PISeL: Pipelining DNN Inference for Serverless Computing

Masoud Rahimi Jafari[*]
University of Connecticut
Storrs, Connecticut, USA

Jianchang Su[*]
University of Connecticut
Storrs, Connecticut, USA

Yifan Zhang
University of Connecticut
Storrs, Connecticut, USA

Oliver Wang
University of Chicago
Chicago, Illinois, USA

Wei Zhang[†]
University of Connecticut
Storrs, Connecticut, USA
wei.13.zhang@uconn.edu

Abstract

Serverless computing offers resource efficiency, cost efficiency, and a "pay-as-you-go" pricing model, which makes it highly attractive to both users and cloud providers. However, serverless computing faces serious cold start problem, especially for deep neural network (DNN) inference, which requires low latency. Existing cold start optimization focuses only on quick container start and fast runtime and library loading. However, DNN application bootstrap (DNN framework load and start, model initialization, model download, deserialization and copy) is the leading factor during the overall cold start time. As the model size grows, the application-level bootstrap becomes more severe.

We present PISeL, a generic and fast application-level cold-start optimization mechanism for DNN inference. We propose a layer-grouping mechanism and policy to pipeline model download, model deserialization and copy and request execution. The grouping policy strikes a balance that minimizes both pipeline bubble risk and synchronization overhead. The pipelining process is transparent to a variety of DNN jobs and is implemented with the hook point in a lightweight manner. PISeL not only greatly reduces the cold start time, but also the peek memory usage which can easily incur OOM (out of memory) problem. Our experiments show that PISeL accelerates cold start time with all experimented system configurations and DNN models. PISeL can speed up cold start times by 37% and 63% using PyTorch framework executed on CPU and GPU and also 29% and 33% using TensorFlow framework executed on CPU and GPU. Furthermore, PISeL reduces maximum memory usage by up to 59% and 30% using PyTorch and TensorFlow frameworks.

CCS Concepts

• **Computing methodologies → Machine learning**; • **Computer systems organization → Cloud computing**.

[*]Both authors contributed equally to this work.
[†]Corresponding author.

Keywords

DNN Inference; Cold-Start Optimization; Model Partitioning

ACM Reference Format:
Masoud Rahimi Jafari, Jianchang Su, Yifan Zhang, Oliver Wang, and Wei Zhang. 2024. *PISeL*: Pipelining DNN Inference for Serverless Computing. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24), October 21–25, 2024, Boise, ID, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3627673.3679824

1 Introduction

Deep neural networks (DNNs) have revolutionized the field of artificial intelligence and have become increasingly popular for a wide range of applications, including image classification, speech recognition, and natural language processing. However, as the DNN models become complex and large, it raises the challenges to properly allocate the required resources for the users and manage the resource at scale to accommodate the complex and varied workloads [12, 28, 37].

Serverless computing offers resource efficiency, cost efficiency, and a "pay-as-you-go" pricing model [15]. A serverless platform runs the service inside the containers and automatically handles resource management. The developers simply upload their code and the platform executes it on their behalf as needed at any scale. Developers do not need to worry about resource provision or server operation, and they pay only for the computing resources used when their code is invoked. This auto-scaling and cost-effective price model makes serverless computing an attractive tool to accommodate the surge in DNN workloads [38].

Serverless computing removes the complex burden of resource management for the users [30]. However, it faces the problem of lengthy execution environment setup. When a service is required or there are new incoming requests, the required running environment has to be setup and provided quickly. It involves container creation and start, runtime and library setup and loading and application bootstrap, which calls cold start time. Only if the execution environment is ready, a request can start to be served. The expensive cold start time adds the substantial latency to the users' request. There are many works focusing on the optimization of cold start time, such as fast container creation and start, quick runtime and library setup and loading. Unfortunately, the application environment setup and bootstrap is overlooked. However, for DNN inference workloads the application bootstrap becomes the leading factor during the overall cold start time, shown in section 2. After a DNN model is initialized, the large model parameters need to be pulled from the remote storage to the compute node. Then the downloaded data is required to be deserialized, and then an expensive memory copy is followed to copy the datas into the device. To practically deploy the DNN inference workloads in the serverless computing platform, we have to optimize the expensive DNN application bootstrap time.

On the other hand, during the DNN application bootstrap, the memory copy for the model parameters loading incurs the significant memory consumption increase. The instant memory peak usage in DNN workloads can easily lead to out of memory (OOM) problem. Although setting up a large memory needs can mitigate this problem, it indeed brings more cost to the users. Furthermore, the public providers usually adds a limitation for the maximum memory a container can take. Eg, AWS Lambda sets up a maximum memory limitation for each container with 10GB. Due to the maximum memory limitation and cost efficiency to the users, reducing the peak memory usage for DNN model at any stage is required in serverless computing environment.

Furthermore, there are a variety of DNN frameworks with the different versions (such as PyTorch, Tensorflow, MXNet et.al) and DNN jobs from the different areas (such as NLP, CV...). It requires a general solution to optimization the cold start primarily incurred by the DNN application bootstrap and peak memory consumption caused by the memory copy during the loading stage. The proposed solution should not require any change or update from the DNN jobs and models and fully be be transparent to them. It should also provide well compatibility with the different DNN frameworks.

To this end, we propose PISeL, a generic and efficient DNN application bootstrap in serverless computing environment. PISeL leverages a key observation that DNN model is usually composed by a number of layers and a request follows a layer-by-layer execution pattern. As such, there is no need to wait for the entire model to be downloaded from the remote storage, then start to do the data deserialization and loading into the device, and finally kick off the request serving. Based on this observation, we design a pipelined model mechanism, which can pipeline the model download from the remote storage, model deserialization and loading and the request execution. However, the stages across the download, deserialization and loading and execution require the synchronization. Only when the previous stage is completed can the next stage start performing its work.

The first challenge is to how to divide the model into groups and overlap the latency across the stages. The naive pieplining strategy is to perform on a layer-by-layer granularity. However, it can incur high data transmission requests and synchronization overhead. To reduce those overhead, PISeL pipeline divides a model layers into groups and partitions, each of which contains several model layers. A bad partition decision could incur serious pipeline bubble, which incurs the unnecessary waiting time across the stages and then dramatically impact the pipeline efficiency [23]. To strike the pipeline bubble and synchronization overhead, we design an optimal model-aware grouping algorithm to find the best grouping strategy for a given model. With the pipelined model, it not only shorten the DNN bootstrap time by overlapping the latency across the stages, it can accordingly reduce the peak memory usage during the loading time. The traditional way needs to download everything and then start the memory copy for loading, which requires double the size of model parameters for memory consumption. In contrast, the memory copy in the partitioned model only involves the parameters of a few layers within a group.

The second challenge is to make the mechanism generic and accommodate a variety of DNN models and DNN platforms with different versions. We provide plugins for the popular DNN platforms including Pytorch, Tensorflow and MXNet to easily implement the pipelining DNN bootstrap with function hook points instead of any model change or deep framework update. To do so, the mechanism can be easily deployed and migrated from one version to another version for the different DNN frameworks.

As the model size grows, PISeL performs more effective. With small model size, the system benefit is limited as expected while not adding extra latency and overhead. In the evaluation section, we run the small, medium and big DNN model sizes to show the system behavior. Because of the limited number of released models from MXNet and the difficulty of model conversion from the other platfroms to MXNet platform, we only run the models on the Pytorch and Tensorflow frameworks. Our evaluation shows that PISeL can reduce response time in cold start scenarios up to 35% on CPU and 30% on GPU. Accordingly PISeL can significantly decrease the peak memory usage. Eg, the peak memory usage of the GPT-2 XL model falls down from 16GB to 9GB with PISeL enabled.

In summary, we make the following contributions:

- We observe the serious impact of DNN application bootstrap to cold start time in serverless computing platform and the serve peak memory consumption caused by DNN model copy during the loading.
- We design a generic and fast application-level cold-start optimization mechanism, PISeL, that can pipeline model download, model deserialization and loading, and computation in a light-weight and transparent manner.
- We propose a group-partition algorithm to intelligently divide layers into a group to pipeline the stages. It can strike the balance between the pipeline bubble risk and synchronization overhead.
- We prototype PISeL on the popular PyTorch and Tensorflow platforms, and then evaluate its effectiveness of improvement in cold start time and peak memory usage.

2 Background and Motivation

This section provides an overview of serverless computing and its application to DNN inference, highlighting the key challenges and limitations that motivate our work. We draw upon relevant prior literature to contextualize our contributions and situate our research within the broader field.

Serverless Computing. Serverless computing, also known as Function as a Service (FaaS), has emerged as a popular paradigm for deploying applications in the cloud [15]. In contrast to traditional cloud computing models, serverless platforms abstract away the underlying infrastructure, allowing developers to focus on writing and deploying stateless functions without the need for server management. Serverless platforms automatically handle resource allocation, scaling, and billing, providing a highly elastic and cost-effective environment for executing event-driven workloads.

The fine-grained, pay-per-use pricing model of serverless computing has made it an attractive option for a wide range of applications, including data processing, web services, and machine learning inference. However, the serverless execution model also introduces new challenges and limitations, particularly in terms of performance predictability and resource efficiency. One of the most significant issues is the cold start problem, which refers to the latency incurred when a function is invoked after a period of inactivity, requiring the platform to provision and initialize a new execution environment [1, 2].

Cold-start in DNN Serving. The application of serverless computing to DNN inference workloads has gained significant attention due to the growing demand for scalable and cost-effective inference serving [40]. However, the cold start problem poses a major challenge for DNN inference in serverless environments, as initializing and loading large DNN models can cause substantial latency.

As shown in Figure 1, serving a DNN inference request in a serverless platform involves several time-consuming steps, including downloading model parameters from remote storage (e.g., S3, Blob storage), loading the parameters into the serverless function's memory, and executing the inference computation on the input data. Each step significantly contributes to the overall inference latency, especially for larger and more complex DNN models.

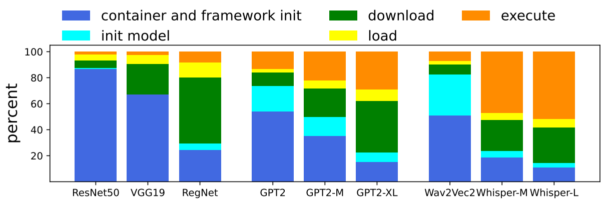
Previous work has explored various methods to reduce cold start latency in serverless environments by optimizing container creation, startup, runtime, and libraries [1, 2, 5, 8, 19, 31, 41]. While promising for general serverless workloads, these techniques may not fully address the unique requirements of DNN inference, which involve large models and strict latency constraints, making application-level bootstrap a leading time cost in the overall cold-start time.

Recent studies have specifically targeted the performance and resource optimization of DNN workloads in serverless environments [11, 18, 22]. Although these studies have made valuable contributions, they do not fully address the challenges of cold start latency and resource efficiency for DNN inference in serverless environments. Our work aims to bridge this gap by proposing a novel, framework-agnostic approach that optimizes the end-to-end performance of DNN inference workloads through pipelined execution and efficient resource management.

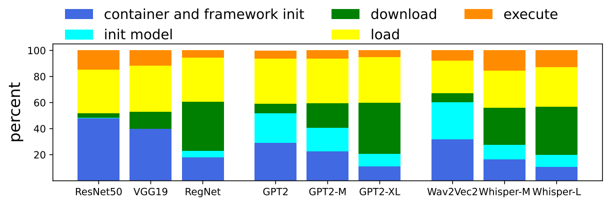
Memory Footprint. Memory consumption is a critical factor in serverless computing [16, 18, 39], as it directly impacts both the cost and performance of applications. Serverless platforms typically charge users based on the amount of memory allocated to their functions, even if the actual memory usage is lower. This pricing model incentivizes developers to carefully optimize the memory footprint of their serverless functions to minimize costs while ensuring adequate performance.

DNN inference workloads are particularly memory-intensive, as they require loading large model parameters into memory during the inference process. As shown in Figure 2, popular DNN frameworks such as PyTorch and TensorFlow can exhibit substantial memory overheads during the model loading phase, often far exceeding the actual runtime requirements of the inference task. This inefficiency stems from the creation of multiple copies of model parameters during the loading process, leading to unnecessary memory consumption and increased costs.

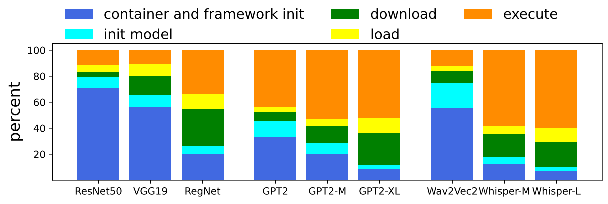
Prior work has investigated various techniques for optimizing memory usage in serverless environments, aiming to reduce costs and improve resource efficiency. However, these solutions do not specifically address the memory inefficiencies of DNN frameworks,



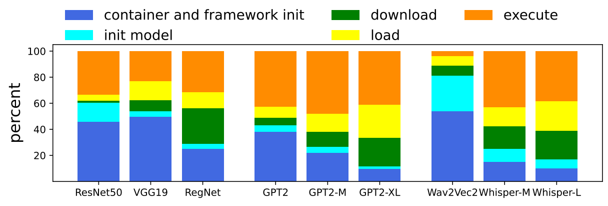
(a) PyTorch CPU



(b) PyTorch GPU



(c) TensorFlow CPU



(d) TensorFlow GPU

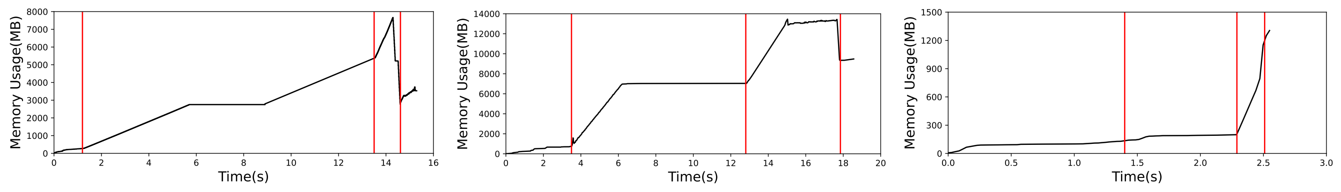
Figure 1: Tasks time in a cold-start scenario

which are a major contributor to the high memory footprint of serverless inference workloads.

Transparency To Frameworks and Platforms. Optimizing DNN inference workloads in serverless environments is challenging due to the lack of transparency and portability across different frameworks and platforms. Existing techniques often rely on framework-specific modifications or deep integration with the serverless infrastructure, limiting their adoptability and generalizability.

Our work addresses this by proposing a framework-agnostic approach that can be easily integrated with popular DNN frameworks like TensorFlow, PyTorch, and MXNet, without extensive modifications to the application code or serverless platform. Using dynamic library interposition and function-level hooks, our solution enables transparent optimization of DNN inference workloads across various serverless environments.

Recognizing the importance of transparency and interoperability in serverless computing, prior work has aimed to establish common standards and abstractions. Our approach aligns with



(a) PyTorch **(b) TensorFlow** **(c) MXNet**

Figure 2: Memory usage during a cold-start

these efforts, providing a transparent, framework-agnostic solution for optimizing DNN inference workloads in serverless environments. This contributes to building a more open, interoperable, and performant serverless ecosystem by eliminating the need for framework-specific modifications and enabling seamless integration with existing serverless platforms.

3 System Design

PISeL is designed to address the challenges of cold start latency and high memory usage in serverless environments for DNN services. The design goals of PISeL are as follows:

- **G1: Minimize the cold start time.** Optimize the DNN application bootstrap time due to the leading factor for the cold start time in the serverless computing platform.
- **G2: Minimize the peak memory usage.** Reduce the memory copy consumption during model loading and substantially minimize the peak memory usage.
- **G3: Minimize the introduced overhead.** Provide a light-weight sychronization mechanism during model loading and completely remove the synchronization after fully load.
- **G4: Provide the transparency to the DNN jobs.** Design the optimization mechanism which is transparent to a variety of DNN jobs, which does not incur any code change or update for DNN jobs.
- **G5: Maintain the compatibility to DNN frameworks.** Provide the compatibility to the different versions of DNN frameworks by avoiding any deep change.

To achieve **G1** and **G2**, PISeL partitions the DNN model into smaller parts, grouping consecutive layers while minimizing overhead. To achieve **G3**, PISeL removes all the hooks and wrappers used for partitioning and parallelization once the model is fully loaded. This ensures zero overhead during steady-state operation, allowing the framework to serve requests efficiently. PISeL maintains transparency and compatibility (**G4** and **G5**) by using hooks and wrappers that can be easily adapted to different DNN frameworks. This enables seamless integration with popular libraries such as PyTorch, TensorFlow, and MXNet.

3.1 System Overview

Figure 3 illustrates the overall architecture of PISeL, showing the interaction between the various components. The system design allows for efficient model partitioning and parallel downloading, addressing **G1** by minimizing cold start times and **G2** by reducing peak memory usage during loading. It achieves **G3** by utilizing a lightweight synchronization mechanism during model loading and removing it once loading is complete, ensuring minimal overhead. The architecture also supports **G4** and **G5**, maintaining transparency and compatibility with various DNN frameworks.

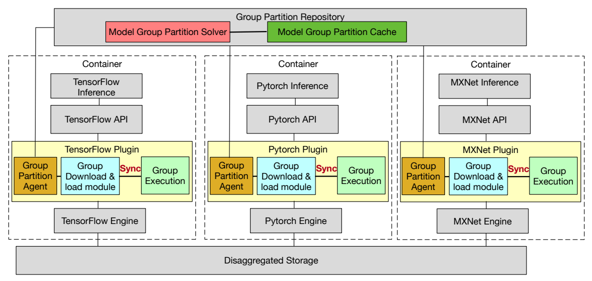


Figure 3: PISeL Architecture

Model Partitioning. The first stage in PISeL involves splitting the DNN model into partitions, where each partition consists of one or more layers. Partitioning the model enables parallel downloading, loading, and execution of the layers, which significantly reduces the cold start time and peak memory usage.

When designing the partitioning strategy, we considered two key factors: the granularity of the partitions and the overhead associated with each partition. Splitting a single layer into multiple partitions does not provide any benefits in terms of memory usage or response time during cold starts. This is because, during the loading phase, the entire layer must be passed to the DNN framework regardless of how it is partitioned. Therefore, our design ensures that each layer remains intact within a partition. Additionally, to achieve optimal performance through parallel execution and loading, our partitioning strategy groups only consecutive layers within a partition. This allows the layers to be loaded in the correct order, enabling the DNN framework to start executing the layers as soon as their dependencies are resolved.

The challenge lies in determining the optimal grouping of consecutive layers into partitions. Two main considerations guide this decision. First, creating a new partition introduces an overhead, which is equal to the request latency (RL) associated with initializing the partition. Second, in the absence of overhead, the ideal scenario for minimizing peak memory usage and cold start response time would be to place each layer in its own partition. However, this approach is not practical due to the overhead associated with managing a large number of partitions. To address this challenge, we introduce a lower bound on the size of each partition. The partition overhead can be avoided if the size of the previous partition exceeds the product of RL and the connection bandwidth(BW).

Model Partitioning Solver. The model partitioning objective is to minimize the response time in cold start scenarios while reducing the maximum memory consumption during model loading. We formulated the problem as below:

Let $D[i]$, $L[i]$, and $E[i]$ denote the time taken for the i-th layer to be downloaded, loaded, and execute the request, respectively. Additionally, let RL represent the time remote storage takes to start transfering a file to the application. The tasks can be defined as $D[0:n]$, $L[0:n]$, and $E[0:n]$, with dependencies $E[i] \rightarrow L[i] \rightarrow D[i]$. Due to CPU and I/O constraints, only one L or E task and one D task can be executed in a thread at any given moment. For each layer, PISeL considers two options:

(1) Merging the layer with the next partition
(2) Splitting the layer from the next partition and starting a new partition

Merging reduces the number of download requests to remote storage cluster but may increase the maximum memory usage during loading and miss the opportunity to perform $L[i]$ and $E[i]$ while the next partition is being downloaded.

The ideal memory management method is sequential layer loading, minimizing peak memory use. However, due to considerable transmission and synchronization overhead, PISeL partitions the model into small-sized partitions, balancing memory efficiency with reduced loading delays.

PISeL proposal a model partitioning algorithm for splitting the DNN model into optimal partitions when it receives a partitioning order from the Partition Agent. The algorithm uses the RL and BW parameters provided in the partitioning order to determine the optimal partition size. It downloads the model from the remote storage (e.g., AWS S3, Google Cloud Storage, Azure Blob Storage, etc.), groups layers into partitions such that each partition size exceeds $RL \times BW$, and uploads the partitions to the Model Partitioning Cache as shown in Algorithm 1.

Algorithm 1 Greedy-based Model Partitioning Algorithm

```
Input: model: a sequence of layers ${L_1, L_2, ..., L_n}$;
       RL: request latency; BW: bandwidth of the network connection
Output: Optimal partitions P of the model
1:  function ModelPartitioning(model, RL, BW)
2:      $P, p_i \leftarrow \emptyset, \emptyset$
3:      $b_{min} \leftarrow RL \cdot BW$
4:      for i, layer ∈ enumerate(model) do
5:          $p_i \leftarrow p_i \cup \{layer\}$
6:          if $|p_i| \geq b_{min}$ then
7:              $P \leftarrow P \cup \{p_i\}$
8:              upload_to_cache($p_i$)
9:              $p_i \leftarrow \emptyset$
10:     if $p_i \neq \emptyset$ then
11:         $P \leftarrow P \cup \{p_i\}$
12:         upload_to_cache($p_i$)
13:     return P
```

The algorithm iterates through the model layers, grouping them into partitions until the size of a partition reaches the minimum bound $b_{\min}$, calculated as $RL \times BW$. Partitions are uploaded to the Model Partitioning Cache for parallel downloading and processing.

The Partition Downloader downloads partitions in parallel, initiating the next download when the remaining size of the current partition equals the overlay size ($RL \times BW$), maximizing network bandwidth utilization. The Partition Loader and Executioner load and execute layers within each partition as they become available, allowing inference requests to be processed without waiting for the entire model to load. Condition locks synchronize the loading and execution processes.

We next proof the correctness of the partitioning algorithm.

Theorem 1. *Algorithm 1 finds the optimal partitioning strategy that minimizes the number of partitions while ensuring that each partition size is at least b_{min}.*

Proof. **Base case:** For $n = 1$, if the size of the single layer L_1 is at least $b_{\min}$, it forms its own partition, which is clearly optimal since no better partitioning exists. If $|L_1| < b_{\min}$, it still forms a partition by itself, as no partitioning can increase its size, fulfilling the minimum condition by definition of infeasibility to combine with other layers.

Inductive hypothesis: Assume that for any sequence of layers $\{L_1, L_2, \ldots, L_k\}$ of length k, where $k \geq 1$, the algorithm finds the optimal grouping strategy.

Inductive step: Consider a sequence $\{L_1, L_2, \ldots, L_{k+1}\}$. Let P be the partitioning strategy produced by the algorithm for this sequence. Suppose P^* is any other optimal partitioning strategy for the same sequence.

Let p_ℓ in P^* be the last partition. If p_ℓ contains L_{k+1} and other layers starting from some L_i, then by optimality of $P^{1:k}$ for $\{L_1, L_2, \ldots, L_{i-1}\}$ (inductive hypothesis), and since $|p_\ell| \geq b_{\min}$, it aligns with the formation in P. If p_ℓ equals $\{L_{k+1}\}$ and $|L_{k+1}| \geq b_{\min}$, L_{k+1} forms a partition in P optimally. If $|L_{k+1}| < b_{\min}$, then in P, L_{k+1} would join the last partition of $P_{1:k}$ unless adding it exceeds a threshold, which should be checked by the algorithm.

Thus, in each case, P adheres to the optimal strategy, completing the induction. □

Partitioning Cache. The Model Partitioning Cache stores pre-computed model partitions and is designed to efficiently handle simultaneous requests from multiple functions, using in-memory and distributed caching techniques for fast access. When a serverless function is triggered, it first queries this cache. If the partitions are available, the function can immediately proceed with downloading and loading them, thereby reducing the latency. This avoids the latency typically associated with on-the-fly model partitioning. If the partitions are not available, the cache triggers the Partition Agent to start the partitioning process. This process involves the Model Partitioning Solver, which determines the optimal way to partition the model based on current system configurations and model requirements. These newly created partitions are then stored back in the cache for future access by other functions.

Partition Agent. When a new function is initialized, the partition agent searches for pre-stored partitions in the Model Partition Cache. If it doesn't find any partitions in the cache, it performs an action. First, it calculates the RL and BW parameters, then sends a partitioning order to the partition solver with those two parameters. RL can be calculated by downloading several tiny objects stored in remote storage, each less than a kilobyte in size, and calculating the average download time. We calculate RL and BW in functions instead of in the partition solver because the partition solver has different resources and limitations compared to functions. If the partition solver were to measure these parameters, the results could be unrealistic. After this process, partitions are stored in the cache and can be used by other functions in the future. Finally, the partition agent passes $RL \times BW$ to the partition downloader, at which point its responsibility ends.

Partition Downloader. The Partition Downloader is responsible for downloading the model partitions from the Model Partitioning Cache. It receives the overlay size from the Partition Agent and initiates a connection to the cache. To hide the overhead of initiating partition downloads, the downloader sends the download request for the next partition when the remaining size of the current downloading partition equals the overlay size. This technique effectively hides the download initiation overhead and maximizes the utilization of the available network bandwidth. Once a partition is fully downloaded, the downloader saves it to a file and notifies the Partition Loader to load the partition.

Partition Loader. The Partition Loader is responsible for loading model partitions into memory. It works closely with the Partition Downloader, which retrieves partitions from the Model Partitioning Cache. As each partition is downloaded, the Partition Loader deserializes the data and loads layer parameters into memory. It also manages condition locks for each layer, ensuring the Partition Executioner can execute a layer only once its parameters are fully loaded.

Parallelize Model Loading And Execution To further reduce the cold start time, PISeL parallelizes model loading and request execution. This is possible because layers can execute requests as soon as their weights are loaded, and the execution process proceeds layer by layer. By leveraging this characteristic, we can start executing requests for the loaded layers while the remaining layers are still being loaded.

However, there are three challenges that must be addressed to enable efficient parallelization:

- A request can only be executed in a layer after all of the weights in that layer are fully loaded, and they should wait if the weights are not yet available.
- The overhead introduced by the parallelization mechanism must be minimal for each layer. Even a small overhead per layer can accumulate and significantly increase the response time, considering the large number of layers in complex DNN models.
- Different types of layers have varying numbers of parameters (e.g., weights, biases), and some layers may not have any parameters at all. The parallelization mechanism must take these differences into account.

To overcome these challenges, PISeL employs a condition locking mechanism before executing layers that have at least one parameter. When a model is initialized, all the locks are closed. As the layer parameters are loaded, the corresponding locks are opened, indicating that the layers are ready for execution. To minimize the overhead, PISeL does not apply locks to layers that do not have any parameters. This optimization reduces the number of lock operations and improves efficiency. Furthermore, to handle the different types of layers and their varying parameter configurations, PISeL dynamically identifies all the persistent parameters for each layer. The lock condition for a layer is updated only when all of its parameters are completely loaded. This ensures that a layer is executed only when it has all the necessary parameters available. By using this condition locking mechanism, PISeL effectively synchronizes the model loading and request execution processes. Layers with loaded weights can start executing requests immediately, while layers with pending weights wait until they are fully loaded. This parallelization significantly reduces the cold start time, as the execution of layers can begin much earlier than in the traditional sequential approach.

The Partition Loader deserializes the downloaded partitions and loads the layer parameters consecutively. As each layer's parameters are fully loaded, the corresponding condition lock is released, allowing the Partition Executioner to process requests for that layer. This synchronization between the Partition Loader and Partition Executioner ensures efficient parallel processing while maintaining the correct execution order.

Transparency and Compatibility. A key aspect of PISeL's design is its transparency to DNN jobs and serverless platforms, and compatibility to DNN frameworks. PISeL requires minimal code changes to integrate with existing systems, making it highly adaptable and easy to adopt. The use of hooks and wrappers allows PISeL to be easily incorporated into various DNN frameworks, such as PyTorch, TensorFlow, and MXNet. This transparency and compatibility enable the wider community to benefit from PISeL's optimizations without the need for extensive modifications to their existing codebase. By avoiding deep changes to the frameworks, PISeL maintains compatibility with different versions of the DNN frameworks, ensuring a smooth integration process. Moreover, PISeL's design is platform-agnostic, meaning it can be easily deployed on various serverless computing platforms without requiring any platform-specific modifications. This flexibility allows the users to leverage PISeL's benefits across the different serverless providers, further enhancing its usability and adoption potential.

4 Implementation

we have implemented a system prototype for PISeL eith 500 lines of code in python, and we have integrated that with PyTorch, Tensorflow and MXnet frameworks. For each framework there are some hooks and wrappers to do the synchronisation between downloading, loading and execution threads and also there is a fuction to calculation partitioning optimization problem. Also there our mechanism to overlay partitions downloading to remove request latency is implemented to a function.

5 Evaluation

In this section, we first demonstrate the response time of PISeL in the cold start scenario and then show the peak memory usage. We next compare the performance between the proposed layer-grouping pipelining and the naive layer-by-layer pipelining. Finally we also evaluate the overhead of the model partition algorithm in PISeL. Due to the limited number of released models from MXNet and the difficulty of model conversion from the other platfroms to MXNet platform, we only evaluate PISeL on the two most popular

DNN frameworks: Pytorch and Tensorflow on both CPU and GPU platforms. To make it clear, our objective is not to compare these frameworks, but to show the improvements contributed by PISeL on each platform.

Setup: We evaluate PISeL on CPU and GPU platforms. The CPU setup includes a server with a 24-core AMD 7402P CPU at 2.80GHz, 128GB ECC Memory (8x16GB 3200MT/s RDIMMs), and a 1.6TB NVMe SSD (PCIe v4.0). The GPU setup includes a server with two Intel Xeon E5-2667 8-core CPUs at 3.20GHz, 128GB ECC Memory, two 960GB 6G SATA SSDs, and a NVIDIA 16GB Tesla V100 SMX2 GPU. Both setups connect to a multi-node, multi-storage MinIO object storage with 10Gb/s throughput.

Workloads: DNN models used include ResNet50 [35], VGG19 [33], RegNet [29], GPT2 [27], LaBSE [9], GPT2-XL [27], Wav2Vec2 [3], Whisper-M [26], and Whisper-L [26]. We use standard configurations for each model.

Metrics: We measure latency and peak memory usage, reporting the average of 30 runs. Improvement is calculated by dividing the response time or peak memory usage with PISeL disabled by the same metric with PISeL enabled.

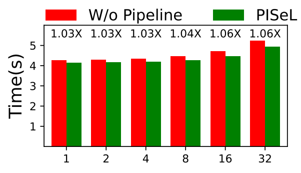
Baselines: Our baselines are "No optimization" and "layer-by-layer" pipeline. "No optimization" performs stages sequentially (download, deserialization, loading, execution), referred to as "without pipeline." "Layer-by-layer" downloads model layer by layer, with deserialization and loading following the same pattern.

5.1 Latency in Cold-Start

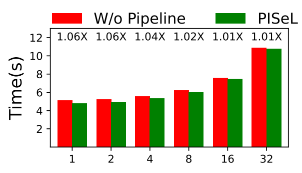
One of the key improvements of PISeL is to reduce response time in cold-start scenario. We vary the batch size from 1 to 32 to evaluate the end-to-end request latency of small, medium and large model size on both CPU and GPU platforms. We tested the improvement of PISeL on both TensorFlow and Pytorch frameworks.

Figure 4 and Figure 5 show the latency and speedup for small, medium, and large models running on the TensorFlow platform in CPU and GPU setups. We observed: 1) As the model size grows, the speedup increases because the DNN model bootstrap time takes an increasing percentage of the overall cold start time. This trend is shown in Figure 1 of § 2. 2) As the batch size increases, the speedup first increases and then decreases. This is caused by the increasing execution latency as the batch size increases. Once the execution latency dominates the overall cold start time, the stage latency across downloading, deserialization & loading, and execution cannot be well overlapped in any partition pattern, impacting pipeline efficiency. 3) For small models, due to the small percentage of DNN model bootstrap in the overall cold start, PISeL does not add significant benefits or overhead, showing similar performance to the baseline without using the pipeline. 4) PISeL shows performance gains on both CPU and GPU platforms. However, the speedup of TensorFlow on the GPU in Figure 5 is generally lower than that of PyTorch on the GPU shown in Figure 7. This is because the models on TensorFlow are not well optimized, resulting in suboptimal performance. Yet, PISeL can still achieve up to 1.32× speedup on large models like Regnet and GPT2-XL.

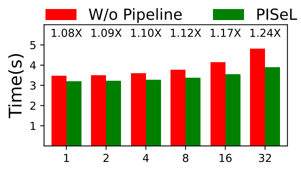
We have the similar observations while running the experiments on Pytorch platform with both CPU and GPU setup, shown in Figure 6 and Figure 7. On CPU, we observe speed-ups of up to



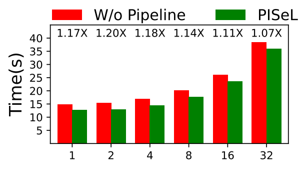
(a) ResNet101(small)



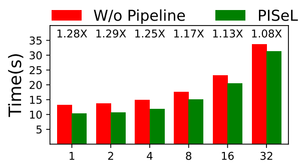
(b) GPT2(small)



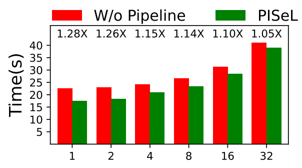
(c) VGG19(medium)



(d) LaBSE(medium)

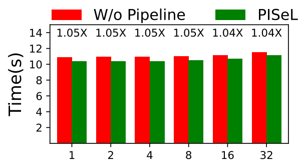


(e) RegNet(large)

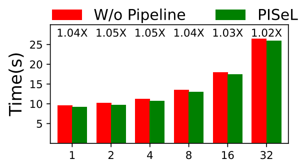


(f) GPT2-XL(large)

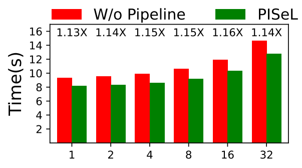
Figure 4: TensorFlow CPU



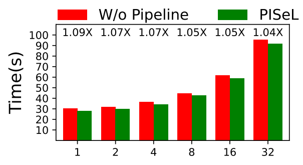
(a) ResNet101(small)



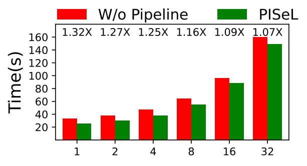
(b) GPT2(small)



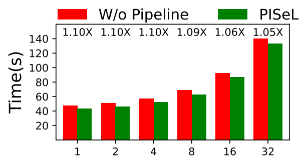
(c) VGG19(medium)



(d) LaBSE(medium)



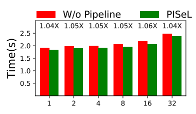
(e) RegNet(large)



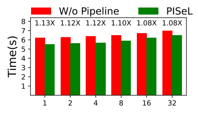
(f) GPT2-XL(large)

Figure 5: TensorFlow GPU

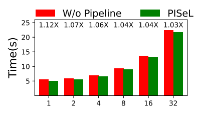
1.36×, with the larger models like GPT2-XL and Regnet seeing the greatest improvements. It has the same reason as the above Tensorflow platform shows. For the larger models, the container creation and framework initialization take a smaller proportion of the total cold-start time, allowing the benefits of pipelining to stand out. On GPU, the speed-ups are even higher as expected, reaching 1.51×. This is attributed to two factors. First, the raw execution time on GPU is much lower than on CPU. Second, the relative loading time is higher on GPU due to another data transmission between CPU and GPU over the PCIe bus. As a result, on GPU there is a greater overlap between the downloading, loading, and execution stages, leading to more significant speed-ups. Interestingly, the peak speed-up is often achieved at an intermediate batch size (e.g., 4 or 8) rather than the smallest or largest size. This is because at very small batch sizes, the execution time is too short to fully overlap with loading, while at very large batch sizes, the execution time dominates the total time, reducing the relative benefit of pipelining.



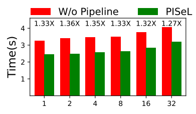
(a) ResNet101(small)



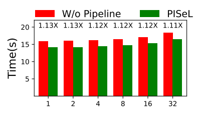
(b) GPT2(small)



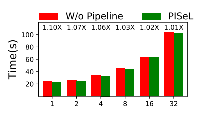
(c) Wav2Vec2(small)



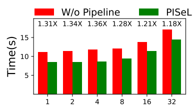
(d) VGG19(medium)



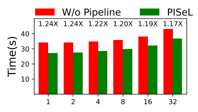
(e) LaBSE(medium)



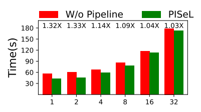
(f) Whisper-M(medium)



(g) RegNet(large)

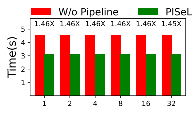


(h) GPT2-XL(large)

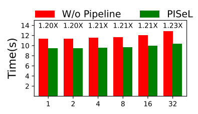


(i) Whisper-L(large)

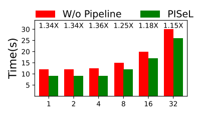
Figure 6: PyTorch CPU



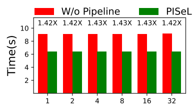
(a) ResNet101(small)



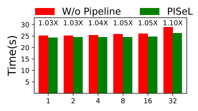
(b) GPT2(small)



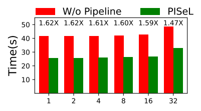
(c) Wav2Vec2(small)



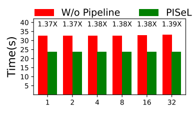
(d) VGG19(medium)



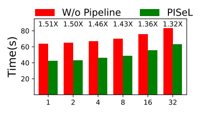
(e) LaBSE(medium)



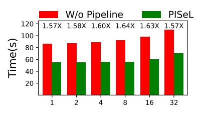
(f) Whisper-M(medium)



(g) RegNet(large)



(h) GPT2-XL(large)

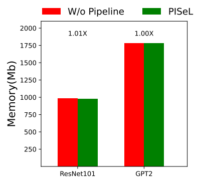


(i) Whisper-L(large)

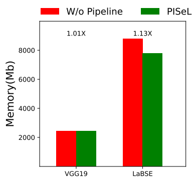
Figure 7: PyTorch GPU

5.2 Peak Memory Usage

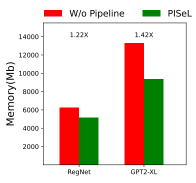
Figure 8 and Figure 9 compare the peak memory usage of PISeL with the original TensorFlow and PyTorch frameworks across small, medium and large models. PISeL consistently reduces the peak memory footprint for all model sizes and frameworks. For PyTorch, the memory savings are most pronounced for large models, with PISeL reducing the peak usage by over 2× for GPT2-XL, Regnet, and Whisper-Large. This is achieved by splitting the model parameters into the different groups, and then loading each group is individually taken at the different time slot. In this way, it can significantly reduce the memory consumption during the loading stage than copying the whole model parameters. For example, loading GPT2-XL with vanilla PyTorch requires over 18GB of GPU memory, while PISeL cuts this down to less than 8GB. The memory savings are smaller but still significant for TensorFlow, ranging from 1.22 – 1.42×. The lower savings are due to an existing optimization in TensorFlow that splits model loading into chunks. However, PISeL still reduces the peak usage by 4GB for GPT2-XL, from 13GB to 9GB, a significant reduction especially when deploying on GPUs with limited memory. For small models like ResNet and Wav2Vec2, the memory overheads are less prominent as the models themselves are much smaller. Nevertheless, PISeL consistently uses less memory than the baselines. These results highlight the importance of PISeL's memory management. By enabling incremental loading, PISeL substantially reduces the memory footprint, allowing larger models to be deployed on a given GPU and enabling denser multi-tenancy.



(a) small

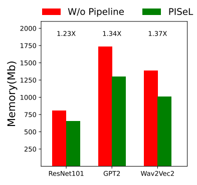


(b) medium

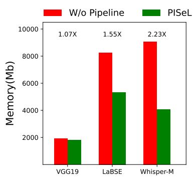


(c) large

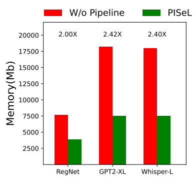
Figure 8: Peak Memory Usage(TensorFlow)



(a) small



(b) medium

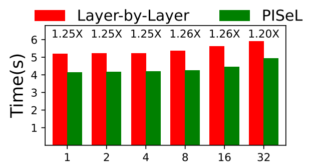


(c) large

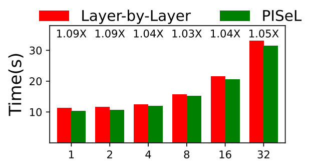
Figure 9: Peak Memory Usage(PyTorch)

5.3 Pipelined Model Transmission and Loading

Figure 10 and Figure 11 shows the total time measured by the client for issuing an inference task until finishing its inference, which includes the latency of container creation, runtime and library loading, and DNN model bootstrap. Layer-by-layer pipeline overlaps the download, deserialization and load, and computation at the granularity of layer. But because it has download and load overhead and synchronization overhead for every layer, it has worse performance on both Tensorflow and Pytorch frameworks for both small and large models. For both Layer-by-layer and PISeL pipeline mechanisms, As the batch size further increases, we do not see the sustained speedup growth. This is because as the larger batch size grows, the computation is expected to increase accordingly, which breaks the well overlapping across the stages if a stage's time is substantial long. Then the pipeline efficiency gets impacted.

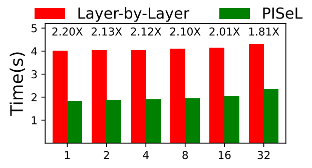


(a) ResNet101(small)

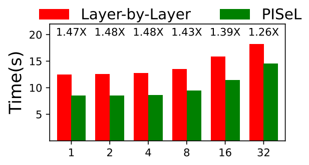


(b) RegNet(large)

Figure 10: Pipeline on TensorFlow



(a) ResNet101(small)



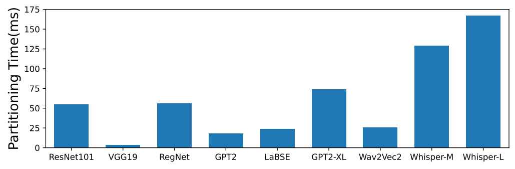
(b) RegNet(large)

Figure 11: Pipeline on PyTorch

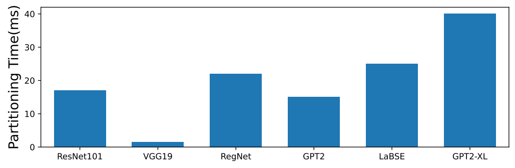
5.4 Partitioning Time

We measure the time taken by PISeL 's optimal grouping algorithm to partition models for PyTorch and TensorFlow. Figure 12(a) shows PyTorch results, while Figure 12(b) shows TensorFlow results.

For PyTorch models, partitioning times range from 3.5 ms for VGG19 to 167 ms for Whisper-L, increasing with model size and complexity. Even the largest model (Whisper-L) partitions in under 200 ms, negligible compared to loading and inference times. TensorFlow models show similar trends, with times from 1.5 ms for VGG19 to 40 ms for GPT2-XL, due to fewer models tested. These times add minimal overhead to inference latency. PISeL 's efficient algorithm achieves these low times by pruning suboptimal partitions based on computation and transmission times and those violating memory constraints, allowing quick optimal partitioning.



(a) Partitioning time for PyTorch models.



(b) Partitioning time for TensorFlow models.

Figure 12: Comparison of partitioning times.

6 Related Work

In this section, we provide a synthetic summary on existing works on serverless computing DNN inference optimization.

Cold-Start Optimization: There have been extensive efforts to improve cold start time. Almost of these works focus only on the optimization of container startup, runtime and libraries. RunD [19], SAND [2], FireCracker [1], Faasm [31] seeked the lightweight virtualization technologies to pursue lower overhead. Seuss[5], Catalyzer[8], Fastlane [17] redesigned the container runtime to optimize the runtime loading. Ping et al. [21], Xanadu [7] pre-creates a container pool with the different resource configurations to hide the container startup time. However, the application-level cold start is often overlooked. PISeL focuses on the optimization of DNN application-level cold start by leveraging the inherent characteristics of DNN models.

DNN Serving: Many techniques and systems have been proposed to improve the performance of DNN serving systems and the resource utilization through the intelligent scheduling like REEF [13], AlpaServe [20], Orion [34], AdaInf [32], Muxflow [42], and Deepboot [6]. There are some other works focusing on batch size selection to optimize the performance of the DNN inference [11, 22]. MLProxy[22] employed a dynamic batching to optimize resource utilization and meet SLAs in serverless ML inference. Tetris [18] explored tensors sharing to reduce the memory usage. Our work is orthogonal to these works. PISeL introduces a novel pipelining approach that is transparent to the DNN framework and serverless platform, and reduces both cold-start latency and memory usage.

Model Pipeline: Pipeline is a classical method to hide latency by overlapping and paralleling the different operations, which is widely used in the computer systems. ByteScheduler [25], Pipedream [24], BytePS [14], and Bamboo [36] overlapped the computation and communication in the distributed training system to accelerate the training time. Pipeswitch [4], and Mobius [10] overlapped and pipelined the communication and computation across the CPU and GPU. PISeL specializes in pipelining for inference workloads in a serverless computing by overlapping the parameter download from the remote storage, deserialization and loading, and computation. The whole pipelining process is designed to be transparent to the DNN jobs and easily maintain the compatibility across the different versions of DNN frameworks.

7 Conclusion

We present PISeL, a system that enables fast application-level cold start for DNN inference in serverless computing. By introducing model pipelining, PISeL reduces cold start time and peak memory usage by overlapping parameter downloads, deserialization, loading, and computation. To optimize this pipeline, PISeL uses a greedy-based partition algorithm to balance pipeline bubbles and synchronization. Additionally, PISeL employs plugin and hook points to ensure transparency and compatibility with various DNN jobs and frameworks. Our experiments on different DNN models and CPU/GPU platforms demonstrate that PISeL significantly reduces cold start time and peak memory usage.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.

[3] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems* 33 (2020), 12449–12460.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[5] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[6] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. 2023. DeepBoot: Dynamic Scheduling System for Training and Inference Deep Learning Tasks in GPU Cluster. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[7] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.

[8] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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*. 467–481.

[9] Fangxiaoyu Feng, Yinfei Yang, Daniel Cer, Naveen Arivazhagan, and Wei Wang. 2020. Language-agnostic BERT sentence embedding. *arXiv preprint arXiv:2007.01852* (2020).

[10] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. 2023. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 489–501.

[11] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[12] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[14] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

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

[16] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 304–312.

[17] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating {Function-as-a-Service} Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 805–820.

[18] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.

[19] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. {RunD}: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 53–68.

[20] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[21] Ping-Min Lin and Alex Glikson. 2019. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221* (2019).

[22] Nima Mahmoudi and Hamzeh Khazaei. 2022. Mlproxy: Sla-aware reverse proxy for machine learning inference serving on serverless computing platforms. *arXiv preprint arXiv:2202.11243* (2022).

[23] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. 2020. Distributed inference acceleration with adaptive DNN partitioning and offloading. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 854–863.

[24] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[25] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[26] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*. PMLR, 28492–28518.

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

[28] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[29] Nick Schneider, Florian Piewak, Christoph Stiller, and Uwe Franke. 2017. RegNet: Multimodal sensor registration using deep neural networks. In *2017 IEEE intelligent vehicles symposium (IV)*. IEEE, 1803–1810.

[30] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *Comput. Surveys* 54, 11s (2022), 1–32.

[31] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[32] Sudipta Saha Shubha and Haiying Shen. 2023. AdaInf: Data Drift Adaptive Scheduling for Accurate and SLO-guaranteed Multiple-Model Inference Serving at Edge Servers. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 473–485.

[33] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[34] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1075–1092.

[35] Sasha Targ, Diogo Almeida, and Kevin Lyman. 2016. Resnet in resnet: Generalizing residual architectures. *arXiv preprint arXiv:1603.08029* (2016).

[36] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 497–513.

[37] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*. 639–653.

[38] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3542929.3563469

[39] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, et al. 2024. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 331–348.

[40] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. 2021. λdnn: Achieving predictable distributed DNN training with serverless architectures. *IEEE Trans. Comput.* 71, 2 (2021), 450–463.

[41] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. 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*. 768–781.

[42] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803* (2023).