearly 1 second. After these three steps are complete, the host machine is ready to perform inferences using the GPU. The experimental configuration for this test is to use a Lambda Labs machine with an Nvidia A10 GPU. The effective network bandwidth observed when downloading the model from AWS S3 in the same region is 2,203 Mbps. In comparison, we measure the effective network bandwidth of 2,936 Mbps for downloading from AWS S3 in the same region as a p3.2xlarge EC2 instance which is rated to up to a 10 Gbit/s network link.

Fig. 3.6 shows the mean of durations for each step in the cold start process across DL models. The values are normalized to the GPU warm start of the DL model. The characterization uses an AWS S3 client to download the model into a Lambda Labs machine with an A10 GPU in the same region. The data is a generalization of data shown in Fig. 3.5.

The characterization shows the first step being 581.2× longer than the GPU warm start, while the second and third steps are 214.0× and 18.0× the GPU warm start respectively. Overall, the first step of downloading the model is over 71.46% of the whole cold start duration and is thus a focus of this work. This indicates that downloading the DL model from cloud storage is prohibitively slow due to network bottlenecks.

Meanwhile, Fig. 3.6 also shows that the load and send to GPU step are sizeable

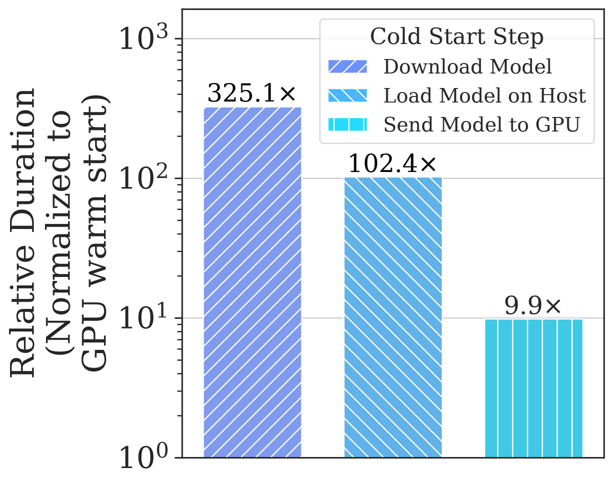


Figure 3.6: Relative duration of steps in a cold start compared to the warm start of the DL model (mean across DL models).

durations compared to real-time latencies and human reaction times of 1,000 ms [111, 77]. For example, the T5-3B model takes 14,138 ms to for the load step while it takes 1,206 ms to send the model to the GPU. In order to scale elastically, these load and send steps also have to be reduced.

3.4.2 Problem Formulation

In this subsection, we tackle **RQ3** "How can we mathematically represent system performance for serverless computing?" and develop an analytical model that mathematically relates different steps of the cold start process to the end performance metric of average end-to-end request latencies. This model is then used to identify relevant factors in minimizing cold start durations and compare their relative impact on system performance.

In this section, we describe the behaviour of the system and formulate the problem as an analytical model derived from queuing theory. We use the notations described in Table 3.2. Firstly, we relate the expected duration of cold starts to the expected end-to-end latency of requests in the system. Then, we identify variables that affect the duration of cold starts, which highlight opportunities to address the issue.

To model the behaviour of the system, we develop a state transition Semi-Markovian Process (SMP) shown in Fig. 3.7 inspired by [66]. Fig. 3.7 shows a state transition diagram for the system having m number of instantiated GPUs for a DL model up to a large number

Table 3.2 Table of Notations

Notation	Description
t	timestamp
T	time interval for autoscaling event loop
m	number of GPU instances with DL model
λ	arrival rate of requests
μ	service rate of GPUs for a DL model
d_e	duration of execution
d_c	duration of the whole cold start
$d_{c,d}$	duration of the download step of a cold start
$d_{c,l}$	duration of the load model on host step of a cold start
$d_{c,s}$	duration of the send model to GPU step of a cold start
d_w	duration of waiting for the request
n_d	number of desired replicas in the system
r	arrival rate of requests
σ	volatility of workload
c_u	compute utilization for a DL model inference as percent of GPU
q_l	queue length for the DL model
l_t	end-to-end latency target

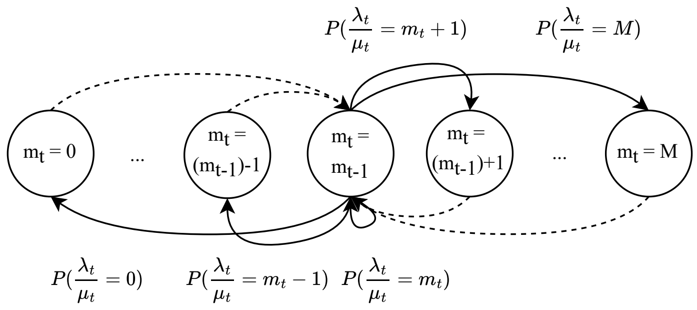

$P(\frac{\lambda_t}{\mu_t} = m_t + 1)$
$P(\frac{\lambda_t}{\mu_t} = M)$
$m_t = 0$
...
$m_t = (m_{t-1})-1$
$m_t = m_{t-1}$
$m_t = (m_{t-1})+1$
...
$m_t = M$
$P(\frac{\lambda_t}{\mu_t} = 0)$
$P(\frac{\lambda_t}{\mu_t} = m_t - 1)$
$P(\frac{\lambda_t}{\mu_t} = m_t)$

Figure 3.7: State transition diagram for the number of instantiated GPUs m for a DL model at a time t.

M at any time t for a discretized time intervals of T . At each time interval, the system has a probability $P(\frac{\lambda_t}{\mu_t} = x)$ to have a desired number of replicas x which can be represented relative to the current number of running instances m_t. The arrival rate λ_t represents the total number of requests arriving into the system at time window t. Meanwhile, μ_t is the service rate of the system at time t defined by (3.1) where $c(f)$ is the maximum number of requests an instance can execute concurrently for a function f.

$$\mu_t = m_t * c(f) \tag{3.1}$$

The definition of $c(f)$ is elaborated in (3.2) where g is the GPU and f is the function.

$$c(f) = min(\frac{s_{memory}}{f_{memory}}, \frac{s_{compute}}{f_{compute}}) \tag{3.2}$$

The probabilities depend on the arrival rate profile λ, which we assume to be a Poisson process with its mean varying over time [4, 112].

When the desired number of replicas for the time window t is greater than the current number of instantiated GPUs in the system, the cold start process for a number of GPUs begin. Due to the non-zero time to instantiate GPUs, sustained excess demand for a function causes an accumulation of requests queued in the system. Assuming that the excess demand is sustained for the duration of the cold start such that $t' = t + d_c$, which represents a worst case scenario, the average number of requests in the system L is defined in (3.3) where $\mathbb{E}(t_c)$ is the expected cold start duration in the system.

$$L = \lambda_t \cdot \mathbb{E}(d_c) - \mu_t \cdot \mathbb{E}(d_c) \tag{3.3}$$

Meanwhile, the end-to-end latency in the asynchronous autoscaler model can be represented as shown in (3.4) as the sum of execution time d_e and wait time d_w. For deep learning models, d_e is roughly constant.

$$\mathbb{E}(d_{e2e}) = \mathbb{E}(d_e) + \mathbb{E}(d_w) \tag{3.4}$$

Applying Little's Law, which is $L = \lambda W$ for the number of requests in the system L, effective arrival rate λ (throughput), which in our case is $\mu_{t'}$, and waiting time W, we combine (3.3) and (3.4) to achieve the following result for the waiting time during the stage of excess demand in (3.5), valid for $\mu_{t'} > \lambda_{t'}$. This equation shows that the average end-to-end latency is proportional to the excess demand as well as expected duration of the cold start. The equation also indicates that although the cold start duration may be large, the longest observed waiting time for a request is not necessarily as long as the cold start duration depending on the ratio of the arrival rate to the service rate.

$$\mathbb{E}(d_w) = (\frac{\lambda_{t'}}{\mu_{t'}} - 1) \cdot \mathbb{E}(d_c) \tag{3.5}$$

Upon scaling up, the system must scale up to a service rate greater than the arrival rate in order to consume messages in the queue, leading to a reduction in queue length.

Meanwhile, as shown in the characterization, the duration of cold starts are proportional to the ratio of cold starts and is modelled by (3.6) where r is the ratio of instances to be created to current instances shown in (3.7), while $d_{c,d}$, $d_{c,l}$, and $d_{c,s}$ are the expected download, load on host, and send to GPU durations of the cold start respectively.

$$\mathbb{E}(d_c) = r * \mathbb{E}(d_{c,d}) + \mathbb{E}(d_{c,l}) + \mathbb{E}(d_{c,s}) \tag{3.6}$$

$$r = \frac{m_{t+T} - m_t}{m_t} = \frac{\lambda_t}{c_f \cdot m_t} - 1 \tag{3.7}$$

The problem formulation in this section shows two main properties that we aim to address. First, longer expected cold start durations lead to longer average end-to-end latencies. Second, larger cold start bursts lead to longer cold start durations due to network congestion.

3.5 Measurement Study Summary

We identified that GPU cold starts are orders of magnitude longer than the actual inference times and that this difference is greater than in CPUs. We then measured the download step for the DL model to comprise a majority of the cold start. We also showed that excess system resources exist and could be used to support shorter cold start durations. We have then showed that network congestion is a problem that extends cold start durations during frequent simultaneous cold starts. Afterwards, we presented an analytical model to formalize the problem of cold starts and identify opportunities for improvement. Furthermore, we investigated the properties of model partitioning with respect to cold start durations and end-to-end latencies, which have not been studied before. Finally, we identified the limitations of stabilization windows in current autoscalers. These were designed due to long cold start durations and we propose the removal of these mechanisms in light of sub-second cold starts achieved by Flashpoint.

Chapter 4

Model Partitioning

4.1 Characterization of Model Partitioning

In Section 3.4.1, we have identified that the cold start process leads to waiting times that are several orders of magnitude greater than execution times. In order to achieve elastic scaling with acceptable performance without overreliance on over-provisioning, the duration of cold starts has to shortened. To achieve these shortened cold starts, we investigate the use of model partitioning.

The technique of model partitioning is possible when multiple machines are selected for DL model initialization. The technique is to load different parts of the model in parallel across the machines, forming a co-dependent set of replicas. In comparison, the default method is to load a full, independent model on each of the machines. Model partitioning provides the opportunity of reducing cold start durations for downloading the model, loading it on the host machine, and sending the tensors to the GPUs due to its parallel nature. However, since the model is loaded in part on different machines, intermediate data between DL model layers must be transferred between the machines through the network to perform the inference, thus adding network transfer delays. With these important aspects of model partitioning, we use the following questions to guide the research in this chapter:

RQ4. What is the main cause for long DL model cold starts?

RQ5. How can the durations of cold start steps with GPUs be reduced?

RQ6. How does model partitioning affect system performance?

4.1.1 Main Cause of Long DL Model Cold Starts

In response to **RQ4**, we identify that the long cold start steps as measured in Section 3.4.1 are caused by the host machine needing to parse the entire archived DL model data into callable tensor modules and send those tensors to the GPU before it can be used for execution. With increasingly large DL model sizes, these steps grow longer, consequently making the system less elastic.

4.1.2 Methods to Reduce the Durations of GPU Cold Starts

For **RQ5** , we identify that model partitioning can be used to reduce cold start durations and reduce average end-to-end request latencies. This is because model partitioning divides the DL model into parts that can be loaded simultaneously on different machines, thus tackling the core issue of loading the whole model sequentially before it begins execution.

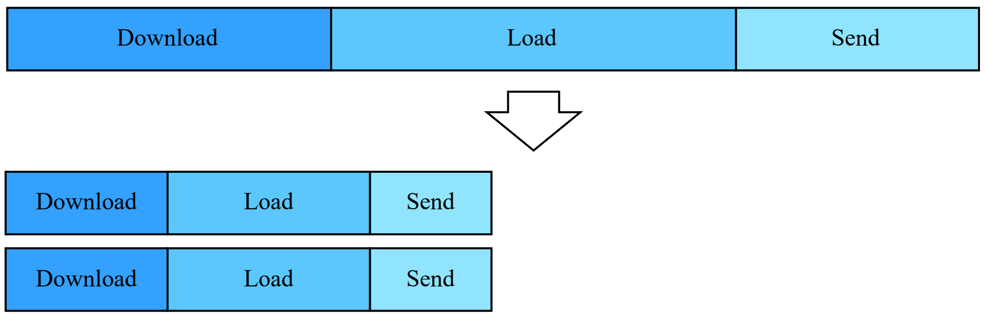


Figure 4.1: Example of parallelizing deep learning cold starts with model partitioning.

Fig. 4.1 shows the technique of model partitioning and its overall effect in reducing cold start durations and addresses **RQ6**. Model partitioning divides the DL model into individual parts that can be loaded simultaneously on different machines. This introduces the opportunity to reduce the cold start duration of the DL model to the maximum of the longest cold start duration of any individual part. The amount of data processed for each part is smaller than the whole. Thus, the durations for downloading the model, loading the model on the host machine, and sending the loaded tensors to the GPU are shorter for each part. By performing these in parallel, the cold start duration is reduced.

Yet, if the model parts are all downloaded from a single source, it can be bottlenecked by the network bandwidth from that source. By loading these individual parts to different destinations and using different source machines with independent download paths, we can ensure that the network uplink or downlink bandwidth of any individual machine is not a bottleneck for the modified cold start process.

In addition, compared to downloading multiple full replicas of the same DL model to different machines, a single copy of the DL model binary is downloaded across replicas in the partitioned model. This leads to lower load on the network, lower network congestion, and reduces the duration of downloads, which is the longest step in the cold start process.

However, the partitioned model incurs additional costs of transferring the output data of earlier parts as the input of the following parts in a series. This trade-off between shorter cold start durations and longer inference times should then managed.

4.1.3 Effects of Model Partitioning on System Performance

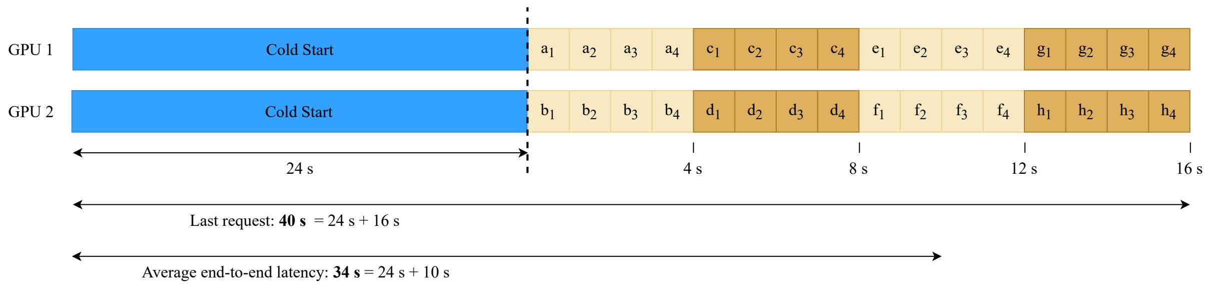
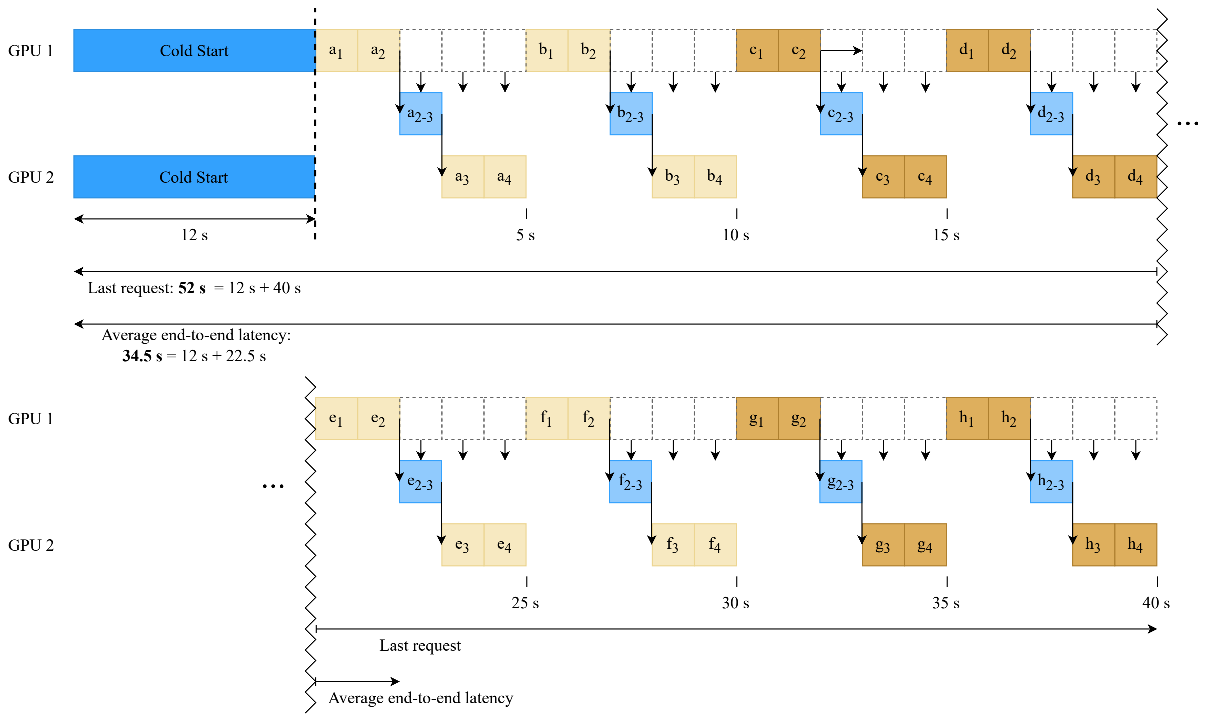

GPU 1
Cold Start
a_1 a_2 a_3 a_4 c_1 c_2 c_3 c_4 e_1 e_2 e_3 e_4 g_1 g_2 g_3 g_4
GPU 2
Cold Start
b_1 b_2 b_3 b_4 d_1 d_2 d_3 d_4 f_1 f_2 f_3 f_4 h_1 h_2 h_3 h_4
24 s
4 s
8 s
12 s
16 s
Last request: **40 s** = 24 s + 16 s
Average end-to-end latency: **34 s** = 24 s + 10 s

Figure 4.2: Example of 8 DL inferences on 2 GPUs without model partitioning.

We first approach **RQ6** by investigating the following examples.

The examples in this section will describe the effect of model partitioning on cold start durations, execution time, wait times, and end-to-end latencies. The examples use the following parameters:

1. A single request takes 4 seconds on the GPU indicated by 4 blocks.
2. The execution of a single model uses 100% of the GPU's compute capacity (i.e. the GPU can execute up to 1 concurrent request).
3. The goal is to complete 8 requests that are waiting in a message queue.
4. There are not enough GPUs to serve these requests, meaning that new replicas must be instantiated



GPU 1
GPU 2
Cold Start
Cold Start
12 s
5 s
10 s
15 s
Last request: **52 s** = 12 s + 40 s
Average end-to-end latency: **34.5 s** = 12 s + 22.5 s
25 s
30 s
35 s
40 s
Last request
Average end-to-end latency

Figure 4.3: Example of model partitioning without pipelining.

5. The full model cold start is 24 seconds, while a 2-part partitioned model divides the cold start into two equal parts, leading to a cold start duration of 12 seconds. This ratio of cold start duration for partitioned models are validated in our experimental results described later in Figure 4.5.

Fig. 4.2 shows that the execution of 8 inferences on two GPUs with the full DL model, which does not have any model partitioning, has an average request completion time is 34 seconds. This is done by each GPU executing requests with a single level of concurrency. This is 24 seconds for the full cold start and an average of 10 seconds for executing 4 requests (4 + 8 + 12 + 16 seconds divided by 4). Meanwhile, Fig. 4.3 shows an example of executing the same 8 inferences with model partitioning but without any pipelining. This means that the first GPU does not start executing the next request until the whole request is completed. In the example, the DL model is cut into two equal parts, and a network transfer delay (e.g. block a_{2-3}) of 1 second is incurred for every request execution. When the first GPU completes executing the first half of the DL model, the output tensor is then sent through the network to the second GPU, which executes the rest. Without

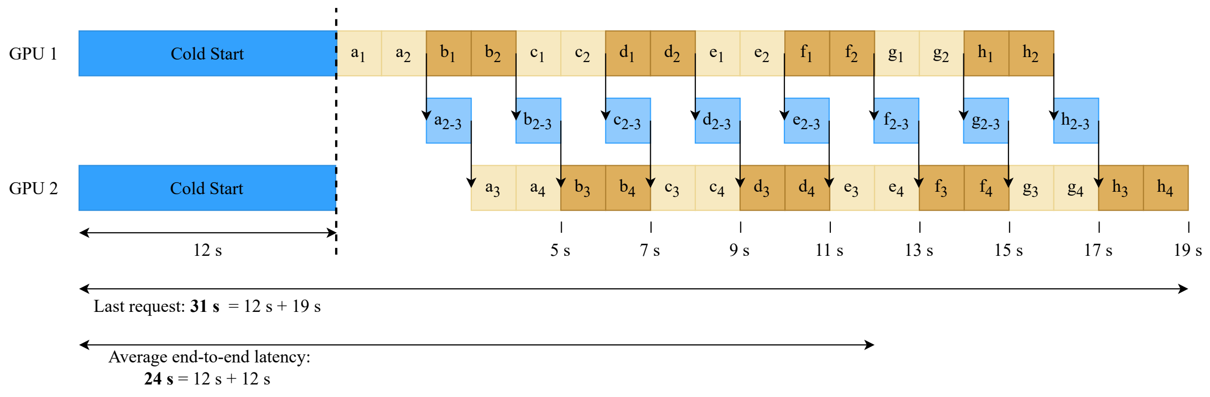


Figure 4.4: Example of model partitioning with pipelining. *Network transfer is performed using extra memory space in GPU

pipelining, subsequent requests do not begin until the previous requests on the GPU are completely finished. This results in completing the 8 requests in 40 seconds after the cold start of 12 seconds, with an average of 34.5 seconds. This is longer than using the full, non-partitioned model.

We address this problem of longer end-to-end request latencies with model partitioning through the use of pipelining. By pipelining, upstream GPUs begins the execution of the next request immediately after it has completed its current request. This increases compute utilization among GPUs. The example in Fig. 4.4 shows that pipelining makes model partitioning worthwhile to consider. In this case, As soon as the first half of the DL model is finished executing in the first GPU, a separate process can transfer the tensor to the next GPU, allowing the first GPU to begin executing other requests in other memory regions in the GPU. This can be performed using RDMA, which instructs the RDMA Network Interface Card (RNIC) of the local and remote machines to perform the transfer at the hardware level without additional overhead and control from the application. The performance impact of pipelining with model partitioning in the example is that cold start durations are reduced without impacting inference durations significantly. This results in an average end-to-end latency for the 8 requests of 24 seconds, which is lower than the original 34 seconds with no model partitioning. This is achieved by the 12 seconds of cold start plus an average of 12 seconds for executing the requests (5 + 7 + 9 + 11 + 13 + 15 + 17 + 19 divided by 8). The examples described in this section motivate investigating model partitioning as a method to reduce DL model cold starts with the trade-off of longer execution times.

We then address **RQ6** by measuring the process initialization and inference times of

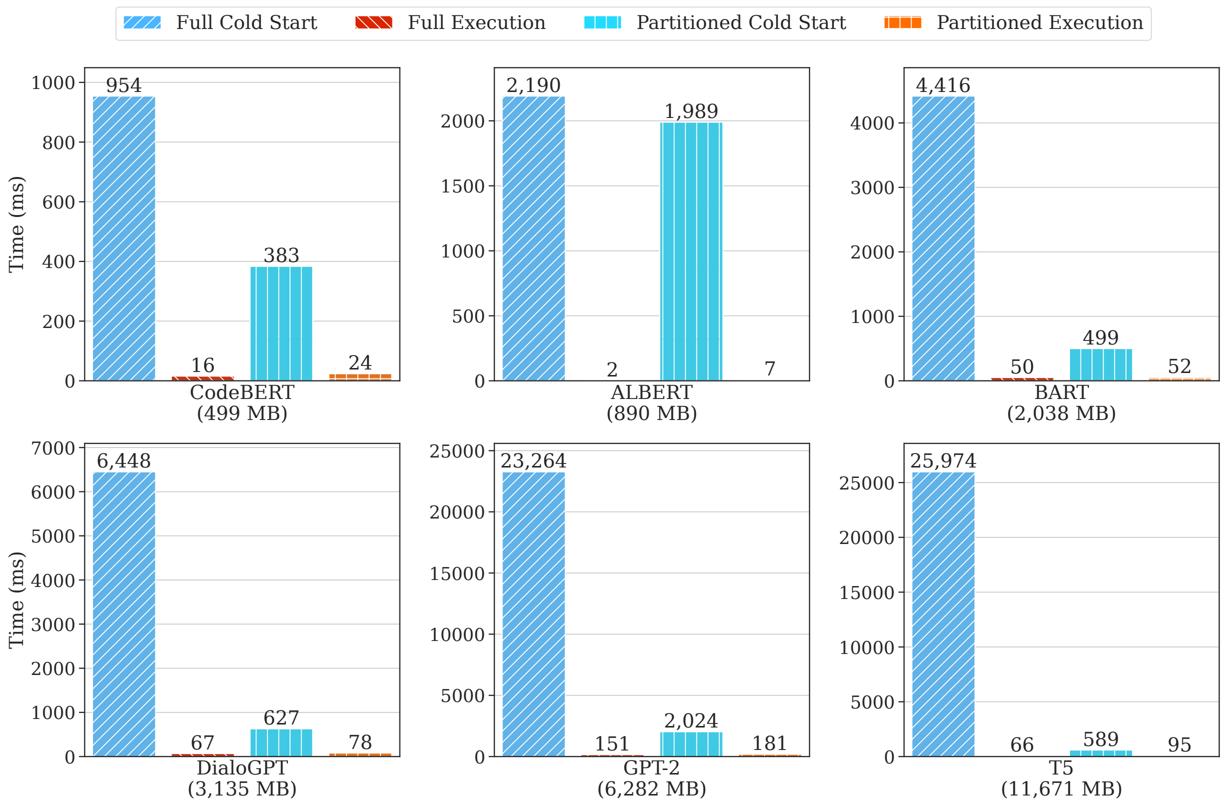


Figure 4.5: Process initialization and inference durations for full and partitioned DL models.

the default full model and optimally partitioned models. Process initialization consists of the steps of loading the DL model on the host machine and sending the loaded objects to the GPU. The optimal partitioning is discussed later in Section 4.2.3. The experiments to measure these are performed using an Nvidia A10 GPU with 24GB of GPU VRAM and 64 GBps PCIe interconnect hosted by Lambda. The DL model is downloaded from an AWS S3 bucket in the same region as the GPU.

Fig. 4.5 shows the process initialization and inference times for the full and partitioned DL model for all models. The process initialization durations for the DL models are optimized with the locality and network techniques in this thesis. For a cold start duration of 25.974 seconds for T5-3B, using at most 55 partitions, which is the number of layers, the expected partitioned cold start becomes 589 milliseconds, which is 44.10× shorter due to the parallel loading processes across GPUs. This also introduces network transfer delays leading to 43.94% longer inference wall times. This analysis is performed by measuring the cold start durations, execution times, and output sizes of individual layers. Then, using the optimized partition plan, we calculate the cold start to be the longest cold start duration of any individual partition. Meanwhile, the inference time is the sum of the execution time of the layers and the network transfer delays between partitions. These analytical results omit networked message overheads and serve as a rough measure of the improvements that can be achieved.

The results show that a mean of a 7.1× speedup in process initialization durations can be achieved across DL models. Meanwhile, the mean of percentage for the network transfer delay to the execution time across DL models is 44.59% or 0.45×. While this percentage may appear large, it is important to note that the reduction in cold start durations outweigh the cost of longer inference times when accommodating excess requests from burst scenarios as described in Fig. 4.4.

Finally, we also identify that the benefit of reducing cold start durations from model partitioning requires that the uplink bandwidth of the source of the DL model parts is not the bottleneck. This then requires that the download path of the DL model part source and target GPUs are independent, otherwise the uplink bandwidth of the source can become the bottleneck.

4.2 Model Partitioning Design

In this section, we look at the actions enabled by the technique of model partitioning and the mathematics behind it. We then design an algorithm to make use of model partitioning and show how it affects the system's architecture. As part of Flashpoint, we present a novel model partitioning algorithm to optimize average end-to-end request latencies. The algorithm incorporates the number of requests in the system as well as the workload profile in scaling decisions. Flashpoint also changes the autoscaling placement decisions by opportunistically splitting and merging models depending on system state.

4.2.1 Trade-offs of Model Partitioning

The notations used in this section is described in Fig. 4.6 and arranged in Table 4.1. In the figure, three requests are shown and served in order with model partitioning and pipelining. The figure assumes that all of the requests are received at the same time at the beginning and only begin execution when it can complete it without delay for visual simplicity. In

Table 4.1 Table of Notations for Model Partitioning

Notation	Description
C	The longest cold start duration of any part
E	The longest execution time of any part
N	The longest network transfer delay of any part
IN	The total network delay incurred by a partition plan
I	The inference time including all execution times and network delays
W	Waiting time for the request excluding the cold start duration
L	Total latency of the request excluding cold start duration (sum of waiting and inference time)
T	Total end-to-end time for the request including cold start duration
c	Number of cuts in a partition plan
p	Number of partitions $(c+1)$ in a partition plan

the example, three partitions are used with three GPUs. Each stage has varying amounts of request execution time on the GPU and network transfer delays. These stages are served in three GPUs. The cold start of the first GPU is set in the example to be the longest among other DL model parts loaded on the other GPUs.

In the notation, C is the cold start duration incurred by the partitioned model, which is the longest cold start duration of any part. E is the execution time of the longest part. In the example, this is part a, among (a, c, e). N is the longest network delay between any partitions, which is part b among (b, d). IN is the total incurred network delay, which is the sum of b and d. I is the inference time, which includes all the execution times and incurred network delay. This is the sum of a to e. W is the waiting time incurred by a request under a partitioned model. This excludes the duration of the cold start to make it proportional to number of requests ahead of it. This proportionality will be discussed later in this section in equation (4.3). L is the latency of the request, which includes the waiting time W and the inference time I. Note that this excludes the cold start duration C, which means it is the end-to-end request latency assuming the DL model is already instantiated. Finally, T is the total time including C, W, and I.

In this subsection, we identify the factors that influence when partitioned models should be used and how many partitions should be made. In the characterization in Section 4.1.2, we identified that a partitioned DL model with pipelining can outperform a full model depending on factors of the reduction in cold start duration, network transfer delays during inferences, and wait times expected by the requests. A pipelined partitioned model can provide lower average end-to-end latencies compared to a full model when considering the case of serving x requests for some large x. This is because the model with pipelining can

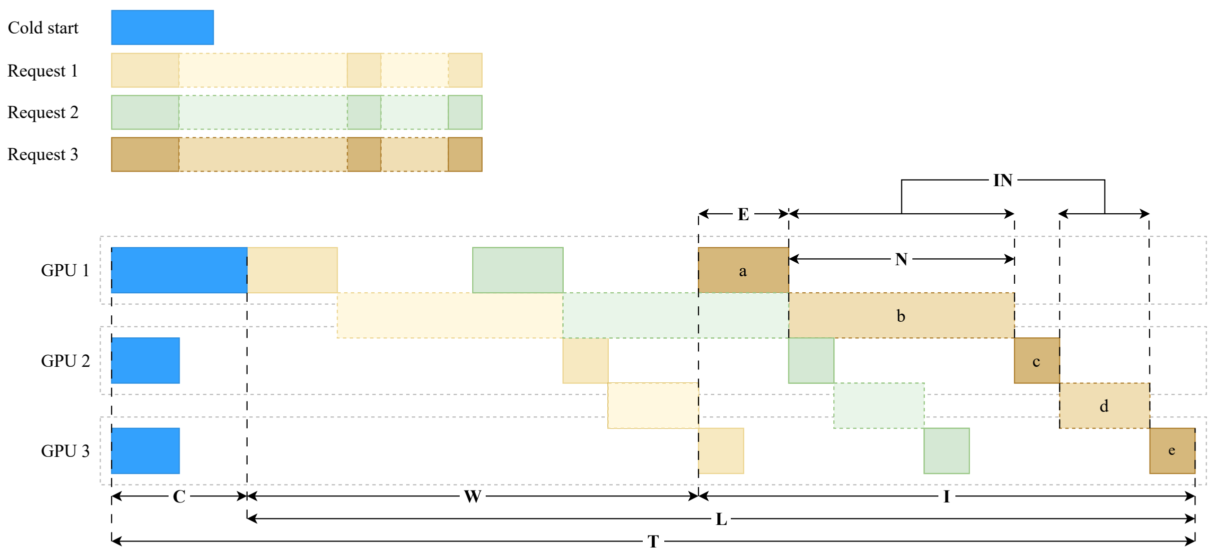


Figure 4.6: Annotated timing sections of a partitioned DL model.

reduce the cold start duration to a larger extent than the amount of increased inference delays it incurs.

To understand when a partitioned model with pipelining can perform better than the full model, we identify when it is advantageous to use model partitioning when a full model and partitioned model have already been instantiated (i.e. omitting the cold start duration). Then, we identify how the cold start affects the average end-to-end latency and formulate a general solution to the problem.

We model the end-to-end latency of the x'th request in the system for a full model and a partitioned one in equation (4.1), where p is the number of partitions. When $p = 1$, a full model is used. The end-to-end request latency without the cold start is then the sum of the waiting time $W(p, x)$ and inference time $I(p)$.

$$L(p, x) = W(p, x) + I(p) \tag{4.1}$$

For a given partition plan with, p, partitions, the inference time is modelled in equation (4.2) as the original inference time of a single partition, $I(1)$, which has no cuts in addition to the incurred network transfer delays $IN(p)$ for the p partitions.

$$I(p) = I(1) + IN(p) \tag{4.2}$$

From the example, $W(p, x)$ can be represented in equation (4.3) as the longest part k from 0 to the maximum number of partitions P, either execution or network transfer of the

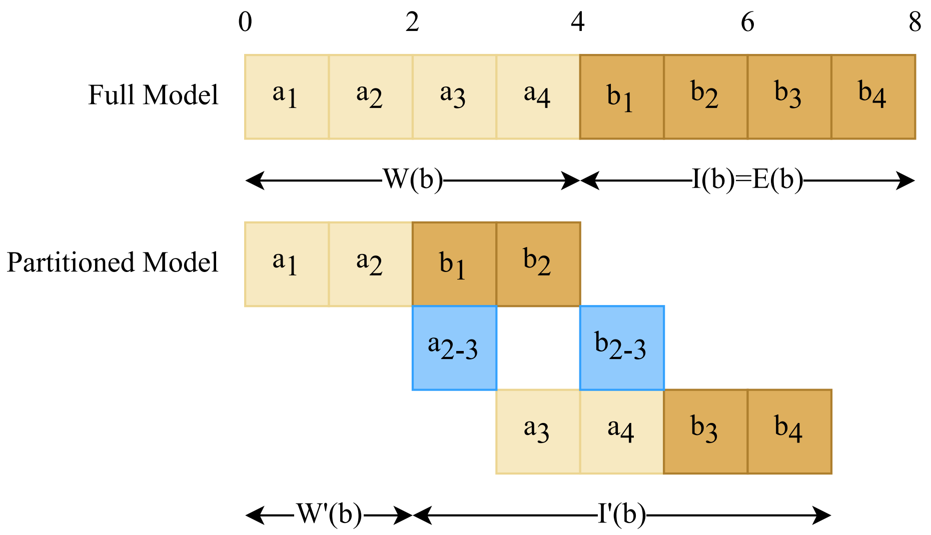


Figure 4.7: Example to determine when a partitioned model should be used compared to a full DL model.

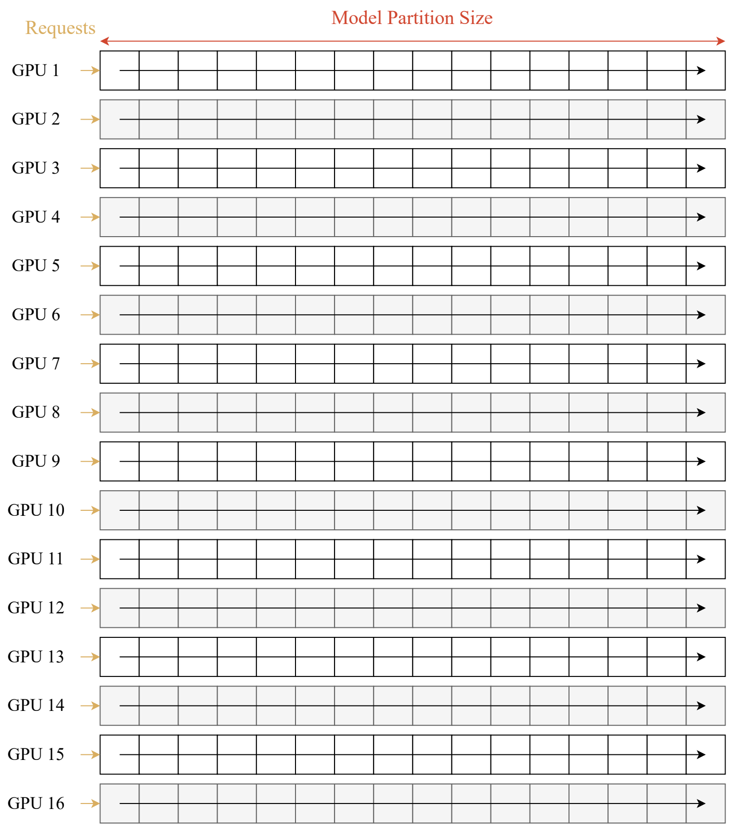
DL model, multiplied by $x-1$ requests pipelined ahead of it. This is because the longest execution time or network transfer delay of any part determines the bottleneck that the $x'th$ request must wait for before it can begin executing without additional waiting delays. This is also shown in Fig. 4.6 where the execution time b is the bottleneck. Thus, the waiting time for the third request, which has annotations, is two times the bottleneck duration b. Formally, the mathematical property of relating the bottleneck and the waiting time can be intuitively achieved by reorganizing the execution and network delays to have the longest part first. To perform the inference, the $x'th$ request must then wait $x-1$ times the duration of the longest execution or network transfer.

$$W(p,x) = \max_{k=0\ldots P}(E(p,k), N(p,k))(x-1) \tag{4.3}$$

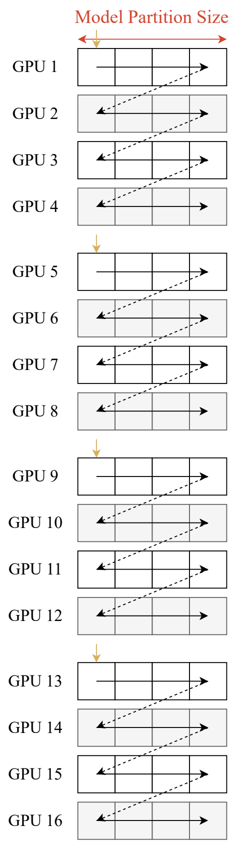
Then, the remaining duration to complete the latency $L(p,x)$ is the inference time of the partitioned model $I(p)$. This leads us to the final formulation of end-to-end latency for a partitioned model without cold starts in equation (4.4).

$$L(p,x) = \max_{k=0\ldots p}(E(p,k), N(p,k))(x-1) + I(1) + IN(p) \tag{4.4}$$

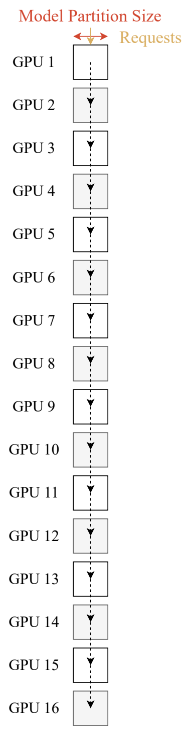
For all values of x, increasing p will always yield longer $L(p,x)$ and lower values for the cold start duration $C(p)$. However, this is also not yet normalized by the number of GPUs that different partitioning schemes use.



(a) Example of 16 GPUs hosting full DL models.



(b) Example 16 GPUs hosting $\frac{1}{4}$th DL models.



(c) Example of 16 GPUs hosting $\frac{1}{16}$th DL models.

Figure 4.8: Example for network bottleneck in concurrent cold starts.

In the next example, we compare the performance of different levels partitioning on equal amounts of resources. In particular, we show how different levels of partitioning can serve the same number of requests concurrently. This means that assuming the parts of the partitioned model are loaded, the throughput of the system is mainly affected by the increased inference time from extra network transfers rather than the number of requests it can serve concurrently. This throughput affects the waiting time of requests when there is excess demand, as leading requests are served first before the new request is served.

Fig. 4.8 shows a comparison of concurrency levels for different amounts of partitions for a DL model. Since a DL model using p partitions requires at least p GPUs, we set the examples to use P (the maximum number of partitions) GPUs for fair comparisons in number of resources used with a value of $P = 16$. The figure shows that for P GPUs, the full DL model can execute with a concurrency of P. In Fig. 4.8a, each GPU hosts a single unpartitioned model. Since there are P GPUs, the concurrency of the system is P. Meanwhile, in Fig. 4.8b, when splitting the DL model into $p = 4$ equal partitions, 4 GPUs can host each partition. Since each of these GPUs host $\frac{1}{4}$th of the DL model, the execution time on each GPU is approximately $\frac{1}{4}$ of the unpartitioned execution time. However, while there are then $\frac{P}{4}$ main replicas, the concurrency remains at P due to pipelining requests. In other words, at maximum, there will be P requests being served by the P GPUs. Finally, for generalization, in Fig. 4.8c, when $p = P$, each GPU hosts one partition. The concurrency becomes 1. This example shows that the concurrency level of the DL model with a given number of GPUs is unaffected by the number of partitions used.

The example in Section 4.1.2 also showed that with various number of partitions p, the first floor($\frac{P}{p}$) requests can be served without waiting delays. Meanwhile, the next P requests need to wait until at least the first partition is completed, and so on. Based on the example in Fig. 4.7, the duration that a request has to wait is defined by the longest part as in equation (4.3). This leads us to model the waiting delay in the partitioned model with equal GPU resources as in equation (4.5).

$$W(p, x, P) = \max_{k=0 \ldots P}(E(p, k), N(p, k)) \cdot (\text{ceil}(\frac{x}{\text{floor}(\frac{P}{p})}) - 1) \tag{4.5}$$

Then, assuming equal partitioning, the cold start delay incurred by each request is independent of x as in equation (4.6). The equation assumes that the partitions are approximately equal in size, leading to the longest and overall cold start duration being $\frac{C(1)}{p}$.

$$C(p) = \frac{C(1)}{p} \tag{4.6}$$

The end-to-end latency for serving request x in a queue is then shown in equation (4.7), which includes the cold start duration.

$$T(p, x, P) = C(p) + W(p, x, P) + I(p) \tag{4.7}$$

The average end-to-end latency for serving all x requests then becomes that in equation (4.8).

$$\bar{T}(p,x,P) = \frac{\sum_{y=1}^{x} T(p,x,P)}{x} \tag{4.8}$$

Then, to identify the theoretically optimal solution, we develop equation (4.9), which uses the probability $Pr(\lambda = x)$ for serving x requests with the model and the cold start duration $C(p)$ for p partitions.

$$\mathbb{E}(\bar{T})_{best} = \min_{p=1\ldots P} \sum_{x=0}^{\infty} (\bar{T}(p,x,P) \cdot Pr(\lambda = x)) \tag{4.9}$$

4.2.2 Precalculation of Partition Plans

Solving equation (4.9) during the autoscaler event loop for changing probability distributions is impractical since it requires multiple dynamic programming solutions for an infinitely large number of x's. Thus, we develop a search algorithm that precalculates the best number of partitions for values of x offline. We identify that for adjacent values of x requests to be served by the partitioned model, the optimal partition scheme is likely to be the same. Thus, there are ranges of x values from n_1 to n_2 where the optimal partition scheme at those x values are the optimal across all possible number of partitions from 1 to a maximum number of partitions, P. For an optimal partition scheme with p partitions, when the maximum number of partitions to be used when scaling up is P' where $P' < p$, then the optimal partition scheme with P' partitions is used. The range of x with the same optimal partition scheme can then be precalculated for a broad range of x values and referenced during the workload. To do this, we set the maximum number of partitions to the total number of layers in the DL model. We then identify that the x value is the only search axis. We first find the best number of partitions (and their cuts) at x = 1 with dynamic programming. Then, we use an exponential search algorithm to identify the first x value k that changes the optimal number of cuts. Searching for the bounds of different optimal values of p is then continued up to a maximum number of $X = 3000$ requests. The value of X is selected as the maximum number of requests in the queue for the baseline frameworks for the workloads. Concurrently, the best partition plans for number of partitions from 1 to P are also saved for each value of x. These partition plans are used when the autoscaling policy decides to scale up by fewer number of replicas than the calculated optimal number of partitions from the partition plan. During the exponential search, we use a memo for calculating $\bar{T}(p,x,P)$ to reuse the sum calculations.

The locality property of the DL model parts in the local host machine as compared to a remote host machine impacts the optimal placement of these DL model parts. Given a limited number of GPUs per host, sequential DL model parts can be placed in the same host to reduce network transfer costs. However, this increases the cold start duration of future scale-up actions due to occupying local GPUs with different DL model parts. By using cold start and network transfer cost values corresponding to non-local cases, we solve for the optimal partitioning plan for the worst case. The actual performance of the system is then improved by locality and guarantees that the partitioning plan always outperforms only using a complete model.

4.2.3 Finding the best cuts for a DL model

The layers of a given DL model can be partitioned in many ways. Whenever a part is cut into separate partitions, the cold start duration becomes the maximum of the cold start durations of either part. This is because each part can be loaded in parallel. Meanwhile, each cut also introduces networking delays of transferring the output tensor of a part to the next. This affects inference times. The impact of these networking delays vary in magnitude depending on how many requests the partitioned model will serve.

To solve for the best partition plan for a given number of requests, we develop the dynamic programming formulation described in equation (4.10). The algorithm is an extension of the dynamic programming solution for the log cutting problem. In comparison to the traditional problem, cuts at different locations in the DL model can result in different costs. As such, the solution must traverse through all possible cut placements for up to the maximum number of cuts, which is the number of layers minus 1. The algorithm takes the cold start durations of each layer and their corresponding execution times and input tensor sizes. The input tensor size is determined by the output tensor size of the previous layer. The algorithm uses memoization to cache previously solved results.

For each position, (i, j, c) in the solution matrix M, where i and j are the starting and ending layers in the DL model, and c is the number of cuts available, the minimum of two types of solutions is taken. The first is when no cuts are used, then the optimal solution is the same as that for $(i, j, c-1)$. Then the other case performs a cut at position k for all values of k from i to j and gets the best value when allowing c_l more cuts on the left side and the remaining $c-1-c_l$ cuts on the right for c_l from 0 to $c-1$. The solution matrix is initialized for all c with the values when no cuts are used.

Additional matrices DC, DE, DN are used to store the duration of the maximum cold start time, execution time, and networking delay for any of the partitions respectively for combinations of i, j, and c. Meanwhile, a matrix DIN is used to store the incurred networking delay for the combinations of (i, j, c). These matrices are grouped together as an object D. Meanwhile, $\overline{T_{dp}}(D, x, P, i, j, c)$ is the average end-to-end latency of x requests given the best partitioning scheme for layers i to j for c cuts, and $\overline{T_{dp2}}(D, x, P, i_l, j_l, c_l, i_r, j_r, c_r)$ is same but for the best partitioning scheme for left side i_l, j_l, c_l and right side i_r, j_r, c_r.

$$M_{i,j,c} = \min \begin{cases} T(D, x, P, i, j, c-1) & \text{for no cut} \\ \min\limits_{\substack{k=i\ldots j \\ c_l=0\ldots c-1}} \begin{cases} \overline{T_{dp2}}(\\ \quad D, x, P, \\ \quad i, k, c_l, \\ \quad k+1, j, c-1-c_l \\ ) \end{cases} & \text{for non-final cut} \end{cases} \tag{4.10}$$

The function in $\overline{T_{dp2}}$, takes the longest best cold start duration $DC_{i,j,c}$ for (i, j, c) on the left side (i, k, c_l) and the right side $(k+1, j, c-1-c_l)$. This is done for both DE and DN. Then To identify the longest part, the max of DE, DN, and $d_n[k]$ is used to calculate W. Finally, $DIN_{i,j,c}$ is the $DIN_{i,j,c}$ for the left and right side, with addition of $d_n[k]$.

4.2.4 Model Partitioning System Architecture

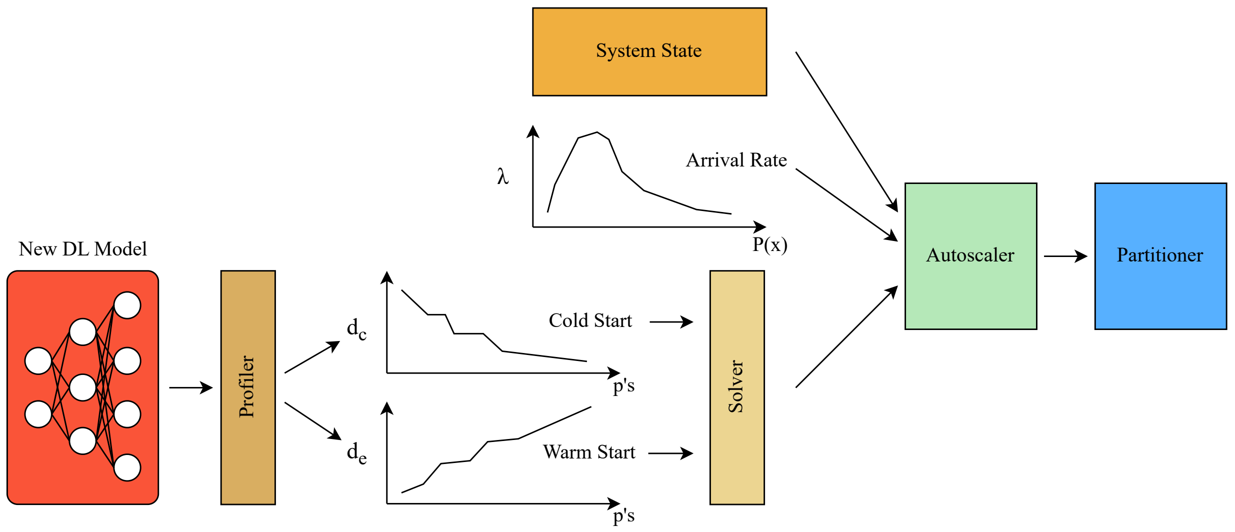


Figure 4.9: System architecture using model partitioning.

Fig. 4.9 shows the components involved in model partitioning and their interactions. When a new DL model is received by the system, it is first partitioned in to layers, and each layer's cold start duration, inference time, and output sizes are analyzed. Then using dynamic programming, the optimal solution for reducing cold starts is identified for different numbers of partitions at different values of x requests expected in the system. This results in precalculated partition schemes for the DL model. Afterwards, during the operation of the autoscaler, multiple factors are used to make scaling and placement decisions. These include the state of the system, which includes how many full and partitioned replicas it has for the DL model, the arrival rate of requests for the DL model, and the profiled information. The decisions from the control plane are then sent to the partitioner for execution.

4.2.5 Model Completion

When the system scales up with a partitioned DL model, the effective service rate of the new replicas are lower than that of the same number of machines with full DL models. This is because the network transfer delays from transferring intermediate data causes each inference request to be longer and reduces the collective throughput of the machines. Because of this reduced throughput, the system will eventually need instantiate more replicas or suffer longer waiting times.

In order to limit this downside of model partitioning, we develop a mechanism called model completion. Model completion instructs the machines of a partitioned DL model to perform the cold start for the other layers continuously after it has completed setting up its initial part. After the time it takes to download the additional layers, both a system using model partitioning and the baseline system are equivalent and have the same number of machines, each with full DL models. Through this mechanism, model partitioning enables the system to serve requests earlier than the baseline, without incurring penalties in the long term.

4.2.6 Model Partitioning with the Autoscaler

To maximize the benefit of model partitioning, we design the autoscaler to control a mix of full and partitioned models. For a set R of replicas for a DL model, R can consist of full and partitioned models as shown in equation (4.11).

$$R = \sum_{p=1}^{P} (R_p) \tag{4.11}$$

The autoscaler's event loop generates a desired number of replicas as determined by the autoscaling policy. Given the current number of replicas , the desired number of replicas can fall into one of three categories in equation (4.12):

$$CAT(R_t, \lambda_t) = \begin{cases} (1) \text{ scale up} & \text{if } \lambda_t + \rho_H > \mu_t \\ (2) \text{ keep the same} & \text{if } \lambda_t - \rho_L < \mu_t < \lambda_t + \rho_H \\ (3) \text{ scale down} & \text{if } \lambda_t - \rho_L > \mu_t \end{cases} \tag{4.12}$$

In equation (4.12), ρ_L and ρ_H are configurable low and high thresholds for number of replicas, while μ_t is defined in equation (4.13) as the sum of service rates of different partitioned models.

$$\mu_t = \sum_{p=1}^{P} \mu_{R_p,t} \tag{4.13}$$

In the case of scaling up, we determine the number of requests expected to be served using the definition in equation (4.14). This equation includes both the number of requests in the system and excess load. Since we bound the duration of model partitioning cold starts to the cold start duration of the full model, we use $C(1)$ to measure the time that the excess load $(\lambda_t - \mu_t)$ adds requests to be served.

$$\mathbb{E}(x) = q_l + (\lambda_t - \mu_t) \cdot C(1) \tag{4.14}$$

Then, from this value of $\mathbb{E}(x)$, we select the best partition plan from the precalculated solutions with at most P partitions where P is the number of replicas to scale up by, as determined by the autoscaler.

Meanwhile, when keeping the number of replicas the same, no changes are made. Finally, when scaling down, the autoscaler prioritizes evicting partitioned models due to its lower service rate from extended execution times. Due to partitioned models using more than one GPU, when scaling down, the replicas are sorted by decreasing number of GPUs used. Iterating through this sorted list, the autoscaler then removes models with at most the n GPUs to scale down by until at most n GPUs have been selected for eviction.

Chapter 5

Locality-Aware Autoscaling

5.1 Characterization of Long Cold Start Downloads

In this section, we are motivated by the problem of long download durations for DL models. To address this issue, we measure the amount of excess resources in the system. We answer the following research questions:

RQ7. What is the main cause of long download durations?

RQ8. How much extra resources are present in a saturated system?

RQ9. How can extra resources be used to reduce download durations?

5.1.1 Main Cause of Long Download Durations

In response to **RQ7**, we identify that long download durations, especially with large DL models, are due to downloading all of the DL model from the same source, limited primarily by the network bandwidth in the route from source to destination. This issue is also only more relevant as the size of DL models continue to grow.

5.1.2 Extra Resources while Saturating GPU Compute

To address **RQ8**, we acquire measurements of resource utilization rates for different resources of compute and memory for the GPU and CPU, and network utilization when the

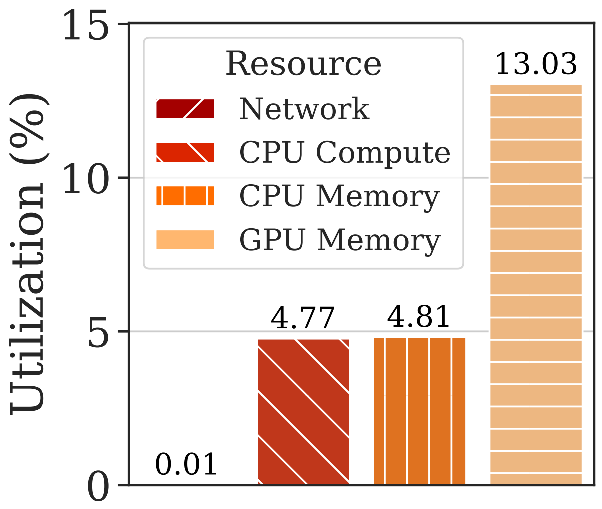


Figure 5.1: Utilization (%) of system resources with saturated GPU compute.

system is processing requests at its maximum throughput. Fig. 5.1 shows the percent of network bandwidth, CPU compute, CPU memory, and GPU memory that is consumed while saturating GPU compute at over 95% utilization and completing requests at maximum throughput across DL models. GPU compute utilization is omitted since it is saturated. The test is performed by initializing a remote queue with 1,000 images from the ImageNet dataset [43], then having torch.multiprocessing GPU processes consume images from the queue, perform the inference, and write the result to a shared file. The test is performed with 1, 2, and 4 processes and is shown to saturate GPU compute utilization with just 1 process. The results shown use means across DL models.

The characterization shows that a single GPU process is enough to saturate GPU compute while leaving extra memory and network resources. This motivates using new physical machines to scale out throughput rather than creating new processes in the same machine since GPU compute is the bottleneck for throughput. In addition, the excess resources provide the opportunity to source DL models from other host machines along with the necessary network bandwidth for the transfers. Since the input data for DL inferences is typically orders of magnitude smaller than the DL model itself, a compute-saturated GPU only requires less than 0.01% of network bandwidth to transfer the inputs and outputs to and from the GPU since they are mostly text data. Similarly, less than 4.77% and 4.81% of CPU compute and memory respectively are used. Finally, only 13.03% of GPU memory is used. Typical GPU-enabled EC2 instances provide excess memory and network resources under GPU compute saturation and can be used as remote memory storage [10].

5.1.3 Opportunity Provided by Excess Resources

Finally, to answer **RQ9**, our results from Fig. 5.1 show that the amount of excess memory is enough to store replicas of inactive DL models and that sufficient network bandwidth can be used to perform transfers of the DL model data between host machines without interfering with request completion throughput.

5.2 Locality-Aware Autoscaling Design

In this section, we design the use of excess resources and locality-awareness in Flashpoint's autoscaler to solve long download durations for large DL models. It involves using remote memory pooling to create a compute-colocated distributed registry which downloads the DL model from nearby host machines. In addition, a Pareto-optimal replica placement algorithm is designed to maximize the benefit of locality.

5.2.1 Remote Memory Pooling

By identifying opportunities in excess resources in the system, we further reduce the probability of using the worst case slowest path of downloading from cloud storage at no added cost. One-shot profiling as described in Section 5.1.2 is used to determine the amount of resources used by a DL model when it saturates GPU compute serving multiple requests.

In Flashpoint, the binary data of the DL model must be downloaded into a host machine's memory, loaded into process memory, and sent to the GPU, consuming GPU memory. These memory requirements are reserved at the start of a cold start to ensure it can be completed. In case of memory pressure in host machines, model data is evicted first before process memory. As memory and network resources are underutilized in modern cloud offerings, remote memory pooling and hierarchical sourcing do not incur additional cost.

This mechanism utilizes the excess memory and network bandwidth across host machines to source the DL model from existing replicas in other host machines when instantiating a new GPU node. This enables the storage of different DL models across host

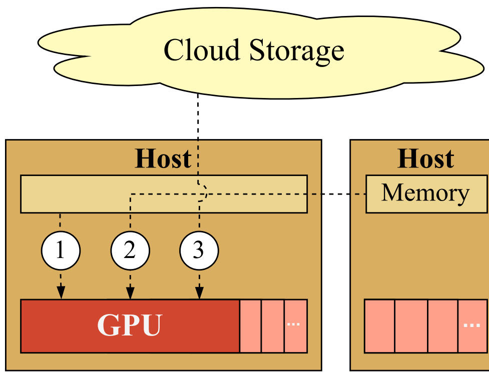


Figure 5.2: Hierarchical sourcing and remote memory pooling.

machines in the system and effectively bounds the cold start duration to intra-data center network speeds in contrast to cloud storage downloads.

While an option is to pre-load all DL models in the system, this work makes no assumption on the number of DL models and number of GPUs in the system. As such, the work is generalizable and aims to instead minimize the footprint of individual DL models.

5.2.2 Compute-colocated Distributed Registry

We identify properties of DL inference serving, in comparison to DL training, that present an opportunity to optimize the cold start process. In contrast to deep learning training, inferences do not have to update model state and thus model data are mostly static. These models can then be replicated across host machines in the system. Furthermore, current data centers typically have multiple physical GPUs attached to each host machine [10]. Finally, current systems handle cold starts by downloading the DL model from cloud storage or a remote Network File System (NFS) server [58, 21].

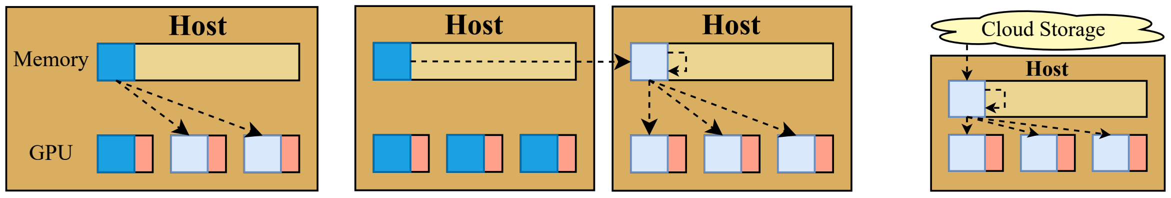
Due to these properties, we design the mechanism of hierarchical sourcing as shown in Fig. 5.2 to source the DL model data for a target GPU node from the $\textcircled{1}$ local host machine first, over $\textcircled{2}$ remote sources of other host machines, and $\textcircled{3}$ cloud storage. This reduces the download step duration. Using in-memory copies in the local host machine, hierarchical sourcing eliminates most of the cold start durations that involve downloading the model to the host machine and loading the data into memory. This leaves the remaining duration to be in sending the loaded tensor data to the GPU.

5.2.3 Pareto-optimal Greedy Placement Algorithm

The main challenge of serverless scheduling is balancing resource efficiency and meeting deadlines. Serverless systems have the option of packing incoming requests on active nodes which can cause queuing delays, or initializing new ones, which lead to higher costs. This leads to a large solution space for GPU machine combinations with varying levels of resource efficiencies, end-to-end latencies, and cold start penalties. To navigate this, we use locality information and simplify the set of candidate GPUs for instantiation. We divide the scheduling problem into two stages: scheduling requests among active instances, and determining when and where to create new instances. The overarching goal of the system is to amortize cold starts across multiple warm start requests while meeting the most deadlines.

We design the scheduler to evenly distribute requests among active nodes using Sparrow-style [70] global queue to assume optimal scheduling for both baselines and proposed system. This optimizes scheduling to reduce waiting delays by having workers pull messages from the queue when they have resources available. In comparison, a round-robin scheduler with per-worker queues lead to older workers having numerous requests while newer nodes have close to an empty queue.

While request batching has been used to amortize cold start costs by grouping requests to a replica, the use of batching in selecting cold start nodes has not been studied extensively. Batching of candidate nodes during burst cold starts naturally occurs because modern autoscalers accumulate waiting requests in the system before scaling up. Thus, the number of nodes to scale up by is typically greater than one. This presents an opportunity to optimally place new instances based on this simultaneous cold start compared to stateless sequential cold starts.



(a) Local host to GPU cold start.

(b) Remote host to GPU cold start.

(c) Cloud storage to GPU cold start.

Figure 5.3: Data path diagrams for locality-aware cold starts with hierarchical sourcing.

```
# autoscaler.py
def get_receivers(self, model_id: UUID, scale: int):
    """
    model_id: Identifier for the DL model to scale up.
    scale: Nat of how many replicas to scale up by.

    Returns:
    - receiver_gpus: List of GPU nodes for cold start
    - receiver_hosts_set: Set of hosts for cold start GPUs
    """
    receiver_gpus = []
    do_break = False
    for host in list_of_hosts:
        for gpu in host.children:
            if can_allocate(gpu, model_id):
                reserve(gpu, fd)
                receiver_gpus.append(gpu)
                leftover_scale -= 1
                if leftover_scale == 0:
                    do_break = True
                    break

                    break
        if do_break:
            break

    receiver_hosts = set(map(lambda gpu: gpu.parent, receiver_gpus))
    return receiver_gpus, receiver_hosts
```

Listing 5.1: Pseudocode for locality-aware Pareto-optimal greedy cold start candidate selection algorithm

In this section, we present a locality-based optimal greedy algorithm for selecting which GPUs to instantiate for burst cold start requests.

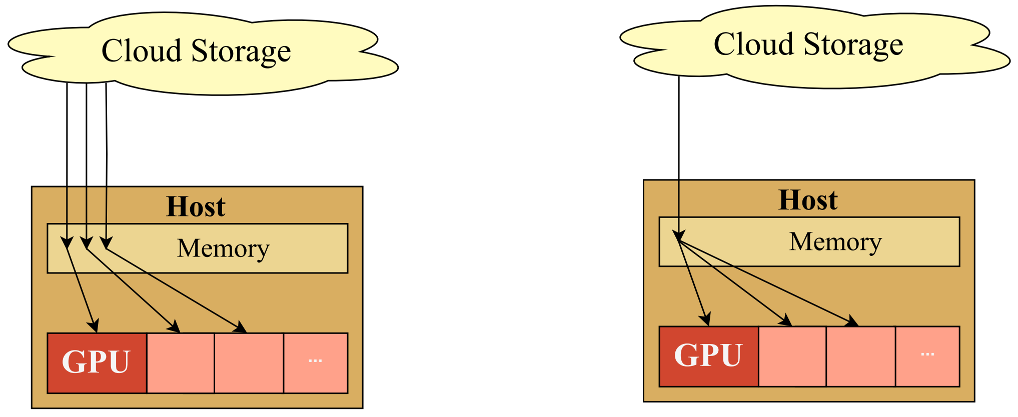
Fig. 5.3 depicts three locality decisions in Flashpoint. In Fig. 5.3a, a copy of the DL model is present in the host memory in blue, which also indicates that at least one other GPU already has the model. In this hierarchical level of cold start, the model takes one step to be sent to a new GPU in light blue. Meanwhile, Fig. 5.3b shows a remote host (on the left) transferring the DL model from its memory to a local host (on the right), loading the model, and then sending the DL model to the GPUs in the local host. Finally, Fig. 5.3c indicates the default mechanism of download the model from cloud storage, loading the model on the host machine, then sending the model to the GPU.

During batched selection of cold start nodes, two extremes are present. First, selecting only GPUs that are in the same host attempts to capture the shortest hierarchical cold start; however, this is suboptimal due to the batched nature of cold starts. In this case, the cold start takes as long as a host-to-host download, loading the model, and sending to the GPU when the target host does not yet have the DL model. This is because of task dependency between downloading the DL model and the remaining downstream actions. Second, selecting only one GPU from each host, since selecting additional GPUs in the same hosts do not present immediate benefits, is also suboptimal due to not capturing the benefit of the shortest hierarchical cold start. To optimize this placement, we present a two-stage Pareto-optimal greedy algorithm. We first select GPUs in host machines that already have the model. This requires that there are host machines with the model. Meanwhile, for the remaining worker nodes, we select single GPUs from other host machines. This second stage primes the system for future cold start bursts and obtains a Pareto-optimal allocation of cold start nodes. The proof of Pareto-optimality for this algorithm is as follows. First, selecting fewer GPUs that all that are primed for the fastest level of cold start reduces the benefit captured by the algorithm. Second, packing the remaining GPUs in the same host machine also reduces the benefit of shorter cold starts that can be captured in the future. This leads to the optimal solution being to fully utilize primed GPUs, while distributing the remaining GPUs across different host machines.

The algorithm is shown in Listing 5.1. It starts by iterating through each GPU of each host machine in the cluster. Then, it attempts to reserve the required resources in the GPU. The GPUs selected in the algorithm must satisfy the following conditions. It must have the memory and compute capacity available for running the DL model process based on profiled averages. Meanwhile, the host machine that the GPU is in must either have the DL model process initialized or have enough memory and compute capacity to first download the DL model into memory as well as load the DL model process. Then, it reserves the resources on the GPU and respective host machine. The algorithm first select

GPUs colocated within the host machines that already have the DL model. Finally, for the remaining replicas to be instantiated, it distributes the cold start GPU candidates across host machines by skipping the remaining GPUs in the host machine if the current selected GPU is the first to load the model in the host.

5.2.4 Burst Download Request Sharing



(a) Example of independent downloads. (b) Example of download sharing.

Figure 5.4: Example figures for download sharing.

Multiple cold starts during burst requests can cause a system to download redundant copies of the same DL model destined for different GPUs in the same host machine. Fig. 5.4a shows an example of independent downloads for different GPUs occurring using the same host machine. To address this redundancy, we implement a download manager agent that checks downloads in progress and performs transfer chaining system to target GPUs from the host machine. This is shown in Fig. 5.4b, where a single download to the host machine is used, and that data is sent to multiple GPUs. This prevents redundant transfer requests for a DL model and allows distribution of the DL model to different GPU nodes as soon as the download on the host machine is completed.

Chapter 6

Network-Aware Autoscaling

6.1 Characterization of Network Congestion

In this section, we measure the frequency and magnitude of network congestion in the Twitter [96] workload. Network congestion occurs when the system's autoscaler scales up the number of provisioned replicas and more than one of these cold starts use the same download source. The network congestion from using the same source extends cold start durations and reduces the elasticity of the system.

We then formulate the following research questions:

RQ10. How does the concurrency of cold starts impact the duration of cold starts?

RQ11. How frequent are simultaneous cold starts in a serverless system?

6.1.1 Effect of Concurrent Cold Starts to Cold Start Duration

The network in a data center is a bottleneck during simultaneous cold starts which occurs in production systems [99, 113, 51] and validated in our physical experiments. Modern autoscalers such as in AWS SageMaker and Kubernetes spin up new instances asynchronously from scheduling requests [29, 1, 9]. These autoscalers use metrics such as a target average invocations per instance, queue latency, compute utilization, etc. to decide the desired number of replicas for a DL model at a given instant. Because of this, queued requests build up in a system during a burst in demand, causing all requests the requests to have long queue wait times before new instances are simultaneously instantiated. In comparison to asynchronous systems, synchronous systems such as [28] require requests to block until cold start instances are initialized, leading to significantly longer wait times. To compound the issue, due to the simultaneous nature of cold starts, multiple download requests going through the same links in the network cause network congestion, which extends each of the cold starts. In particular, a one-to-many host-to-host transfer causes congestion in the link from the host machine to the top-of-rack switch in the data center.

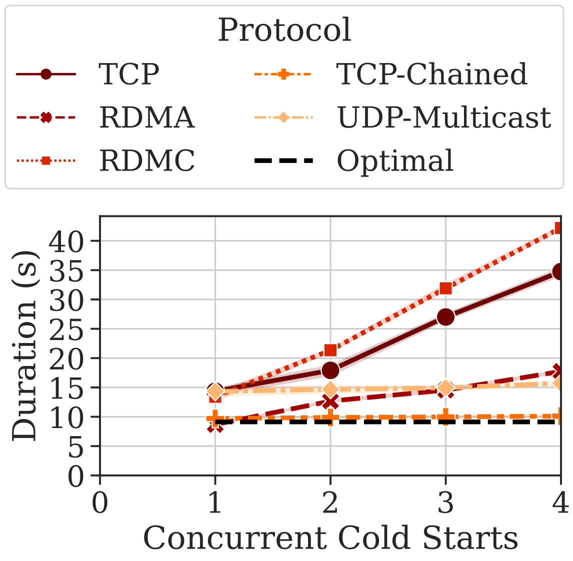


Figure 6.1: Duration of host-to-host cold starts compared to number of concurrent cold starts for T5-3B.

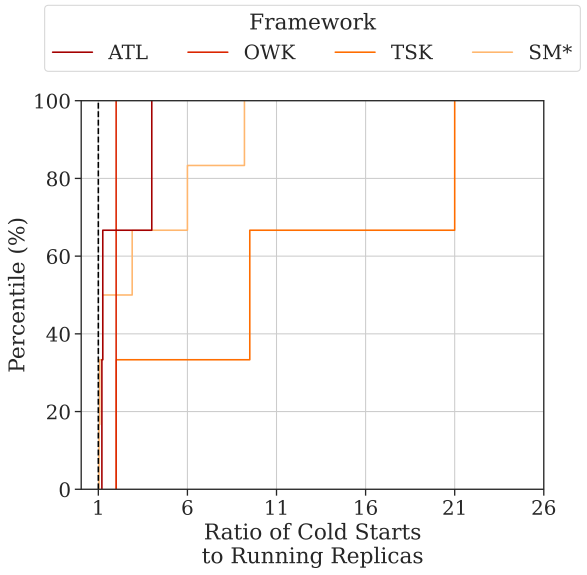


Figure 6.2: Cumulative distribution function (CDF) of the ratio of new replicas requested (cold starts) to the number of running replicas.

To answer **RQ10**, we measure the duration of cold starts compared to the number of concurrent cold starts for the 11 GB T5-3B model in Fig. 6.1. Results for the other models are similar in shape. This experiment is performed using 5 servers, each having 12 Intel® Xeon® E5-2620 v2 CPUs, 64 GiB of RAM, and 10 Gbit/s network connections.

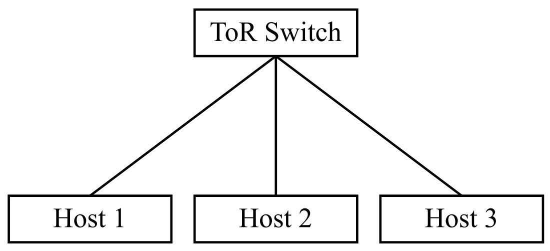
The figure shows a linear increase in cold start duration with number of concurrent cold starts for baseline transfer protocols using TCP, memcached, and Remote Direct Memory Access (RDMA). This performance degradation is attributed to network congestion in the data center where the download duration is extended. Results for Encrypted UDP based FTP with multicast (UFTP) [19] and A RDMA Multicast (RDMC) [15] using TCP and

RDMA are omitted due to the results being greater than 5× the optimal download time for the model.

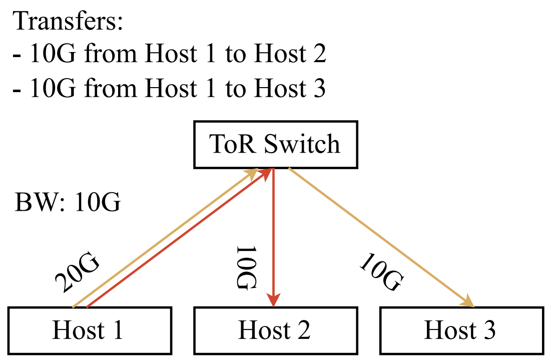
6.1.2 Frequency of Concurrent Cold Starts

Meanwhile, to answer **RQ11**, we measure the the ratio of requested replicas (cold starts) to running replicas for scale up actions for different autoscaling policies for the Twitter workload. Fig. 6.2 shows a cumulative distribution function (CDF) of these ratios. The ratio represents the best case scenario of evenly mapping a set of receiver host machines to a set of source host machines. For example, if the ratio is 0.5 and the number of running replicas is 10, half of the running replicas can act as sources for the 5 new replicas being instantiated. The results show that all scale up actions require a ratio of new replicas request to current running replicas greater than 1. This indicates that the case of source host machines sending to multiple receiver host machines is fairly common. In combination with Fig. 6.1, this characterization motivates the need to improve download management to mitigate prolonged cold starts.

6.2 Network-Aware Autoscaling Design



(a) Example network structure of three hosts connected to a Top-of-Rack (ToR) switch.



(b) Example of two 10 Gigabit transfers occurring simultaneously and congesting the network with 10 Gigabit/s (G) links.

Figure 6.3: Example for network bottleneck in concurrent cold starts.

Current serverless systems execute the scale-out of DL model replicas individually, with each new instantiation process unaware of the other. In the workloads tested, we observe that bursts in requests lead a burst of cold start instantiations which all occur at the same time. This leads to network congestion in the host machines that act as data sources for the DL model.

Fig. 6.3 shows an example of the network bottleneck during concurrent cold starts. In the example, host-host network transfers to distribute the model. Fig. 6.3a shows a simple network structure of 3 host machines each connected to the same Top-of-Rack (ToR) switch, which is a common structure in data centers. In the network structure, 10 Gigabit/s (G) duplex links are used. Fig. 6.3b shows a scale out action of ratio 2. In this case, when evenly distributed, every host machine that contains the DL model can send the model out to two other hosts. In the example, Host 1, sends a DL model of 10 Gbits to Hosts 2 and 3. Because the network links have a capacity of 10 Gbits/s, individual transfers would normally take 1 second in isolation. However, since the transfers occur simultaneously and are unaware of each other, the transfers congest the uplink bandwidth of Host 1, and take twice as long. This leads to degraded download times and longer cold start durations.

To address this issue, we employ the use of multicasting, which has not been applied to reduce cold start durations in serverless deployments. Multicasting reduces network congestion by enabling servers to send a single packet to be received at multiple clients. In one protocol, network hardware primitives could be used to replicate packets at network switches, rather than the sender machine, effectively reducing the amount of redundant information sent from the sender to the switch.

The use of multicasting comes with several issues. One is that multicasting uses the User Datagram Protocol (UDP), which does not guarantee that packets are received by the other hosts such as in a TCP protocol. In addition, current reliable multicasting protocols are shown to have high synchronization overheads, leading to longer download durations compared to multiple TCP downloads.

The method employed in Flashpoint to reduce network congestion in a multi-node transfer is to use chained TCP flows. This method uses the full-duplex bandwidth of modern data centers to execute the transmission of data from a receiver node to another node concurrently as it receives data. It works by receiving the packet in a stream at a node's buffer and immediately copying the packet and forwarding it to the next node in the chain. While this method has its drawbacks including longer time-to-first-byte (TTFB) latencies in downstream nodes in longer chains, it is sufficient to achieve close-to-optimal scaling with greater cold start concurrency and is sufficient for this work. An example of

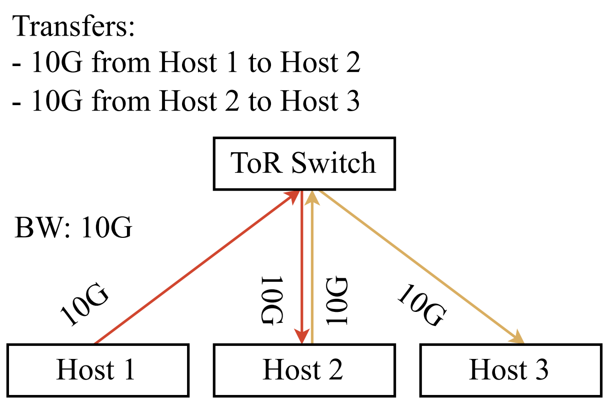


Figure 6.4: Example of TCP chaining with no network congestion.

the TCP chain protocol is shown in Fig. 6.4 where two 10 Gigabit transfers from Host 1 to Host 2, and from Host 2 to Host 3 then occur concurrently, allowing both transfers to complete in approximately 1 second.

In comparison, a Remote Direct Memory Access (RDMA) protocol bypasses the operating system kernel and other protocols by performing transfers through a data path that does not involve using CPU compute cycles. This path starts from the application's buffer, then the local RDMA-enabled Network Interface Card (RNIC), followed by the remote RNIC, and finally to the remote application's buffer. This avoids redundant copies being made on the critical path and is a promising direction for multicasting. In non-RDMA protocols, copies from the application's buffer may be copied to socket and transport protocol driver buffers before they arrive at the local RNIC.

However, current RDMA-based reliable multicasting protocols such as [15] and [19] are unable to outperform TCP chaining due to the lack of necessary network primitives that allow group packet multiplication at switches or packet copying in streams without application-level control. While [52] works on using P4 software-defined networking solutions to enabling reliable multicasting, the controller functions are not yet available for this work.

6.2.1 Reliable Multicasting with TCP Chaining

To perform TCP chaining, Flashpoint employs a master controller which resides in the autoscaler to manage connection parameters and updates as shown in Fig. 6.5. During cluster creation, a long-lived TCP chain manager application is launched in each host

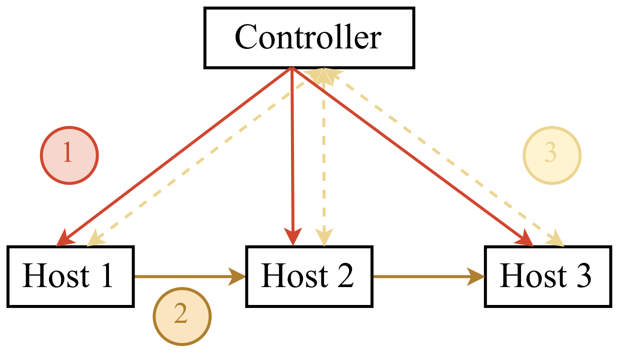


Figure 6.5: Controller and host machine interactions for TCP chaining.

machine. Then, when a scale up decision is made, as shown in Fig. 6.5, ① the first step of the TCP chaining protocol is to set up the chain of recipients. The set of recipients is generated from the Flashpoint autoscaler's locality-aware Pareto-optimal greedy algorithm. The controller then sends chain parameters to each node. These parameters include the ID of the DL model to be transferred and the address of the next node if it is not the end of the chain. Once the chain has been established, ② the controller initiates the transfer of the model in a TCP stream. The recipient, upon receiving each packet, then copies the packets in its memory and forwards the data to next recipient in the chain. This process continues along the chain of recipients until the last recipient has received the model.

The TCP chaining protocol employs flow control mechanisms to ensure efficient network usage. The protocol monitors network conditions, such as network congestion, and adjusts the transfer rate of the model accordingly to minimize network congestion and reduce transfer time. Since Flashpoint is designed for data center usage with stable connections, fault tolerance mechanisms are not a critical requirement. However, the protocol is designed to be robust and handle certain types of network failures such as node failures or slow network conditions. In the case of a node failure, ③ the controller can re-route the transfer to the next recipient in the chain, ensuring the delivery of the model to the remaining recipients. The TCP chain managers in host machines can also communicate with the controller to request missing packets.

In summary, the TCP chaining protocol provides a reliable and efficient solution for the transfer of deep learning models to multiple recipients in a data center environment. By leveraging the reliable delivery mechanism of TCP and utilizing a chaining mechanism, the protocol ensures the simultaneous transfer of the model to multiple recipients, with efficient network usage and robustness against certain types of network failures.

Chapter 7

Overall System Design

In this chapter, we outline the design of the Flashpoint system that incorporates the three main solutions designed in the previous sections. Flashpoint affects the general serverless computing system architecture by modifying the autoscaler and adding a profiler.

7.1 Components and Control Flow

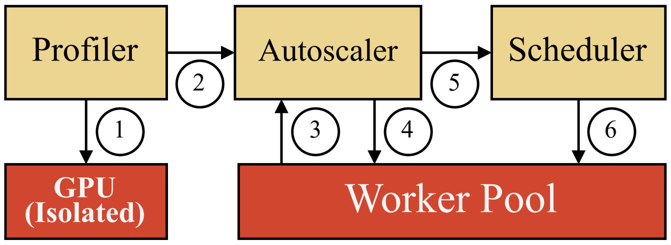


Figure 7.1: The components and control flow in Flashpoint.

The components of Flashpoint and their interactions are described in Fig. 7.1. The main components are the scheduler, profiler, and autoscaler.

When a new DL model is received by the system, (1) the profiler uses an isolated GPU to measure execution time, working process size, compute usage, etc. for new DL models. Then, (2) the profiler sends the data to the autoscaler. Meanwhile, (3) the autoscaler regularly receives target metric (e.g. queue length) and DL model status information on active workers for DL models. In a separate loop, (4) the autoscaler applies an autoscaling policy to scale DL model replicas up or down. Whenever there are updates, ⑤ the autoscaler informs the scheduler of new or removed instances for a DL model. While steps ① to ⑤ occur, ⑥ the scheduler asynchronously assigns requests for a DL model to their respective warm nodes. A warm node is a GPU that is ready to serve requests for the DL model.

Given our observation that the GPU compute is the bottleneck in scaling, we make the design decision of delegating new instance allocation and sourcing decisions to a centralized autoscaler. This offloads tasks such as reserving memory and performing cold starts from the critical path of serving requests. Through this, the compute power in GPU nodes are maximally utilized to perform inferences instead of maintaining additional state and performing graph searches to identify DL model sources.

The autoscaler and profiler are implemented in the simulator using Python with 2.8K and 2.3K Lines of Code (LOC) respectively.

Overall, the developer impact of integrating these techniques are mainly for the technique of model partitioning. To enable the use of model partitioning the DL model's layers and respective intermediate data must be defined in code. Meanwhile, locality and multicasting has minimal impact on the developer as the changes required are at the serverless platform level. The serverless platform must add support for the controller and add the downloads and connections manager agent in the host machines. These altogether control the download sources and network transfer protocols used for downloading the model.

7.2 Isolation Properties

In Flashpoint, each host machine uses a single container runtime to manage requests for different DL models among its GPUs. This level of isolation provides performance improvements in terms of reducing container instantiation overheads. In contrast, systems that isolate each workload in separate containers provide stronger isolation properties, but with greater instantiation overheads which are incurred as new DL models are loaded on a host machine.

While a container with request load balancing logic can be instantiated quickly on the host machine, the cold start process for the DL model must still be completed separately. The overheads of downloading the DL model, loading it on the host machine, and sending the tensor modules to the GPU, which are in the order of seconds, dominate container instantiation time, which are in the order of milliseconds [101].

Table 7.1 Metrics recorded by Flashpoint.

Metric	Description
1. System Information	
system_structure	Number of host machines and number of GPUs per host machine
2. One-shot Model Profile	
isolated latency	Average execution time on an isolated GPU in milliseconds
threshold	Additional percentage to accept run-time performance degradation
deadline	Inference deadline in ms, based on isolated latency and threshold
static_memory	Size of the model in GB
run-time_memory	Additional memory usage of model during run-time in GB
3. Continuously Profiled Model Information	
execution_time	Moving average of measured request execution time
download_time	Moving average of model download time
load_time	Moving average of loading model on host time
send_to_gpu_time	Moving average of sending model to GPU time

7.3 Implementation

7.3.1 Autoscaler

The logic for scaling up in the autoscaler is shown in Listing 7.1. The algorithm first adjusts the amount of replicas to scale up by using the maximum amount in the deployment configuration. Then, it acquires a partitioning plan based on the current state of the system, the workload, and target increase in replicas. Afterwards, the algorithm selects the receiver GPUs based on the partition plan using $get\_receivers()$, which also checks if the GPU and its encompassing host machine have the memory and compute capacity required to host the DL model. It then collects the senders that have the target DL model, which defaults to the cloud container registry. The sender hosts are then mapped to an approximately even list of receiver host machines. This mapping is used to initiate the multicast transfer. The multicast transfer occurs in three stages. First, a multicast group address is selected and receiver hosts are instructed to register their IP in that address. Subsequently, the data transfer is performed. Finally, the machines are deregistered from the group address and the address is released.

```python
# autoscaler.py
def scale_up(self, model_id: UUID, scale: int):
    """
    model_id: Identifier of the DL model to scale up.
    scale: Nat of how many replicas to scale up by.
    """
    config = self.functions_deployments[model_id].scaling_config
    current_replicas = self.replica_count[model_id]

    # Limit max by maximum scale config
    scale = math.min(scale, config.scale_max - current_replicas)
    # Do nothing if nothing to scale up by
    if scale == 0: return

    # Prepare and send multicasts
    # Select partitioning plan
    partition_plan = self.get_partition_plan(model_id, scale)

    # Select GPUs by locality (with Pareto-optimal greedy algorithm)
    # Also reserves the resources as they are selected
    receiver_gpus, receiver_hosts_set = self.get_receivers(model_id, scale, partition_plan)

    # Acquire the senders with the model including cloud storage
    sender_hosts_set = self.get_senders(model_id)

    # Create a map of senders and receivers
    sender_receiver_map = self.map_senders_to_receivers(sender_hosts_set, receiver_hosts_set)

    # Create the flows
    flows = self.create_multicast_flows(sender_receiver_map, model_id)
    for flow in flows:
        flow.start()

    # Trigger the rest of the initialization process in the receiver_gpu
    reverse_sender_receiver_map = self.get_reverse_map(sender_receiver_map)
    for receiver_gpu in receiver_gpus:
        receiver_host = receiver_gpu.parent
        sender_host = reverse_sender_receiver_map.get(receiver_host, receiver_host)
        self.deploy_replica(model_id, receiver_gpu, sender_host)
```

Listing 7.1: Logic for scaling up with locality-awareness, multicasting, and model partitioning in Flashpoint

7.3.2 Profiler

Flashpoint gathers three categories of information described in Table 7.1. System information is gathered upon setup where the network topology, number of hosts, and number of GPUs in each host are acquired. One-shot model profile information includes the isolated latency, threshold, deadline, static model memory size, and run-time memory consumption of the DL model. Finally, continuously profiled model information includes the average execution time, download time, load time, and send to GPU time for the DL model. The profiled information is used for determining whether the GPU and its respective host machine have the capacity to host the model.

Chapter 8

Evaluation of Flashpoint

In this chapter, we compare the performance of the techniques used in Flashpoint against other systems. We first evaluate each technique against related baselines and state-of-the-art systems using physical experiments and prototype implementations. We also evaluate the practicality of the simulations by comparing its results with the physical experiments. Then, we perform evaluations across techniques through full-scale simulations using parameters from the physical experiments.

8.1 Evaluation of Model Partitioning

In this section, we evaluate the technique of model partitioning used in Flashpoint. We use the analytical model and partition solver developed in Section 4.2.3 to compare the improvements that Flashpoint achieves for partitioned models at different numbers of requests served by the partitioned model. Model partitioning is the technique where we divide a DL model into parts to download them in parallel in a GPU chain Through these divisions, Flashpoint also loads these parts on the host machines and sends them to the respective GPUs in parallel. This technique introduces a trade-off of increased inference durations due to intermediate data being transferred across GPUs.

8.1.1 Optimality of Model Partitioning Schemes

In this subsection, we compare Flashpoint's model partitioning solver using the analytical model developed in Section 4.2.3 against state-of-the-art systems. The measurements of cold start durations, execution times, and input sizes used in the analytical model were acquired using servers with Nvidia A10 GPUs (24 GB VRAM, 64 GBps PCIe interconnect), 30 Intel(R) Xeon(R) Platinum 8358 CPUs, 200 GiB of RAM, and 50 Gbps network connections in the us-west-1 region from Lambda Labs. The cloud storage used is an AWS S3 bucket also in the us-west-1 North California region. The DL model is then downloaded from cloud storage to the cluster through the internet.

Fig. 8.1 shows the optimal number of cuts ($p - 1$) for the different DL models for the expected range of number of requests x based on the analytical model developed in Section 4.2.1. The results show that the optimal value of p does not change rapidly across the expected values of x. These results demonstrate that model partitioning is a feasible approach to consistently achieve greater performance than the default of full DL models in serverless.

Fig. 8.1 also shows the percentage benefit in average end-to-end latencies for the optimal number of cuts at different values of x. The figure shows that the partition scheme selected by Flashpoint consistently outperforms baseline model partitioning policies in terms of reduction of average end-to-end latencies. In particular, for CodeBERT, BART, and GPT-2, Flashpoint always selects partition plans that are different from the baselines and always achieves a better average end-to-end latency. For ALBERT, Flashpoint decides not to partition the model at larger values of x requests, while the baselines perform cuts and suffer from network transfer delays, leading to worse performance than a deploying full replicas. For DialoGPT, the model performs different cuts, but achieves approximately the same benefit as baselines. Finally, for T5, Flashpoint selects nearly the same number of partitions as Gillis, and fewer than SerFer. However, Flashpoint outperforms Gillis by also considering the reduction in cold start duration. For this model, Flashpoint achieves similar results as SerFer with fewer partitions. Overall, the evaluation shows that the model partitioning baselines achieve suboptimal partitioning at different workload scenarios. This is attributed to the lack of consideration of system and workload state.

8.1.2 Performance of Model Partitioning

We implemented and deployed Flashpoint's automated model partitioning technique on a Lambda Labs testbed and compared it to an implementation without model partitioning. The implementations were done using Python and consists of 3K Lines of Code (LOC).

In the physical implementation, the following were used: 10 servers with one Nvidia A10 GPU (24 GB VRAM, 64 GBps PCIe interconnect), 30 Intel(R) Xeon(R) Platinum 8358 CPUs, 200 GiB of RAM, and 50 Gbps network connections. The cloud storage used

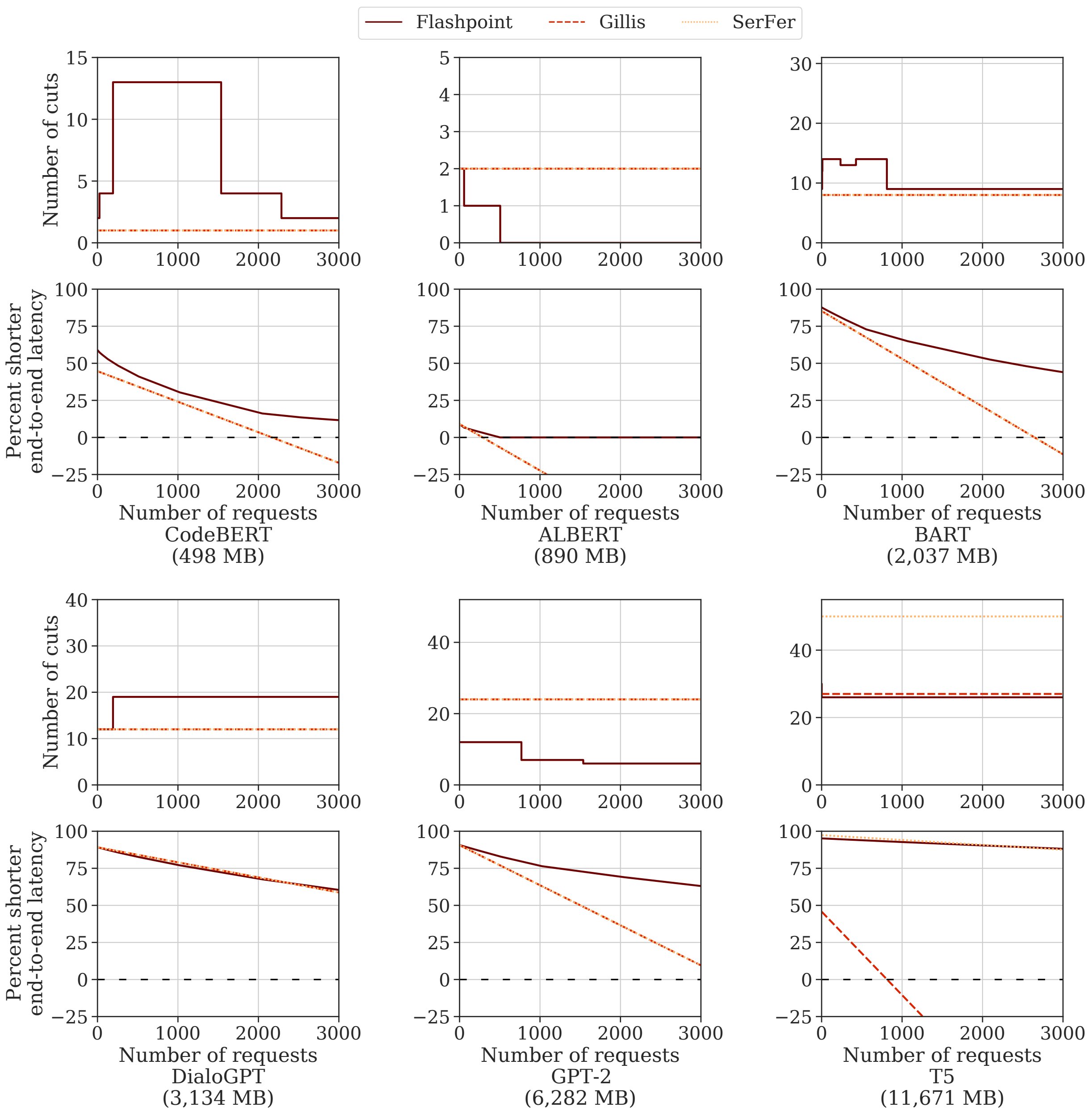


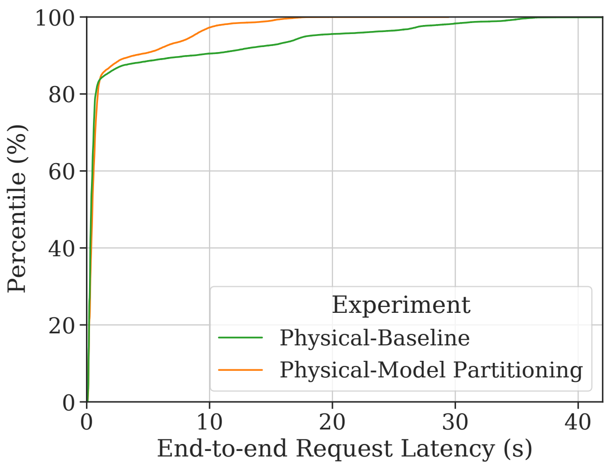
Figure 8.1: Number of cuts $(p-1)$ and percentage benefit on average end-to-end request latencies for optimal partitions for different DL models for different values of x based on the analytical model.

is an AWS S3 bucket in the us-west-1 North California region. The DL model is then downloaded from cloud storage to the cluster through the internet. The workload used for this deployment is a 15% scaled down version of the Twitter workload due to the scaled down number of servers. A separate server is used to host the controller and generate the workload. The same configurations are used for the simulation.

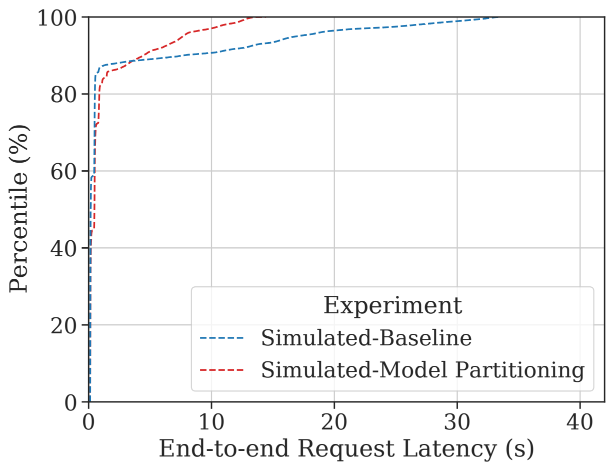
Scaling down the infrastructure configuration introduces a limitation that optimal partition schemes requiring a high number of GPUs cannot be used. For example, due to only having 10 machines, partition schemes that use more than 10 machines are not considered. Using the memoized solutions, the best partition plan given the limited number of machines is used instead. In addition, fewer machines are requested when scaling up due to the smaller absolute gap between valleys and peaks in the arrival rate of requests in the scaled down workload.

The autoscaler used in the deployment is based on Atoll which uses the 99th percentile arrival rate of requests scaled by the average request execution time to determine the desired number of replicas. This autoscaler is selected due to it selecting the desired number of replicas independent of the current state of the system, leading to simpler, more straightforward causes of improvements. The request scheduling in the deployment mimics a Sparrow scheduler, where the first available worker receives the request. This is performed through random selection of available worker nodes. A worker is considered available when it is not processing any request. When all workers are busy, the request blocks until a worker is made available.

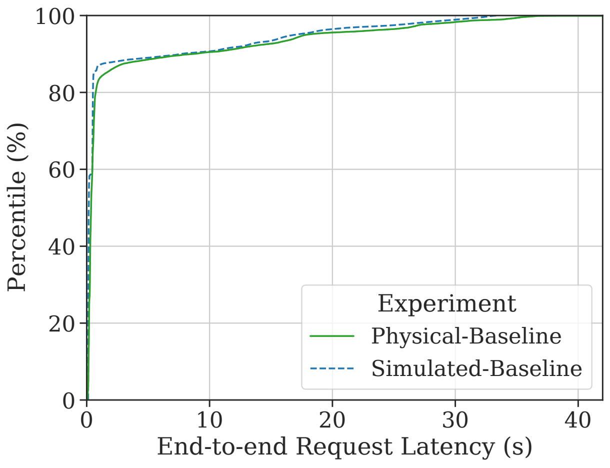
Fig. 8.2a shows the cumulative distribution function (CDF) of end-to-end request latencies for the baseline system and model partitioning in the physical deployment for the largest DL model (T5-3B) and the Twitter workload. The results show that the average end-to-end request latency for the baseline framework is 2.61 seconds while model partitioning achieves 1.39 seconds, which is a 46.74% reduction. These improvements are attributed to the shorter cold start duration that model partitioning achieves. These shorted cold starts enable new GPUs to serve requests sooner than their baseline counterpart and reduce waiting times for the requests. The partition plan, which determines the number of parts in the group, is selected by Flashpoint's partition solver. This selection is optimized to maximize the benefit from the cold start reduction while minimizing the penalty of increased inference times. In addition, by performing model completion, Flashpoint converges to using full models, further limiting the upper-bound of performance of the system to the level that the baseline system achieves.



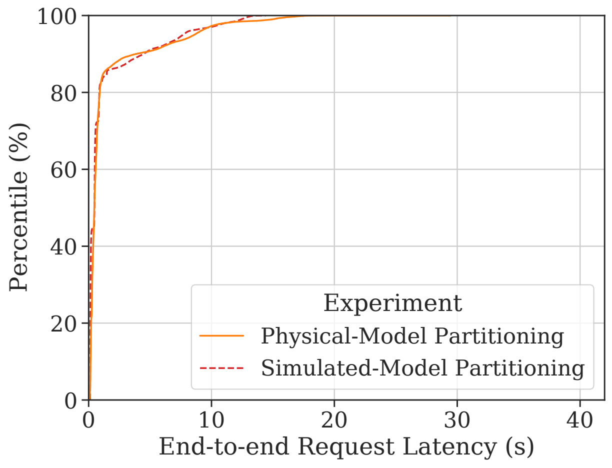
(a) Comparison of baseline and automated model partitioning on physical deployment.



(b) Comparison of baseline and automated model partitioning on simulation.



(c) Comparison of physical baseline and simulated baseline.



(d) Comparison of physical and simulated automated model partitioning.

Figure 8.2: Cumulative distribution function (CDF) of end-to-end request latencies for the physical deployment and simulation of model partitioning for T5-3B for the Twitter workload.

8.1.3 Simulator Practicality

Fig. 8.2 also shows the cumulative distribution function (CDF) of end-to-end request latencies for the simulation of model partitioning and the baseline system for the T5-3B DL model and the Twitter workload. From the evaluation, we observe that the average end-to-end request latency for the baseline framework in the physical system is 2.61 seconds while an equivalent simulation results in 2.23 seconds. Meanwhile, for Flashpoint's automated model partitioning in the physical system and simulation, they are 1.39 and 1.30 seconds respectively. The median absolute error of the physical deployment and simulation for automated model partitioning is 0.31 seconds while for the baseline, it is 0.20 seconds. We believe this small difference between the results of the physical system and the simulator provides evidence that the simulator generates valid results.

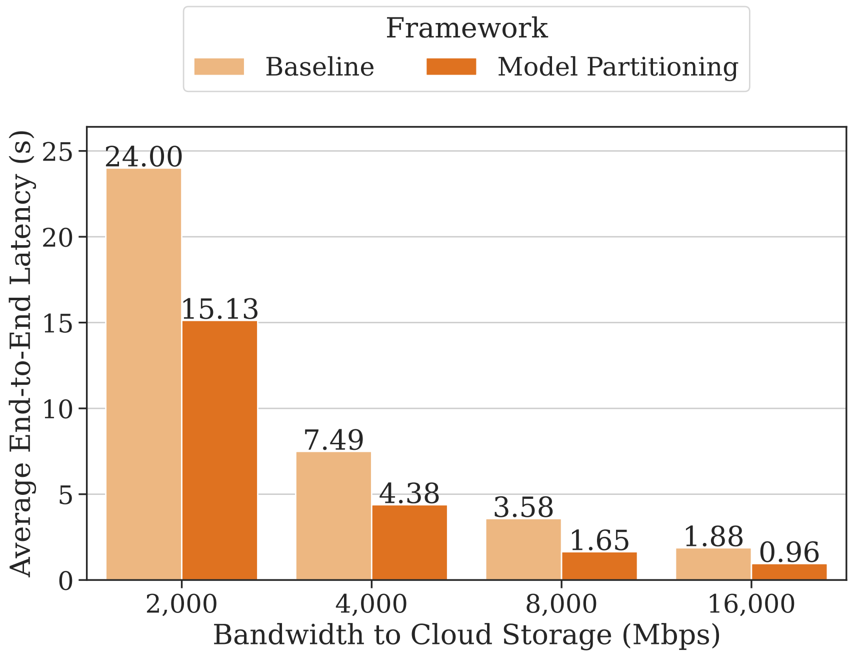


Figure 8.3: Comparisons of average end-to-end request latencies of a system using model partitioning and the baseline for different data center to cloud storage network bandwidths using the simulator.

Through the experiments, we identify two factors as being important in determining the average end-to-end request latencies. The first is the execution time of the function, while the other is the network bandwidth to cloud storage. The execution time determines the tail end-to-end request latencies wherein during a burst of requests, the later requests must wait for others ahead of it to complete. This waiting time is dependent on the execution time and the number of requests ahead of it. Meanwhile, the network bandwidth to cloud storage is the bottleneck for the cold start duration. The length of the cold start duration then impacts the end-to-end request latency for a portion of requests.

Fig. 8.3 shows comparisons of average end-to-end request latencies at different network bandwidths from the compute cluster to cloud storage for model partitioning and the baseline system using the simulator. In this experiment, the full scale of the workload is used with a full system with a spine-leaf data center structure containing 1600 GPUs across 200 host machines in 10 server racks. The results show that increasing the bandwidth reduces the percentage of requests with long waiting times as well as the longest tail latency. This is because the faster cold start duration enables the new replica to serve requests sooner and allows more requests to be served immediately when they are received.

In the case of 2000 Mbps of effective bandwidth from the compute cluster to cloud storage, a low percentage of requests are executed immediately. This is because the long duration of cold starts performed simultaneously cause congestion in the network. Without flow prioritization, these cold starts all compete for network bandwidth and take a prolonged time to complete, leading the system to have low service capacity for most of the requests

The results show that model partitioning improves end-to-end request latencies at each level of effective bandwidth tested. The average end-to-end latency is shown to be 1.59× shorter for model partitioning compared to the baseline for 2,000 Mbps in effective bandwidth based on measurements. The measured bandwidth for using AWS S3 clients from the Lambda Labs cluster is 2,203 Mbps across 10 tests.

We also evaluate the average cold start duration of model partitioning with different levels of maximum partitions of 1, 2, 4, 8, 16, 32, and 64. We perform this evaluation in the simulation with all 1600 simulated GPUs and at a 100% scale of the workload. Fig. 8.4 shows the cumulative distribution function of the end-to-end request latencies in the system for the different configurations. While not all model partitioning actions are limited by this cap, the results show that increasing the maximum number of partitions enables greater improvements in cold start durations.

8.2 Evaluation of Locality-Aware Autoscaling

In this section, we evaluate the technique of locality-aware autoscaling in Flashpoint. We first compare the cold start durations across different hierarchical sources for acquiring the DL model of cloud storage, remote host memory, and local host memory using physical experiments. Then we compare Flashpoint's locality-aware autoscaling to state-of-the-art systems through full-scale simulations parameterized with the physical experiments.

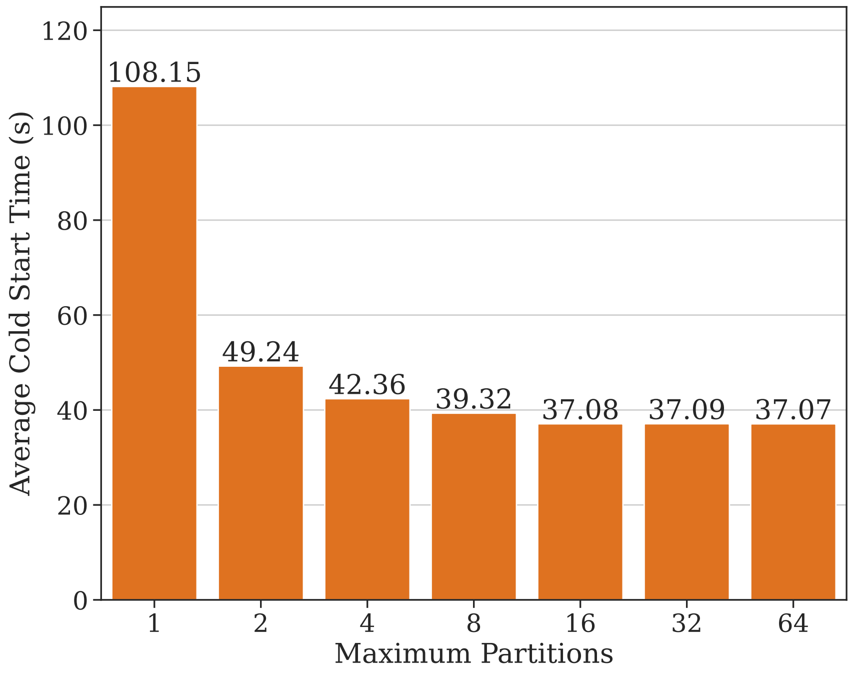


Figure 8.4: Comparisons of average cold start durations for model partitioning with different maximum number of partitions.

In the technique of locality-aware autoscaling, Flashpoint uses excess memory and network resources from host machines in the compute cluster to store DL models and download them into other host machines on-demand. Flashpoint also uses a Pareto-optimal greedy algorithm to optimize the selection of GPUs and maximize the improvements from faster hierarchical sources.

8.2.1 Evaluation of Hierarchical Sourcing

In this section, we evaluate the performance of different hierarchical sources using a physical deployment. The measurements for this evaluation are obtained using two p3.2xlarge EC2 instances with a single C++ S3 client in the us-east-2 (Ohio) region and a single AWS S3 bucket using in the same region. The EC2 instances each have 1 Nvidia V100 GPU with 16 GiB of memory, 8 vCPUs with 61 GiB of memory, and up to 10 Gbit/s network link.

Fig. 8.5 shows the mean cold start latencies of different hierarchical sources with a logarithmic scale. The results in the figure show the magnitude of speedups for cold starts from hierarchically local sources over remote. They show that sourcing the DL model from

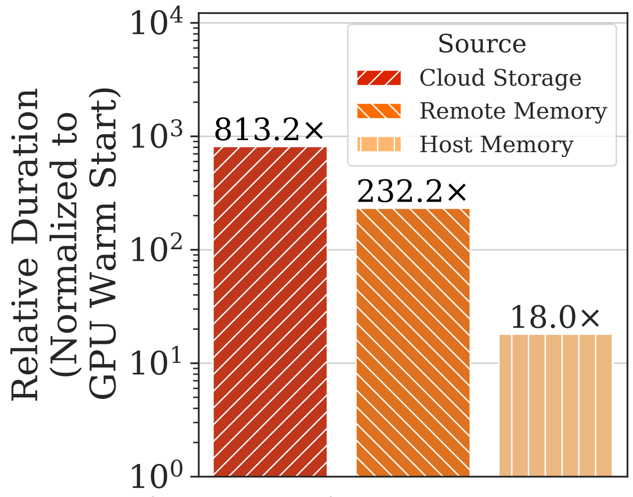


Figure 8.5: Relative duration of cold starts from hierarchical sources compared to warm starts (mean across DL models).

local host memory is 18.0× of a GPU warm start since it only needs to perform the third step in the cold start process of sending the model to the GPU. By exploiting underutilized resources with remote memory pooling, we source from remote host memory to achieve second-level cold start durations at 232.2× of a GPU warm start. Finally, the fallback of cloud storage is shown to take 813.2× of a GPU warm start. The remote memory and cloud storage sources are longer than the individual cold start steps shown in Fig. 3.6 since the former source also downloads from remote memory, while the latter includes steps of loading the model and send it to the GPU. Overall, hierarchical sourcing shows a 45.8× mean speedup for local copies over default cloud storage downloads. The effective host-to-host download bandwidth for the experiment is 7,506.89 Mbps on average for 5 tests on the Lambda Labs environment. This environment is using two machines with one Nvidia A10 GPU each. We then validate that the simulator achieves cold start durations for these hierarchical sources within 5% of the physical experiments.

8.2.2 Evaluation of Locality-Aware Autoscaling Compared to Other Systems

In this subsection, we compare the performance of Flashpoint's locality-aware autoscaling to state-of-the-art and baseline systems. For this evaluation, the following configurations are used:

Baselines. We use the following system to compare against Flashpoint. **Baseline** is the system without any optimizations. The download source is cloud storage using AWS S3. **Atoll** [90] uses lazy eviction to keep DL model data in the memory of host machines. It selects machines from the pool of workers with the DL model data first, but does not keep the process running in the host machine. **GPU-enabled FaaS** [117] also performs lazy eviction and adds the use of caching the DL model data from the host machine to the GPU as well as prioritizing the selection of GPUs where the DL model data is present in the host machine. In comparison, Flashpoint provides these improvements and enables the use of the remote host memory of other machines in the compute cluster to download the DL model data from.

In addition, while [14] suggests using AWS Elastic Container Registry (ECR) using Docker to download function containers, in the case of large DL models such as T5, the cold start download durations are consistently longer than using AWS S3. This is because using ECR with Docker requires that the DL model is persistently stored in the machine, leading to additional overheads, whereas using AWS S3 with only in-memory data avoids these overheads. Thus, the use of ECR is omitted from the results.

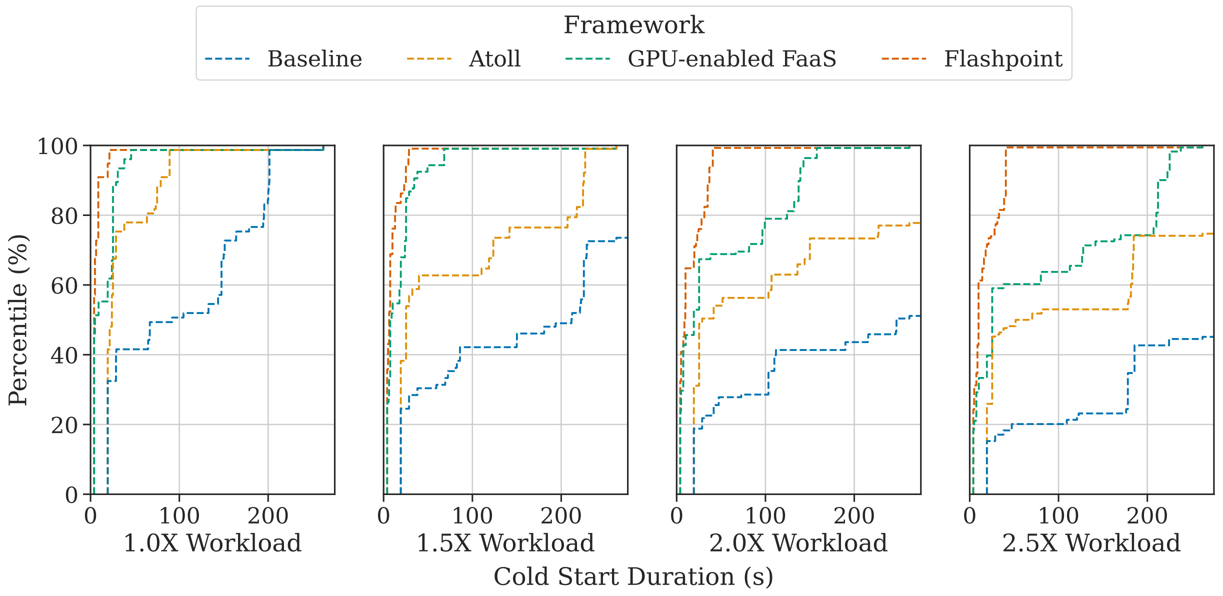
Simulation. The simulator is built using a customized version of faas-sim [75], which simulates network congestion through the framework, Ether [74], and is built on top of the Python simulation framework, SimPy [84].

In this evaluation, it simulates a fixed number of GPUs (1600) available within a data center with 8 GPUs per host machine. The structure of the network used is a spine-leaf architecture with 100 Gbit/s links from spine to leaf nodes and 50 Gbit/s links from leaf nodes to host machines with 20 host machines per leaf node. This work assumes homogeneous GPUs for simplicity and could be modified to incorporate the variety in compute capacity for heterogeneous GPUs.

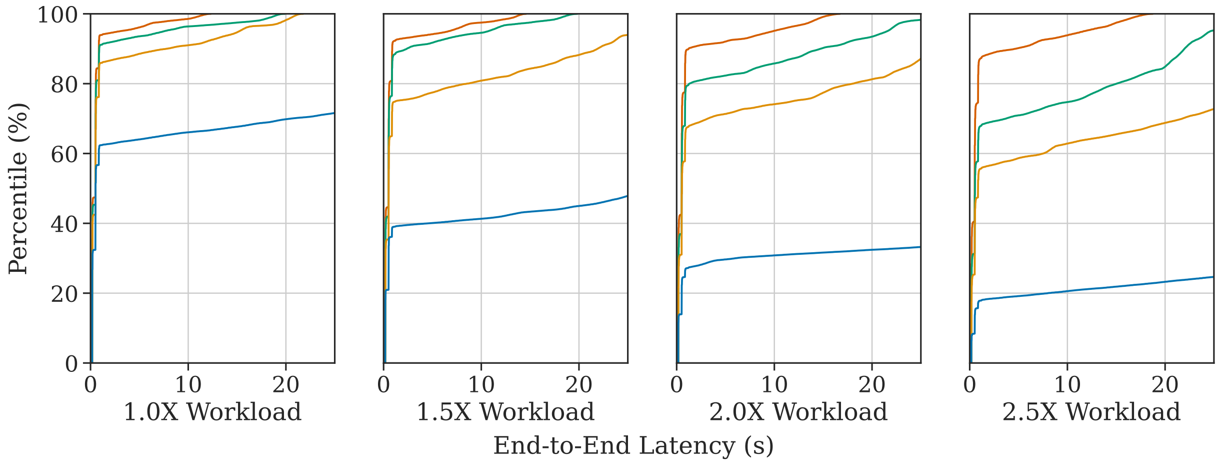
The execution times, loading times, sending to GPU times, network request overheads, and network bandwidths are parameterized by our physical experiments. The practicality of the overheads and parameterized used in the simulation was discussed earlier in Section 8.1.3. The simulation incorporates the time to perform autoscaling and scheduling decisions. This means that the actions do not take effect until time is incremented by the wall time it took to make the decisions.

In the simulation, each host machine is set to have the PyTorch base Docker image preloaded, meaning that a web server with PyTorch dependencies loaded is already running on each host machine and does not have to be re-downloaded every time a new DL model is initialized in a GPU of the host machine.

Fig. 8.6 shows the cumulative distribution function (CDF) of cold start durations and



(a) Cold start durations.



(b) End-to-end request latencies.

Figure 8.6: Comparison of different systems for different workload scales.

end-to-end request latencies for the baseline, state-of-the-art systems, and Flashpoint with only the technique of locality-aware autoscaling for increasing scale of the Twitter 1 hour workload using the request rate autoscaler and the T5 DL model. The results show that increasing the scale of the workload provides more opportunities for the contribution of locality-aware autoscaling in Flashpoint to reduce cold start durations.

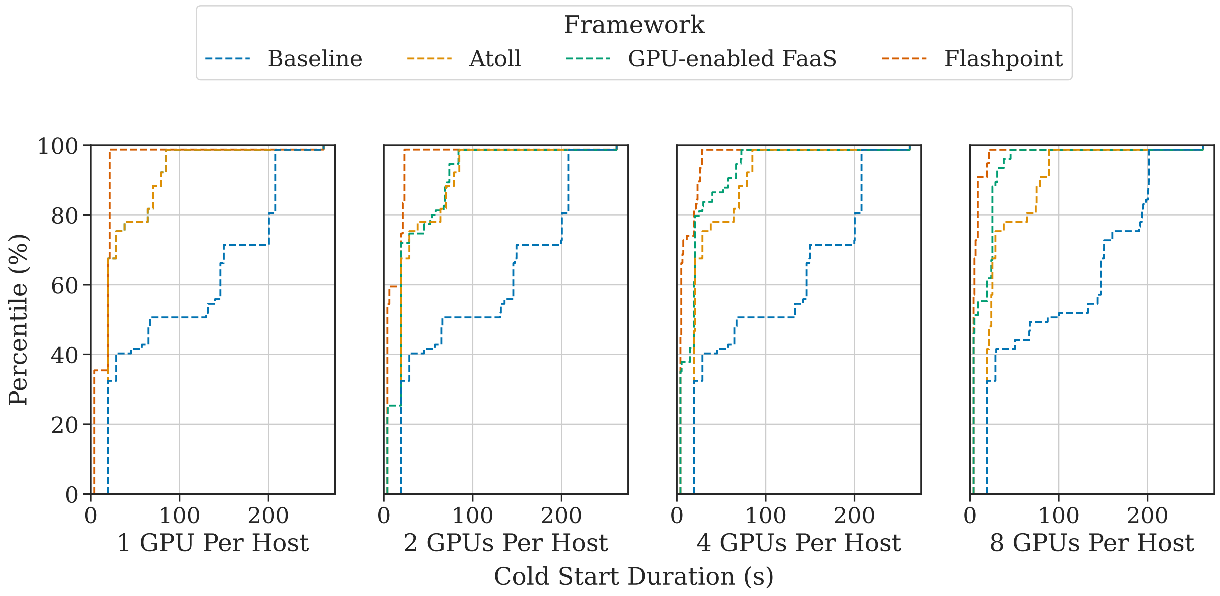
In the example of 2.5× workload scale in Fig. 8.6a, majority of cold start durations are bounded by the duration to download requests within the compute cluster, which has a higher network bandwidth and larger number of independent paths for downloading the DL model data than cloud storage. In comparison, other systems suffer from degraded cold start duration times due to requiring new host machines to download the data and congestion in the network from the compute cluster to cloud storage. Consequently, in Fig. 8.6b the improvements of Flashpoint in end-to-end request latencies compared to other systems become larger with increasing workload scale.

Meanwhile, Fig. 8.7 shows the CDF of cold start durations and end-to-end request latencies for increasing numbers of 1, 2, 4, and 8 GPUs per host for the Twitter 1 hour workload using the request rate autoscaler and the T5 DL model. The results in Fig. 8.7a with the CDF of cold start durations show that increasing the number of GPUs per host improves the performance of Flashpoint as well as GPU-enabled FaaS. This is because increasing this number enables more opportunities for the DL model data to be cached directly from the host machine to the GPU. Similarly, the improvements in end-to-end request latencies in Fig. 8.7b show similar improvements for both systems. We discuss the implications of the findings in Fig. 8.6 and 8.7 in the following section.

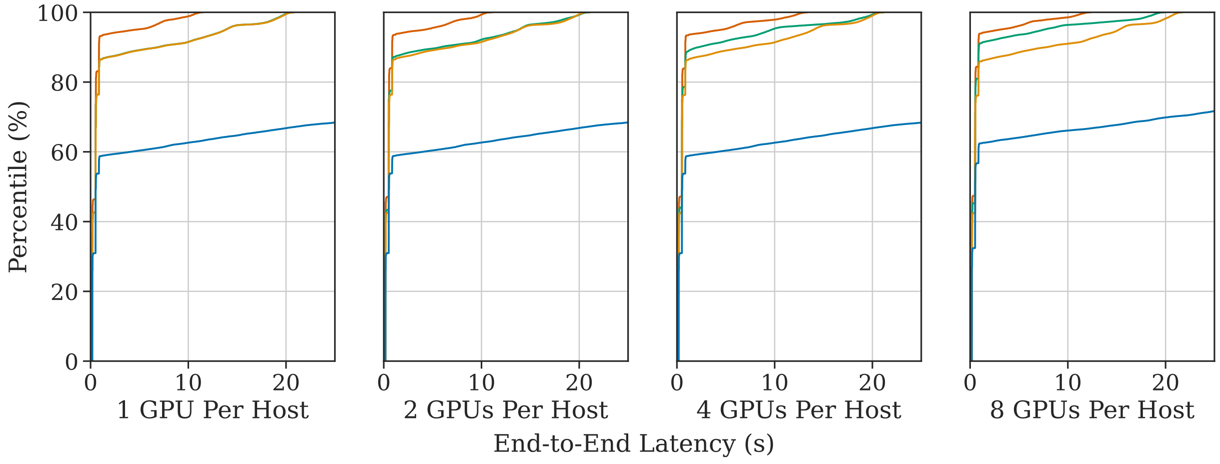
8.3 Evaluation of Network-Aware Autoscaling

In this section, we compare the network-aware autoscaling technique of Flashpoint to state-of-the-art and baseline protocols. In th evaluation, we demonstrate the practicality of Flashpoint's TCP chaining through a physical experiment. Flashpoint's network-aware autoscaling uses TCP chaining as a protocol of multicasting to reduce network congestion when large numbers of new replicas are simultaneously requested by the autoscaler.

We have shown in the characterization in Section 6.1.1 that cold start cases where host-host DL model transfers with a one-to-many ratio is a common case. With this, Fig. 8.8 shows the duration of cold starts for various levels of concurrency with different protocols for the T5-3B model. Other DL models show similar results. The figure includes an optimal line where the full network bandwidth is realized as throughput. The measurements for this physical experiment were acquired using a local testbed with 5 servers, each having 12 Intel® Xeon® E5-2620 v2 CPUs, 64 GiB of RAM, and 10 Gbit/s network connections.



(a) Cold start durations.



(b) End-to-end request latencies.

Figure 8.7: Comparison of different systems for different numbers of GPUs.

The figure shows that baseline protocols of using TCP and RDMA grow linearly in duration as more transfer requests occur simultaneously. This is because of network congestion with multiple unicasts from a source to different destinations share the uplink bandwidth

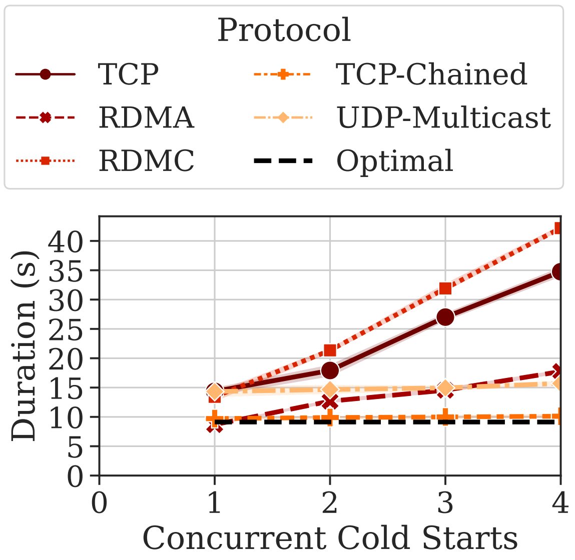


Figure 8.8: Duration of host-to-host cold starts compared to number of concurrent cold starts for T5-3B.

from the source to the first switch. Meanwhile, while UDP multicast can be used, reliability mechanisms designed ensure the complete file is sent correctly still lead to suboptimal scaling. Meanwhile, an RDMA-based reliable multicast protocol such as RDMC [15] is also suboptimal due to high synchronization overheads as from using RDMA unicasts with an overlay mesh. Additionally, multicasting using UDP is shown to have the ideal constant scaling; however, additional synchronization overheads needed to make it reliable cause it to be suboptimal in duration.

In contrast to the baseline protocols, the figure shows that chaining TCP flows allows near-optimal scaling at higher levels of concurrency, being within 9.6% of the optimal duration. TCP chaining is also shown to be 3.30× faster than the baseline TCP protocol at a concurrency level of 4. While a hardware-bound protocol such as RDMA could ideally be used to reduce CPU consumption on the host machines that deliver the DL models, the current state of the RDMA library, libibverbs [50], does not support the required primitives to chain transfers or multicast quickly and reliably.

8.4 Evaluation of Techniques

In this section, we evaluate the performance improvements of the Flashpoint system from each of the techniques used. We focus on the metrics of average end-to-end request latencies and average cold start durations. These metrics are important for the latency sensitive workloads that motivate the thesis. We evaluate the performance improvement of each technique individually and then in combination. The overall results show that Flashpoint improves both end-to-end request latencies and average cold start durations while consuming similar levels of resources as baseline systems.

The following configurations were used for the evaluation:

Deep Learning Models. Table 3.1, which was shown previously in Section 3, shows the six popular DL language models used in this measurement study. These are representative of various model sizes in the domain of Natural Language Processing (NLP). The models used include CodeBERT [34], ALBERT [55], BART [56], DialoGPT [116], GPT-2 [72], and T5 [73], which range from 499 MB to 11 GB in size. By testing with these models, we show that our observations hold for various model sizes.

Simulation. This evaluation uses simulations to measure the improvements of the techniques developed. The simulator is built using a customized version of faas-sim [75], which simulates network congestion through the framework, Ether [74], and is built on top of the Python simulation framework, SimPy [84].

In this evaluation, the simulation creates a fixed number of GPUs (1600) available within a data center with 8 GPUs per host machine. The structure of the network used is a spine-leaf architecture with 100 Gbit/s links from spine to leaf nodes and 50 Gbit/s links from leaf nodes to host machines with 20 host machines per leaf node. This work assumes homogeneous GPUs for simplicity and can be modified to incorporate the variety in compute capacity for heterogeneous GPUs.

The execution times, loading times, sending to GPU times, network request overheads, and network bandwidths are parameterized by our physical experiments. The practicality of the overheads and parameterized used in the simulation was discussed earlier in Section 8.1.2. The simulation incorporates the time to perform autoscaling and scheduling decisions. This means that the actions do not take effect until time is incremented by the wall time it took to make the decisions.

In the simulations, each host machine is set to have the PyTorch base Docker image preloaded, meaning that a web server with PyTorch dependencies loaded is already running on each host machine and does not have to be re-downloaded every time a new DL model is initialized in a GPU of the host machine.

Autoscalers. We use the autoscaling policy as a controlled variable. This indicates the target number of replicas at different times in the workload is determined only by the autoscaling policy and then implemented by Flashpoint. In these evaluations, we use four autoscaling policies to demonstrate the improvement of Flashpoint with different scaling decisions. These autoscalers are: TorchServe on Kubernetes (TSK), Atoll (ATL), OpenWhisk on Kubernetes (OWK), and AWS SageMaker (SM*). These baselines represent both state-of-the-art and commercial baseline systems.

TSK uses queue latency as its autoscaling target metric with a default target of 7 seconds. ATL similarly uses queue latency as a metric for autoscaling resources based on the arrival rate of requests. In the experiments in this thesis, ATL is set to use the arrival rate of requests. OWK relies on the Kubernetes horizontal pod autoscaler, which uses the CPU compute utilization of host machines with a default target utilization of 60%. In this thesis, this is modified to use GPU compute utilization. Finally, SM* uses compute utilization, invocations per instance, or a custom metric for its autoscaling policy. For our experiments, SM* is set to use invocations per instance, which is the number of requests served by each replica on average.

In order to fairly evaluate and isolate the improvements of the techniques from other factors, the thresholds for the target metric of the autoscalers are modified to have the autoscaler use approximately equal amounts of resources in average replicas per second within 5% of the baseline. Changing this parameter to scale up to more instances generally improves performance while incurring higher costs, while reducing this generally reduces performance with lower costs. Meanwhile, the number of replicas that the autoscaler scales up by is a factor of the number of replicas currently running. Since the techniques used in Flashpoint reduce cold start durations, this count of running replicas increases rapidly and causes the autoscaler to scale up to a larger number of instances than the baseline. By adjusting the threshold of the autoscaler based on the technique, the number of replicas used in Flashpoint matches that of the baseline and improvements from increased replica counts are eliminated.

Workloads. An hour-long trace of the Twitter workload [96] and Microsoft Azure workload [87] are used for the evaluation. The Twitter trace has a median of 57 requests per second and peak-to-average ratio of 4.68. Meanwhile, the median and peak-to-median ratio for the Microsoft Azure trace is 18 requests per second and 2.54 respectively. These public workloads are used due to the lack of public DL inference serving traces. However, the user-facing nature of the related products are expected to result in similar access patterns to low-latency DL services such as ChatGPT and GitHub Copilot. Compared to the Twitter trace, the Microsoft Azure trace is a low-frequency workload that is used to illustrate the performance of the system when scaling up from a low number of replicas.

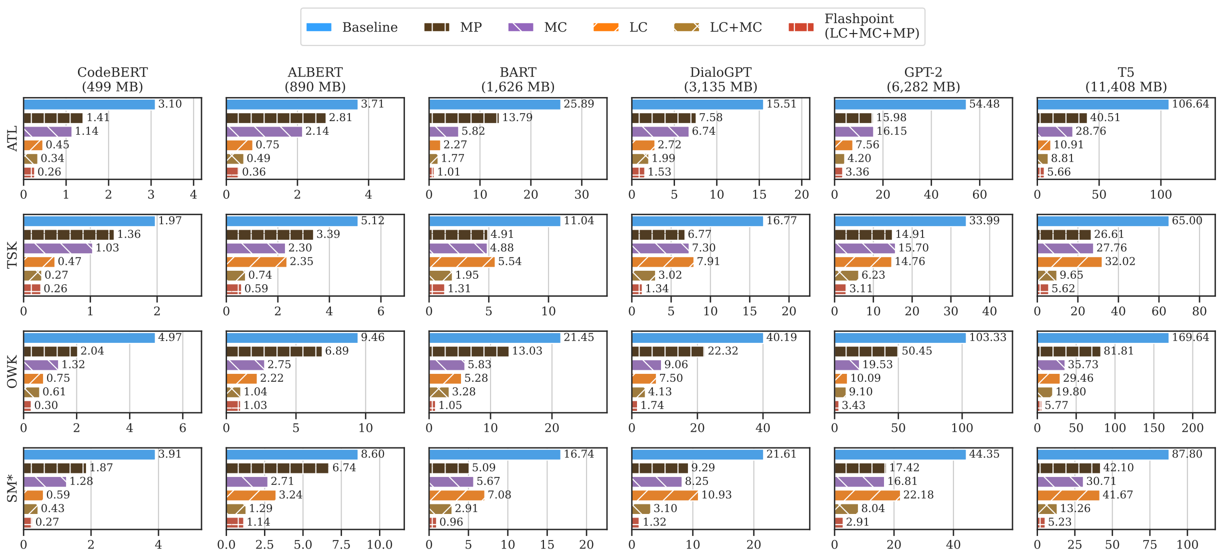
Request Scheduling. As the focus of this work is in autoscaling, we set the default load balancing protocol to have worker nodes pull from a global queue whenever compute capacity is available. This load balancing protocol reduces imbalance in comparison to a round-robin load balancer where older instances have a large number of requests scheduled while new instances created during a burst have fewer instances scheduled in the worker's queue. The load balancing protocol using a global queue represents a well-balanced load balancing protocol. This protocol improves the performance of Flashpoint and the baselines equally. Furthermore, when using model partitioning where parts of the request are executed sequentially on different machines, the machine does not consume a request from the upstream machine or queue until its current request is sent to the downstream machine and it has reserved the necessary resources for the request. This is to prevent slower machines from accumulating multiple requests while still allowing the machines to collectively serve multiple requests concurrently.

8.4.1 Evaluation of Individual Techniques

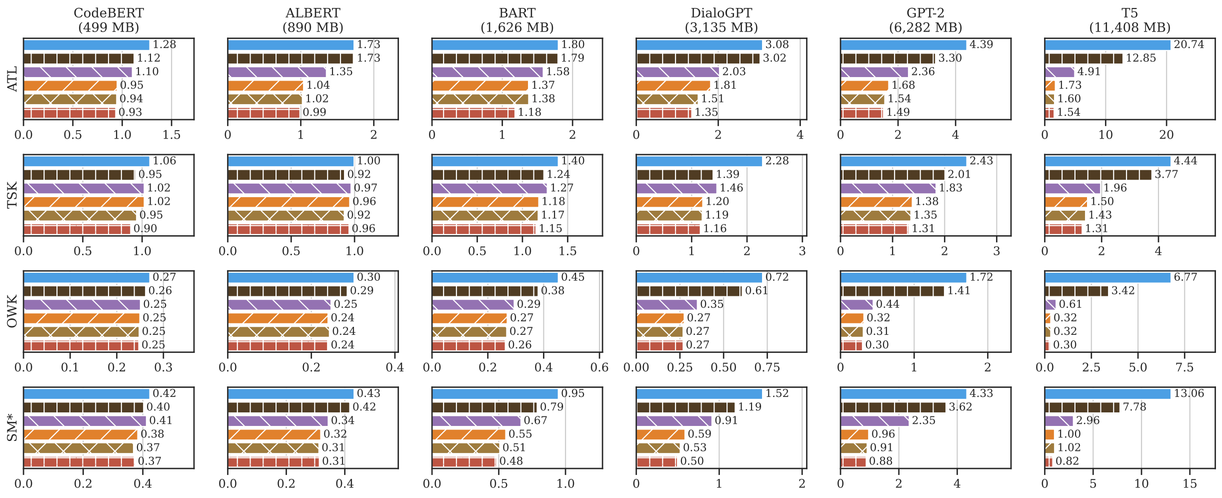
In this section, we evaluate the performance of the individual techniques of Model Partitioning (MP), Multicasting (MC) and Locality (LC) compared to the baseline system (BL) for each of the autoscalers. We use the metrics of average cold start duration and average end-to-end request latency. The resource usage in terms of average replicas per second for the different techniques are kept within 5% of the baseline by modifying the autoscaler threshold. The results show that the improvements provided by the techniques are consistent among different autoscalers and are more prevalent in larger models.

Fig. 8.9a shows the average cold start duration of individual techniques compared to the baseline for different autoscalers and DL models for the Twitter workload. Each row shows the results for the same autoscaler, while each column is for the same DL model.

Comparing different autoscalers, the request rate autoscaler in ATL and the invocations per instance autoscaler in SM* tend to scale up in bursts along with the request rate as it reacts to the workload with lagging indicator. Meanwhile, the queue latency autoscaler in TSK scales up in larger batches because the changes in the target metric is more delayed than in ATL and SM*. Meanwhile, the GPU compute utilization autoscaler in OWK is more conservative and over-provisions resources. In addition, with a default target utilization of 60%, the maximum scale-up ratio of target number of replicas to current



(a) Average cold start durations of techniques.



(b) Average end-to-end request latencies of techniques.

Figure 8.9: Results for the Twitter workload.

number of replicas is approximately two times, leading to slower scaling up compared to other autoscalers. Increasing the value of this target utilization reduces the amount of replicas over-provisioned for the workload but also increases the end-to-end request latency as fewer machines are available to handle bursts in requests. The opposite occurs for reducing this target value.

Across different autoscalers, the results show the different techniques individually reduce cold start durations with increasing improvements from MP, MC, then LC. For example, for the T5 DL model using the baseline ATL autoscaler, the average cold start duration is 3.10 seconds. Meanwhile, for MP, MC, and LC, it is 1.41 seconds, 1.14 seconds, then 0.45 seconds. These are equal to 2.19×, 2.72×, and 6.82× reductions in latency for the respective techniques. The results show that the mean of the reduction in cold start durations is 2.12×, 3.09×, and 15.41× for MP, MC, and LC respectively.

Both MP and MC result in similar reductions in cold start durations when using reactive autoscalers such as ATL, and SM*. This is due to both techniques addressing the network bottleneck of downloading n full copies of the DL model to n machines and changing it such that effectively one copy is distributed among the n machines.

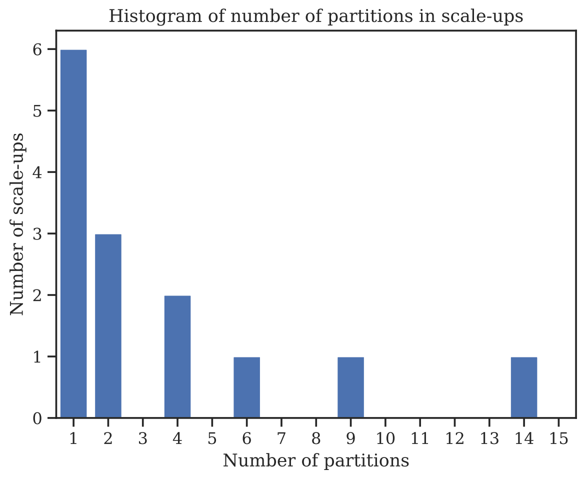
In contrast to MC, MP results in shorter cold starts when using autoscalers that batch cold starts in larger numbers such as TSK. The larger batches enable more machines to download smaller parts, leading to greater parallelism and earlier starting times for executing requests compared to waiting to download the full model. Contrary to this, in the case of the OWK autoscaler, MP provides lower reductions in cold start duration due to fewer replicas to scale up by and fewer opportunities for parallelism. Meanwhile, LC outperforms both MP and MC by directly alleviating the network bottleneck from cloud storage to the compute cluster, leading to greater reductions in the download time for the DL model.

Fig. 8.9b shows the average end-to-end request latency of the individual techniques compared to the baseline for different autoscalers for the Twitter workload. Across different autoscalers, the results show the different techniques individually contribute to latency reductions with increasing improvements from MP, MC, then LC. The improvements in average end-to-end request latency are attributed to the reductions in cold start durations. For example, for the T5 DL model using the baseline ATL autoscaler, the average end-to-end request latency is 1.28 seconds. Meanwhile, for MP, MC, and LC, it is 1.12 seconds, 1.10 seconds, then 0.95 seconds. These are equal to 1.14×, 1.16×, and 1.35× reductions in latency for the respective techniques. Taking the mean of improvements across autoscalers leads to 1.23×, 2.17×, and 4.07× reductions in latency from the baseline for MP, MC, and LC respectively.

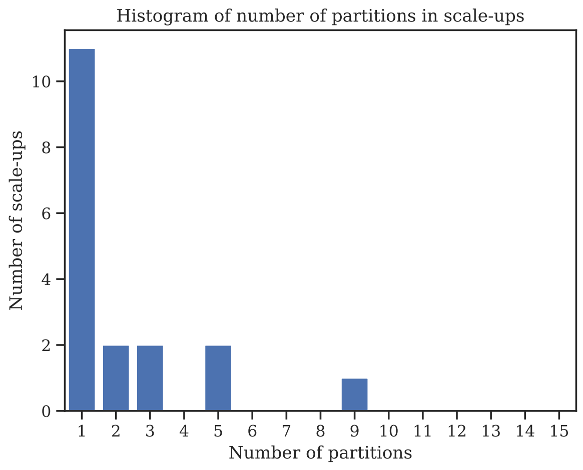
Compared to the trends in average cold start durations, the improvements in average end-to-end latency are greater for larger models. This is because as the DL model becomes smaller, the ratio of the cold start duration compared to the inference time becomes smaller. At this lower level, network overhead requests become more dominant and reduces the impact of cold start reductions.

There are also important differences in the improvements in average end-to-end request latencies achieved by the techniques compared to their average cold start durations. For example, while MP and MC can offer similar improvements in cold start durations, MP incurs longer inference times due to additional network transfer delays of intermediate data between machines while MC does not. This leads to MP providing having longer average end-to-end latencies compared to MC. In contrast, LC continues to achieve the best average end-to-end latencies among all techniques in majority of the experiments.

Analysis of Model Partitioning Results



(a) T5-3B using the ATL autoscaler for the Twitter workload.



(b) DialoGPT using the ATL autoscaler for the Twitter workload.

Figure 8.10: Examples of number of partitions used for scale up actions.

Fig. 8.10 shows the number of partitions used for scale up actions for example configurations. These values demonstrate the effect of model partitioning on the performance of the system. Fig. 8.10a shows the number of partitions used for the T5-3B model using the ATL autoscaler for the Twitter workload. In this configuration, the average end-to-end request latency of using model partitioning is 12.85 seconds, while the baseline is 20.74 seconds. This is a 38.04% reduction in latency. Meanwhile, Fig. 8.10b shows the values for the DialoGPT model. In this configuration, the average end-to-end request latency of using model partitioning is 3.02 seconds, while the baseline is 3.08 seconds, leading to a 1.95% reduction in latency.

In comparison, the configuration using the T5-3B model selects a larger number of partitions, which achieves greater improvements than for the DialoGPT model, which uses fewer number of partitions. Specifically, over 50% of scale up actions only use one partition for the DialoGPT model. This is due to the cost of increased execution times being too large as determined by Flashpoint's model partitioning algorithm. Therefore, less than half of the actions can benefit from model partitioning.

8.4.2 Evaluation of Combined Techniques

In this section, we evaluate the performance improvements in cold start durations and end-to-end request latencies when incrementally combining the techniques in Flashpoint. We start with the best performing technique of LC, then add MC multicasting. Then, on top of these, we add automated model partitioning.

Fig. 8.9a also shows the performance of different combinations of techniques in terms of average end-to-end request latency for different autoscalers and DL models for the Twitter workload. Meanwhile, 8.9b also shows the average cold start durations for different combinations of techniques for different autoscalers and DL models similar to the previous section.

Beginning with locality-awareness (LC) using the ATL autoscaler and the T5 DL model, the average cold start duration is 0.45 seconds and the average end-to-end request latency is 0.95 seconds. By introducing this technique, the download path of the DL model is made to be independent from each other, avoiding a network bottleneck from cloud storage. This is because the source of the DL model parts no longer come solely from cloud storage and can be other machines in the compute cluster. This route independence relieves the network bottleneck and improves download times for the DL model.

Then, by adding multicasting to locality (LC+MC), the average cold start duration becomes 0.34 seconds. This is a 26.18% improvement in duration compared to only using LC. Network-awareness improves the system's performance by reducing the amount of redundant packet transfers from source nodes to multiple receiver nodes when downloading DL model parts. With the addition of this technique, Flashpoint achieves a average cold start duration of 0.94 seconds, which is a 0.53% improvement compared to using only LC.

Finally, while both multicasting and locality reduce the download time for initializing DL model, the time to load the DL model and send it to the GPU then becomes a greater proportion of the whole cold start duration. The addition of model partition to Flashpoint (LC+MC+MP) then addresses this issue by reducing the amount of data that is downloaded to the compute cluster and reducing the startup time of each partition in the DL model. This results in an average cold start duration of 0.26 seconds, which is a 42.67% improvement. Automated model partitioning also carefully avoids excessive inference time penalties from transferring intermediate data between GPUs by calculating the optimal number of partitions to use given the state of the system and workload. The result is an improvement of 1.55% for an average end-to-end request latency of 0.93 seconds.

Taking the mean of improvements in average cold start durations across autoscalers and DL models result in a 75.21% and 75.21% reduction for LC+MC and LC+MC+MP respectively. These then result in respective improvements of average end-to-end request latencies of 8.22% and 8.22% compared to using only LC.

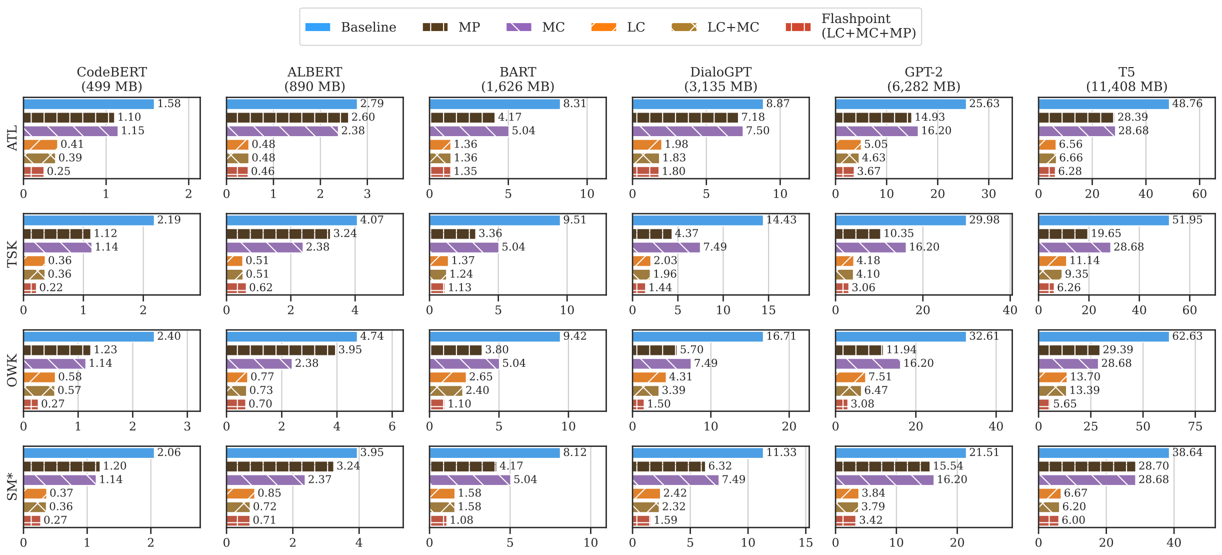
Together with all techniques, Flashpoint achieves a 93.51% reduction in average cold start durations compared to the baseline, leading to a 75.42% reduction in average end-to-end request latencies across DL models and autoscalers for the Twitter workload. At the 99th percentile, the mean reduction in end-to-end request latencies is 66.90%.

Compared to the state-of-the-art, Flashpoint achieves a 78.46% reduction in average cold start durations and reductions of 20.63% and 19.69% for the average and 99th percentile end-to-end request latencies across DL models and autoscalers for the Twitter workload.

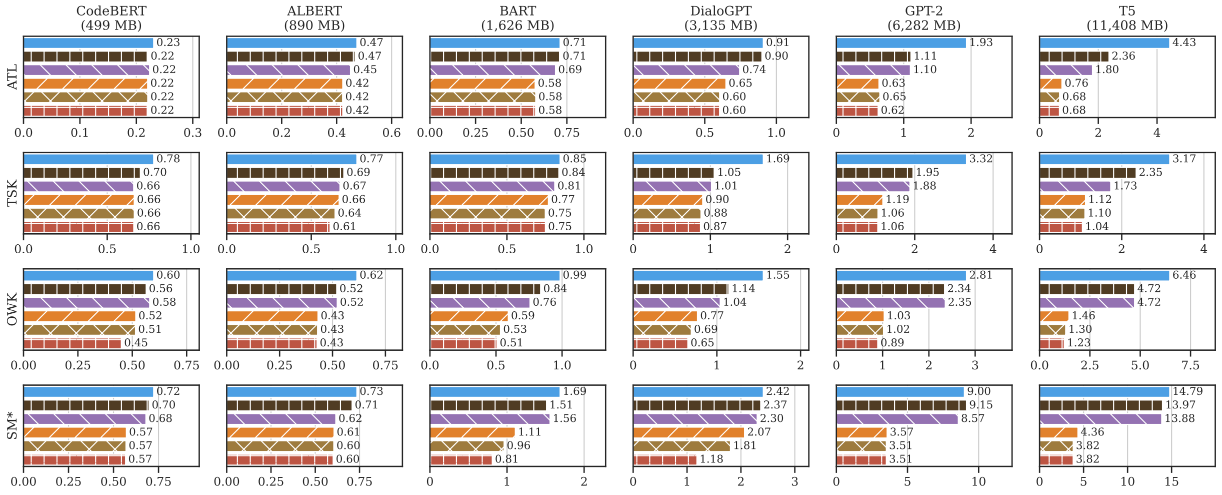
Fig. 8.11 shows the average cold start durations and average end-to-end request latencies for different techniques for the Microsoft Azure workload. The results show similar trends as the Twitter workload and reinforces the findings. The improvements are shown to be greater in the larger workload for Twitter than in Microsoft Azure.

For the Microsoft Azure workload, the mean of improvements in average cold start durations across autoscalers and DL models result in respective improvements for MP, MC, LC of 1.95×, 1.73×, 7.93×. These then result in respective improvements of average end-to-end request latencies of 1.22×, 1.30×, 2.28× respectively. Meanwhile for LC+MC and Flashpoint (LC+MC+MP), the improvements are 34.86%, and 34.86% for average cold start durations, and 10.13%, and 10.13% respectively for average end-to-end request latencies.

Compared to the baseline for the Azure workload, Flashpoint achieves a 87.39% reduction in average cold start durations compared to the baseline, leading to a 56.23% reduction in average end-to-end request latencies across DL models and autoscalers for the Azure workload. At the 99th percentile, the mean reduction in end-to-end request latencies is 56.67%. Then, compared to the state-of-the-art, the reductions in average cold start



(a) Average cold start durations of techniques.



(b) Average end-to-end request latencies of techniques.

Figure 8.11: Results for the Microsoft Azure workload.

durations, average end-to-end request latencies, and 99th percentile end-to-end request latencies are 68.50%, 14.54%, and 11.20% respectively.

Overall, Table 8.1 shows the performance improvements in average end-to-end request latencies for Flashpoint compared to GPU-enabled FaaS [117] and the baseline system.

Table 8.1 Percentage improvements in average end-to-end latency for Flashpoint over GPU-enabled FaaS and the baseline system for different workloads.

Model	Size (MB)	GPU-enabled FaaS		Baseline	
		Azure	Twitter	Azure	Twitter
CodeBERT	499	4.44%	7.24%	17.25%	16.52%
ALBERT	890	6.33%	8.17%	20.79%	27.02%
BART	1,626	14.52%	18.59%	37.44%	37.66%
DialoGPT	3,135	23.72%	26.18%	49.11%	60.05%
GPT-2	6,282	15.99%	19.44%	66.57%	74.12%
T5	11,408	19.08%	36.23%	78.64%	92.79%

These values are gathered by using the ratios of average end-to-end request latency of the compared system and that of Flashpoint for each autoscaler, then taking the mean across autoscalers, converted into percentage improvements. For example, the 92.79% improvement for Flashpoint compared to the baseline system for the Twitter workload and the T5 DL model means that the average end-to-end request latency for Flashpoint is smaller than the baseline by 92.79%. In the case of the T5 model for the baseline, Flashpoint provides greater improvements from 78.64% to 92.79% going from the smaller Azure workload to the larger Twitter workload. Meanwhile, for the Twitter workload, the improvements in Flashpoint compared to the baseline increases from 16.52% to 92.79% starting from the smaller CodeBERT DL model to the larger T5 DL model.

The trends observed are that on average, both larger workloads and larger DL models show greater improvements for Flashpoint compared to the state-of-the-art and baseline systems. Larger workloads cause more instances of scaling up, which provides more opportunities for Flashpoint to reduce cold start durations and reduce average end-to-end request latencies to a greater extend than other systems. Meanwhile, larger DL models also lead to longer cold start durations for all systems; however, Flashpoint is able to minimize these durations also for more scenarios than other systems, leading to greater improvements in performance. The importance of these trends are described later in Section 8.7. In the discussion, we examined trends and provided evidence to show that the growing sizes of DL models and growing demand for such models exceeds the growth in computational power. These observations highlight the need for employing the techniques used in Flashpoint to address the future demand for DL models effectively.

8.5 Resource Consumption

In this section, we describe the resource consumption of the systems in the experiments. Increasing the overall resource consumption of systems leads to lower average end-to-end request latencies but at higher costs, and vice versa. Throughout the experiments across techniques, we maintain resource consumption as a constant to observe the performance difference between techniques that is not attributable to just increasing the amount of resources used.

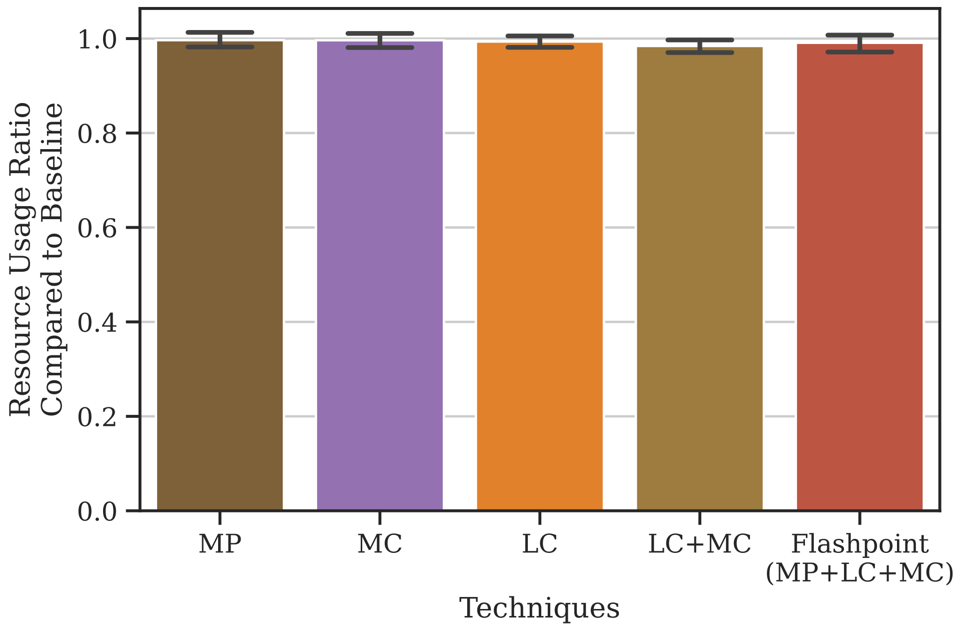


Figure 8.12: Resource usage ratio for different techniques compared to the baseline

Fig. 8.12 shows the ratio of average number of replicas per second used for each technique and combination of techniques compared to the baseline across autoscalers and workloads with a confidence interval of 95%. The metric of average number of replicas per second is calculated by measuring the time that each replica is reserved until it is evicted, taking the sum of these across replicas, and then dividing it by the duration of the workload. The results show that the amount of resources used across autoscalers are approximately the same within 5% of the baseline system for each configuration.

While the current experiments show that Flashpoint achieves better performance with the same amount of resources, we also perform a search to identify the amount of resources that the baseline systems need to achieve the same performance as Flashpoint in terms of average end-to-end request latencies. The search is performed by modifying the thresholds of the autoscaler until the compared systems are within 10% of Flashpoint's average end-to-end request latency. For fairness, we keep the average end-to-end request latencies of the compared systems above Flashpoint, which requires less resources for the compared system. For the Azure workload across autoscalers and DL models, this results in the state-of-the-art and baseline systems respectively requiring 13.92% and 30.63% more resources to match the performance of Flashpoint. Meanwhile, for the Twitter workload, these are additional resources required of 17.30% and 38.13% respectively. In the case of the T5 model for the Twitter workload, the additional resources needed for the state-of-the-art and baseline systems to match Flashpoint's performance are 18.67% and 53.28% respectively.

8.6 Evaluation Summary

As a summary of all evaluations, including those of individual techniques in the previous chapters, Flashpoint improves system performance by reducing cold start durations and end-to-end request latencies. It achieves this through various techniques such as partitioning the model to load it in parallel, co-locating download sources in the compute cluster, and minimizing network congestion. Specifically, Flashpoint improves the system's cold start durations and end-to-end request latencies in the following ways for the larger Twitter workload:

1. Flashpoint decreases DL model loading durations by 7.1× across 6 DL models shown in Chapter 4. Flashpoint does this through model partitioning, which divides the model into modular sections and loads them in parallel.

2. Flashpoint reduces download durations by up to 45.8× across the DL models with a compute co-located replica registry and locality-aware autoscaling shown in Chapter 5.

3. Flashpoint attains near-optimal scaling of cold start durations with respect to number of concurrent cold starts by reducing network congestion, achieving a 3.3× reduction of download times at a concurrency level of 4 shown in Chapter 6.

4. Altogether, Flashpoint achieves 93.51% shorter average cold start durations than the baselines and achieves a 75.42% and 66.90% reduction in average and 99th percentile end-to-end latencies across DL models and autoscalers compared to baselines.

5. Compared to the state-of-the-art, Flashpoint achieves 78.46% shorter average cold start durations than the baselines and achieves a 20.63% and 19.69% reduction in average and 99th percentile end-to-end latencies across DL models and autoscalers.

6. To match the performance of Flashpoint in the Twitter workload in terms of average end-to-end request latencies, baseline and state-of-the-art systems require 53.28% and 18.67% more resources for the T5 model across autoscalers.

7. The evaluations show that the improvements of Flashpoint are greater with larger workloads and larger DL models. Evidence also supports that growing demand and DL model sizes exceed growth in computational power, highlighting the importance of employing the techniques in Flashpoint to address future demand.

8.7 Discussion

Growth in Demand Compared to Computational Power

In this section, we discuss the findings and implications of our evaluations on the performance of locality, in comparison to the baseline and state-of-the-art.

The advantages of Flashpoint compared to the other systems are greater with increasing deep learning model size and larger demand in proportion to the compute capacity of GPUs per host machine. In recent years, there has been a significant increase in demand for large language models and growing sizes that support the advantages of Flashpoint. This evidence includes the rapid growth of users of ChatGPT from release to 100 million users in just two months [68], growing search volumes for "Large Language Models" [30]. In addition, the largest model size has observed a growth rate of $15,000\times$ in 5 years, an annual growth rate of $6.8\times$ [71]. Meanwhile, the annual growth rate in visits of ChatGPT is $56.11\times$ for the months of November 2022 to June 2023 [33]. These growth rates are comparably larger than the $1.7\times$ growth rate in GPU compute power per year in 2018 [103] and even lower with a longer-than-2-year doubling time reported in 2022 [42]. Meanwhile, the growth rate of GPUs per server also has a $1.4\times$ growth rate based on the 2-year doubling time from the release of the 8-GPU DGX-1 system in 2016 [92] and 16-GPU DGX-2 system in 2018 [97]. These differences in growth rates create a challenge for effectively handling future large-scale workloads.

To address this challenge, Flashpoint leverages compute cluster locality to overcome the network limitations of downloading the model from cloud storage and reduces cold start durations when the change in the workload exceeds the capacity of GPUs to use cached data on the corresponding host machine. By distributing the cold start of DL models across multiple machines, among other techniques, Flashpoint increases the elasticity and end-to-end request latency performance of serving DL inferences to accommodate the growing demand for computational power.

For smaller workload volatility that can be handled within a single host machine, Flashpoint's performance improvements were modest compared to other systems. However, as the workload size is increased, representing the demands of the future, Flashpoint demonstrates significant advantages. By effectively utilizing compute cluster locality, Flashpoint outperforms other systems in terms of cold start durations and end-to-end request latencies with the same amount of resources used.

In conclusion for this discussion, our study demonstrates that Flashpoint offers significant advantages over other systems when the scale of the workload exceeds the capacity of a number of GPUs per host machine. As the demand for large language models continues to rise, our system provides an effective solution for handling large-scale workloads and overcoming the limitations of individual host machines. These findings highlight the importance of considering system-level optimizations and distributed execution approaches to meet the future demands of large language models effectively.

S3 Cloud Storage and Network Bandwidths

In the experiments, an effective bandwidth of 2,203 Mbps was observed when downloading files from S3 from the Lambda Labs test bed in the same region. While other works have identified methods to parallelize downloads from S3 [67], the improvement has been shown to be up to 23%. These improvements are likely bottlenecked by other parts of network bandwidth, such as the bandwidth from the compute cluster to the cloud storage nodes, which may use public-facing Internet connections. In addition, no further optimizations on S3 cloud storage based on standard S3 download patterns in commercial systems use one S3 location [6]. Meanwhile, the effective bandwidth achieved when downloading the DL model binary file from other host machines in the Lambda Labs test bed is 7,506 Mbps. Techniques such as jumbo frames with an MTU of 9000 bytes is used. Since the network interface card has a limit of 50 Gbps, there is room to optimize host-to-host transfers. Such improvements would yield greater results for Flashpoint.

Comparison of TCP Chaining and Peer-to-peer Networks

While both TCP Chaining and peer-to-peer networks such as BitTorrent optimize network bandwidth usage within a cluster of machines, this work uses TCP chaining to simplify the controller state and tracking information. A peer-to-peer network can provide stronger fault tolerance with further improvements in download durations. For example, in TCP chaining, the last node in the chain may receive longer first-byte latencies. While other multicasting protocols can be used, in this paper, we use TCP chaining as a simple protocol that can demonstrate the impact of improving network utilization in reducing model load time.

Model Versioning

In this thesis, we identify each Deep Learning (DL) model as separate from other versions. When updating a model's layers, the model is treated as a separate replica from its original version. In this case, the data will be redistributed among the compute nodes as the requests for the new version arrives to the system.

In comparison, a DL model can be updated by only changing its weights. Updating the weights of the model is typically done for online learning systems, which are frequently retrained based on new results. When changing the weights of the DL model, the execution time is generally unchanged. This enables the reuse of the precalculated partition plans for each value of x number of requests served in Flashpoint's model partitioning since the optimal partition plans depend on only the following properties of the DL model's layers: file size, intermediate data size, and execution time.

Chapter 9

Conclusion

In this work, we have identified and characterized the GPU cold start problem in detail for Deep Learning (DL) inference serving in GPU-enabled serverless systems. We have then investigated various solutions and measured their impact on the problem. To tackle the problem, we introduced Flashpoint, which makes use of automatic model partitioning, remote memory pooling, a compute-colocated distributed registry, locality-aware autoscaling and network multicasting.

Our results show that, compared to baselines, Flashpoint achieves 93.51% shorter cold start durations across 6 DL models and 4 different autoscalers. As a result, Flashpoint achieves 75.42% and 66.90% shorter average and 99th percentile end-to-end request latencies respectively. Alternatively, to match the performance of Flashpoint, state-of-the-art and baseline systems require 18.67% and 53.28% more resources in the example of the largest DL model, T5, across autoscalers for the Twitter workload. We also gathered evidence that highlights the importance of employing the techniques in Flashpoint to address future demand. Through the advancements, Flashpoint creates the infrastructure needed to support the cost-effective deployment of large DL models with low-latency requirements.

References

[1] Aaqib, Jeremiah Chung, Jagadeesh J, Dhanasekar Karuppasamy, Harsh Bafna, Gunand Rose4U, Geeta Chauhan, Shivam Shriwas, Sadra Barikbin, Mark Saroufim, Hongbo Miao, Dhaniram Kshirsagar, and Meng Meng. Torchserve on kubernetes, 2019. URL: https://github.com/pytorch/serve/blob/master/kubernetes/README.md#torchserve-on-kubernetes.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

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

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[5] Erwan Alliaume and Benjamin Le Roux. Cold start / warm start with aws lambda, August 2018. URL: https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/.

[6] Inc. Amazon Web Services. Use your own inference code with hosting services.

[7] George Anadiotis. Sagemaker serverless inference illustrates amazon's philosophy for ml workloads, April 2022. URL: https://venturebeat.com/ai/sagemaker-serverless-inference-illustrates-amazons-philosophy-for-ml-workloads/.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, February 2009. URL: http://berkeleyclouds.blogspot.com/2009/02/above-clouds-released.html.

[9] The Kubernetes Authors. Horizontal pod autoscaling. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[10] AWS. Amazon ec2 instance types. URL: https://aws.amazon.com/ec2/instance-types/.

[11] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 126–138, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3357223.3362706.

[12] Nilesh Barla. Deploying large nlp models: Infrastructure cost optimization, Jun 2023. URL: https://neptune.ai/blog/nlp-models-infrastructure-cost-optimization.

[13] Batch inference with torchserve. URL: https://pytorch.org/serve/batch_inference_with_ts.html.

[14] Jan Bauer. Using container images to run pytorch models in aws lambda, February 2021. URL: https://aws.amazon.com/blogs/machine-learning/using-container-images-to-run-pytorch-models-in-aws-lambda/.

[15] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82, 2018. doi:10.1109/DSN.2018.00020.

[16] Rajneesh Bhardwaj, Felix Kuehling, and David Yat Sin. Fast checkpoint restore for amd gpus with criu, September 2021. URL: https://media.ccc.de/v/xdc2021-5-fast_checkpoint_restore_for_amd_gpus_with_criu.

[17] Andrew Bloomenthal. Production possibility frontier (ppf): Purpose and use in economics, August 2022. URL: https://www.investopedia.com/terms/p/productionpossibilityfrontier.asp.

[18] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3392698.

[19] Chris Caron. Uftp: Mass file distribution using multicasting, 10 2014. URL: http://nuxref.com/2014/10/01/mass-file-distribution-using-multicasting/.

[20] Eole Cervenka. All you need is one gpu: Inference benchmark for stable diffusion, October 2022. URL: https://lambdalabs.com/blog/inference-benchmark-stable-diffusion/.

[21] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 20–32, 2022. URL: https://proceedings.mlsys.org/paper/2022/file/0777d5c17d4066b82ab86dff8a46af6f-Paper.pdf.

[22] NVIDIA Corporation. Gpu-based deep learning inference: A performance and power analysis, November 2015. URL: https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf.

[23] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3419111.3421285.

[24] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 613–627, USA, 2017. USENIX Association.

[25] Abdul Dakkak, Cheng Li, Simon Garcia de Gonzalo, Jinjun Xiong, and Wen mei Hwu. 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). IEEE, jul 2019. URL: https://doi.org/10.1109%2Fcloud.2019.00067, doi:10.1109/cloud.2019.00067.

[26] Jaime Dantas, Hamzeh Khazaei, and Marin Litoiu. Bias autoscaler: Leveraging burstable instances for cost-effective autoscaling on cloud systems. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3493651.3493667.

[27] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 356–370, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3423211.3425690.

[28] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 585–599, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472883.3486982.

[29] Define a scaling policy. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-auto-scaling-add-code-define.html.

[30] Cem Dilmegani. Large language models: Complete guide in 2023, Feb 2023. URL: https://research.aimultiple.com/large-language-models/.

[31] Distributed rpc framework. URL: https://pytorch.org/docs/stable/rpc.html.

[32] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378512.

[33] Fabio Duarte. Number of chatgpt users (2023), Jul 2023. URL: https://explodingtopics.com/blog/chatgpt-users.

[34] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL: https://arxiv.org/abs/2002.08155, doi:10.48550/ARXIV.2002.08155.

[35] Peter Forret. Image filesize calculator. URL: https://toolstud.io/photo/filesize.php?imagewidth=518&imageheight=518.

[36] Eva G. Will chatgpt remain free? can it replace software engineers? what does it mean for future business opportunities and job market?, December 2022. URL: https://medium.com/swlh/3-questions-puzzled-me-about-openais-chatgpt-and-here-is-what-i-learned-1dda74b5f6db.

[37] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. Crum: Checkpoint-restart support for cuda's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 302–313, 2018. doi:10.1109/CLUSTER.2018.00047.

[38] Jim Gemmell, Todd Montgomery, T. Speakman, and J. Crowcroft. The pgm reliable multicast protocol. *Network, IEEE*, 17:16 – 22, 02 2003. doi:10.1109/MNET.2003.1174173.

[39] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[40] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 280–295, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3423211.3425683.

[41] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 779–794, 2022. URL: https://proceedings.mlsys.org/paper/2022/file/ea5d2f1c4608232e07d3aa3d998e5135-Paper.pdf.

[42] Marius Hobbhahn and Tamay Besiroglu. Trends in gpu price-performance, Jun 2022. URL: https://epochai.org/blog/trends-in-gpu-price-performance.

[43] Imagenet object localization challenge. https://www.kaggle.com/competitions/imagenet-object-localization-challenge/overview.

[44] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform, 2017. URL: https://arxiv.org/abs/1710.08460, doi:10.48550/ARXIV.1710.08460.

[45] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018. doi:10.1109/IC2E.2018.00052.

[46] Twinkle Jain and Gene Cooperman. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[47] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Amps-inf: Automatic model partitioning for serverless inference with cost efficiency. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472456.3472501.

[48] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. Improving the performance of distributed tensorflow with rdma. *Int. J. Parallel Program.*, 46(4):674–685, aug 2018. doi:10.1007/s10766-017-0520-3.

[49] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. Improving the performance of distributed tensorflow with rdma. *Int. J. Parallel Program.*, 46(4):674–685, aug 2018. doi:10.1007/s10766-017-0520-3.

[50] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[51] Simon Kassing, Ingo Müller, and Gustavo Alonso. Resource allocation in serverless query processing, 2022. arXiv:2208.09519.

[52] Xin Zhe Khooi, Cha Hwan Song, and Mun Choon Chan. Towards a framework for one-sided rdma multicast. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '21, page 129–132, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3493425.3502766.

[53] Peter L. King. Crack the code understanding safety stock and mastering its equations, July 2011. URL: https://web.mit.edu/2.810/www/files/readings/King_SafetyStock.pdf.

[54] Rao Divate Kodandarama, MD Mohan, and Shreeshrita Patnaik. Serfer: Serverless inference of machine learning models, 2019.

[55] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019. URL: http://arxiv.org/abs/1909.11942, arXiv:1909.11942.

[56] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019. URL: https://arxiv.org/abs/1910.13461, doi:10.48550/ARXIV.1910.13461.

[57] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458817.3476168.

[58] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/li-jie.

[59] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *CoRR*, abs/1903.12221, 2019. arXiv:1903.12221.

[60] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. Flashcube: Fast provisioning of serverless functions with streamlined container runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 38–45, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477113.3487273.

[61] Will Liu. The cloud waste problem: How to stop overprovisioning resources in 2022, February 2022. URL: https://cast.ai/blog/the-cloud-waste-problem-how-to-stop-overprovisioning-resources/.

[62] Load testing your auto scaling configuration. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html.

[63] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. Servermix: Tradeoffs and challenges of serverless data analytics. *CoRR*, abs/1907.11465, 2019. URL: http://arxiv.org/abs/1907.11465, arXiv:1907.11465.

[64] Joseph P. Macker, Carsten Bormann, Mark J. Handley, and Brian Adamson. NACK-Oriented Reliable Multicast (NORM) Transport Protocol. RFC 5740, November 2009. URL: https://www.rfc-editor.org/info/rfc5740, doi:10.17487/RFC5740.

[65] Kunal Mahajan and Rumit Desai. Serving distributed inference deep learning models in serverless computing. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 109–111, 2022. doi:10.1109/CLOUD55607.2022.00029.

[66] Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing*, 10(4):2834–2847, 2022. doi:10.1109/TCC.2020.3033373.

[67] Varun Nandimandalam. Parallelizing large downloads for optimal speed, Jun 2016.

[68] Nerdynav. 73 important chatgpt statistics & facts for march 2023 + infographic, March 2023. URL: https://nerdynav.com/chatgpt-statistics/.

[69] Openwhisk architecture. URL: https://openwhisk.apache.org/documentation.html#openwhisk_architecture.

[70] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522716.

[71] Dhawal Patel, Gili Nachum, and Madison Van Horn. Train and deploy large language models on amazon sagemaker, Nov 2022. URL: https://d1.awsstatic.com/events/Summits/reinvent2022/AIM405_Train-and-deploy-large-language-models-on-Amazon-SageMaker.pdf.

[72] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[73] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL: http://jmlr.org/papers/v21/20-074.html.

[74] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing plausible infrastructure configurations for evaluating edge computing systems. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020. URL: https://www.usenix.org/conference/hotedge20/presentation/rausch.

[75] Thomas Rausch and Philipp Raith. faas-sim: A trace-driven function-as-a-service simulator. URL: https://github.com/edgerun/faas-sim.

[76] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. Fa2: Fast, accurate autoscaling for serving deep learning inference with sla guarantees. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 146–159, 2022. doi:10.1109/RTAS54340.2022.00020.

[77] Reaction time test. URL: https://humanbenchmark.com/tests/reactiontime.

[78] Resnet50. URL: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html.

[79] Mike Roberts. Analyzing cold start latency of aws lambda, June 2020. URL: https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda.

[80] Mike Roberts. Cold starts in aws lambda, January 2021. URL: https://mikhail.io/serverless/coldstarts/aws/.

[81] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (ATC)*, pages 397–411. USENIX Association, July 2021.

[82] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507750.

[83] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[84] Stefan Scherfke, Ontje Lünsdorf, Peter Grayson, Eric LaFevers, Thomas Pinckney, Cristian Klein, Sundar Vaidya, Larissa Reis, Sean Reed, Zhe Liu, Thomas Deitrich, Brian Merrell, Jacob Söndergaard, Robert Kirchgessner, and Matthew Grogan. Simpy. URL: https://gitlab.com/team-simpy/simpy/-/tree/master.

[85] Serverless inference. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html#serverless-endpoints-how-it-works-cold-starts.

[86] Services, load balancing, and networking. URL: https://kubernetes.io/docs/concepts/services-networking/.

[87] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[88] Sharding container pool balancer. URL: https://github.com/apache/openwhisk/blob/master/core/controller/src/main/scala/org/apache/openwhisk/core/loadBalancer/ShardingContainerPoolBalancer.scala.

[89] Mannat Singh, Laura Gustafson, Aaron Adcock, Vinicius de Freitas Reis, Bugra Gedik, Raj Prateek Kosaraju, Dhruv Mahajan, Ross Girshick, Piotr Dollár, and Laurens van der Maaten. Revisiting Weakly Supervised Pre-Training of Visual Perception Models. In *CVPR*, 2022.

[90] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472883.3486981.

[91] Jeff Smith. Model serving in pytorch, May 2019. URL: https://pytorch.org/blog/model-serving-in-pyorch/.

[92] Ryan Smith and Dr. Ian Cutress. Nvidia unveils the dgx-1 hpc server: 8 teslas, 3u, q2 2016, Apr 2016. URL: https://www.anandtech.com/show/10229/nvidia-announces-dgx1-server.

[93] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. ENSURE: efficient scheduling and autonomous resource management in serverless environments. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020*, pages 1–10. IEEE, 2020. doi:10.1109/ACSOS49614.2020.00020.

[94] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. Transparent checkpoint and restart technology for cuda applications, April 2016. URL: https://on-demand.gputechconf.com/gtc/2016/presentation/s6429-akira-nukada-transparen-checkpoint-restart-technology-cuda-applications.pdf.

[95] Xuehai Tang, Peng Wang, Qiuyang Liu, Wang Wang, and Jizhong Han. Nanily: A qos-aware scheduling for dnn inference workload in clouds. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 2395–2402, 2019. doi:10.1109/HPCC/SmartCity/DSS.2019.00334.

[96] Twitter streaming traces, 2018. URL: https://archive.org/details/archiveteam-twitter-stream-2018-04.

[97] Mark Tyson. Nvidia launches the dgx-2 with two petaflops of power, Mar 2018. URL: https://hexus.net/tech/news/systems/116672-nvidia-launches-dgx-2-two-petaflops-power/.

[98] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[99] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual*

Technical Conference (USENIX ATC 21), pages 443–457. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/wang-ao.

[100] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[101] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast rdma-codesigned remote fork for serverless computing, 2022. URL: https://arxiv.org/abs/2203.10225, doi:10.48550/ARXIV.2203.10225.

[102] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/weng.

[103] Alex Woodie. Nvidia riding high as gpu workloads and capabilities soar, Mar 2018. URL: https://www.hpcwire.com/2018/03/27/nvidia-riding-high-as-gpu-workloads-and-capabilities-soar/.

[104] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302424.3303975.

[105] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/3007203.3007217.

[106] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*,

ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507709.

[107] Ging-Fung Yeung. *Proactive Interference-aware Resource Management in Deep Learning Training Cluster*. PhD thesis, Lancaster University, 2022. doi:10.17635/lancaster/thesis/1673.

[108] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40, 2021. doi:10.1109/ACSOS52086.2021.00023.

[109] Hanfei Yu, Hao Wang, Jian Li, and Seung-Jong Park. Harvesting idle resources in serverless computing via reinforcement learning. *CoRR*, abs/2108.12717, 2021. URL: https://arxiv.org/abs/2108.12717, arXiv:2108.12717.

[110] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148, 2021. doi:10.1109/ICDCS51616.2021.00022.

[111] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

[112] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

[113] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinistore: Elastic serverless cloud storage, 2023. arXiv:2209.01496.

[114] Qi Zhang, Quanyan Zhu, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L. Hellerstein. Dynamic service placement in geographically distributed clouds. *IEEE Journal on Selected Areas in Communications (JSAC)*, 31(12):762–772, Dec. 2013.

[115] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3357223.3362723.

[116] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. Dialogpt: Large-scale generative pre-training for conversational response generation, 2019. URL: https://arxiv.org/abs/1911.00536, doi:10.48550/ARXIV.1911.00536.

[117] Ming Zhao, Kritshekhar Jha, and Sungho Hong. Gpu-enabled function-as-a-service for machine learning inference, 2023. arXiv:2303.05601.