FuncPipe: A Pipelined Serverless Framework for Fast and Cost-Efficient Training of Deep Learning Models

YUNZHUO LIU, Shanghai Jiao Tong University, China
BO JIANG, Shanghai Jiao Tong University, China
TIAN GUO, Worcester Polytechnic Institute, U.S.
ZIMENG HUANG, Shanghai Jiao Tong University, China
WENHAO MA, Shanghai Jiao Tong University, China
XINBING WANG, Shanghai Jiao Tong University, China
CHENGHU ZHOU, Chinese Academy of Sciences, China

Training deep learning (DL) models in the cloud has become a norm. With the emergence of serverless computing and its benefits of true pay-as-you-go pricing and scalability, systems researchers have recently started to provide support for serverless-based training. However, the ability to train DL models on serverless platforms is hindered by the resource limitations of today's serverless infrastructure and DL models' explosive requirement for memory and bandwidth. This paper describes FUNCPIPE, a novel pipelined training framework specifically designed for serverless platforms that enable fast and low-cost training of DL models. FUNCPIPE is designed with the key insight that model partitioning can be leveraged to bridge both memory and bandwidth gaps between the capacity of serverless functions and the requirement of DL training. Conceptually simple, we have to answer several design questions, including how to partition the model, configure each serverless function, and exploit each function's uplink/downlink bandwidth. In particular, we tailor a micro-batch scheduling policy for the serverless environment, which serves as the basis for the subsequent optimization. Our Mixed-Integer Quadratic Programming formulation automatically and simultaneously configures serverless resources and partitions models to fit within the resource constraints. Lastly, we improve the bandwidth efficiency of storage-based synchronization with a novel pipelined scatter-reduce algorithm. We implement FUNCPIPE on two popular cloud serverless platforms and show that it achieves 7%-77% cost savings and 1.3X-2.2X speedup compared to state-of-the-art serverless-based frameworks.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Additional Key Words and Phrases: Serverless Function, Distributed Training, Pipeline Parallelism

Bo Jiang is the corresponding author.
Authors' addresses: Yunzhuo Liu, liu445126256@sjtu.edu.cn, Shanghai Jiao Tong University, China; Bo Jiang, bjiang@sjtu.edu.cn, Shanghai Jiao Tong University, China; Tian Guo, tian@wpi.edu, Worcester Polytechnic Institute, U.S.; Zimeng Huang, lukehuang@sjtu.edu.cn, Shanghai Jiao Tong University, China; Wenhao Ma, mwh1233@sjtu.edu.cn, Shanghai Jiao Tong University, China; Xinbing Wang, xwang8@sjtu.edu.cn, Shanghai Jiao Tong University, China; Chenghu Zhou, zhouch@lreis.ac.cn, Chinese Academy of Sciences, China.

47

(a) LambdaML performance.

(b) Training with three configurations.

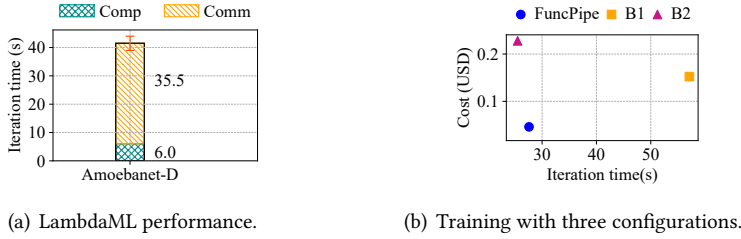Fig. 1. **(a)** LambdaML encounters communication bottleneck when training an AmoebaNet-D model. **(b)** Optimized model partition and serverless resource configurations greatly improve the overall performance.

1 INTRODUCTION

Serverless computing has recently been exploited for distributed training as an alternative to traditional VM-based training [11, 35, 68, 72]. Serverless-based training has many attractive properties. First, it relieves machine learning (ML) practitioners from management tasks such as configuring VMs' environment and setting up distributed training clusters [11, 68]. Second, its true pay-as-you-go pricing helps ML practitioners avoid paying for idle resources, e.g., during the trial-and-error process of model training [68]. Such trial-and-error processes can last a long time: based on our analysis of two popular DL training traces, Philly [32] and Helios [26], users spent more than half of the end-to-end training time on this process. Third, it exhibits good resource elasticity and can auto-scale to many *workers*, i.e., serverless functions [65, 68]. The increased parallelism is especially beneficial for DL training, e.g., the ability to launch many workers for fast hyperparameter tuning and the flexibility to terminate workers for early-stopped configurations [15, 38].

However, today's cloud serverless platforms, e.g., AWS Lambda, impose stringent limits on available memory and bandwidth that make it difficult to utilize them to train resource-intensive DL models directly. Despite recent system efforts in *enabling model training* on cloud serverless platforms [35, 72], ML practitioners still do not have access to fast and cost-efficient serverless-based training. Our empirical analysis reveals the following two key challenges.

First, serverless functions have *restricted communication capability* compared to traditional cloud VMs that does not meet the growing communication demand for training DL models. For instance, the maximum bandwidth of an AWS Lambda function is only about 70 MB/s (0.5 Gb/s) [36, 70] while a VM can have up to 100 Gb/s bandwidth. Moreover, serverless functions lack the ability for *direct inter-function communication*, which makes their communications rely on intermediaries such as Amazon S3 and ElastiCache [35, 68]. Compounding with other training options like data parallelism, existing serverless-based training frameworks can suffer severe communication bottlenecks. Fig. 1(a) shows the average iteration time for training a 900 MB *AmoebaNet-D* with 8 AWS Lambda functions using LambdaML [35], a state-of-the-art serverless-based training framework. The computation takes only 6 seconds for each iteration, while communication takes nearly 6X of that.

Second, serverless functions are allowed a much smaller memory footprint than traditional VMs, hindering their ability to achieve a cost-efficient *computation-to-communication ratio*. For example, AWS Lambda offers up to 10 GB memory size for a serverless function [6], while a VM has up to TBs of memory. In contrast, the memory consumption during training can easily reach tens of GBs and increases with the model size and the *activation size* which is proportional to the batch size. We observe in Fig. 1(a) that increasing the computation-to-communication ratio of the *AmoebaNet-D* model from 0.17 to 0.45 (with local batch size 32) would require about 30GB of memory, far above

the current memory cap of AWS Lambda functions. Existing serverless-based training frameworks provide no effective solution to improve this low computation-to-communication ratio [11, 35, 68].

Our work aims at improving the speed and cost-efficiency of training DL models on cloud serverless platforms. In designing FuncPipe, we address the above challenges through two major approaches, *utilizing model partition techniques* and *improving storage-based communication efficiency*. Our key insight is that model partitioning is not only good for overcoming the memory constraint but also useful in relieving the communication burden in training. Through model partition, we can increase the computation to communication ratio by supporting a larger training batch size (e.g., 32 vs. 8 without partition for AmoebaNet-D model) on each serverless function. Model partition also reduces the size of gradients, compared to data parallelism, on each serverless function but at the cost of additional communication, i.e., exchanging outputs between different partitions. Because these outputs are much smaller than the gradients, the total amount of data transfer with model partition is still a small portion of data parallelism-based training. On the other hand, to further speed up the function-storage communication, we design a new scatter-reduce algorithm for synchronization that pipelines the upload and download tasks. Our pipelined scatter-reduce design *simultaneously* utilizes both uplink and downlink bandwidth of serverless functions, a desirable feature not supported by LambaML, recent work for serverless-based training [35].

At the core, FuncPipe explores pipeline parallelism [17, 27, 49, 60], a type of parallel structure based on the model partition, for *fast and low-cost* serverless-based training. We answer two key design questions: *(i)* how to partition the DL model for the pipeline; and *(ii)* how to allocate resources for each serverless function. Though the first question has been widely studied in server-based pipeline training [17, 49, 63], it poses a more complicated optimization question in a serverless environment. Specifically, those prior work often assume static training resources, i.e., a fixed number of workers with fixed resources, with the goal to only maximize training throughput. In contrast, FuncPipe simultaneously determines the model partition, the number of replicas for each partition (hence the number of workers) and the resource configuration for each worker, with a large search space (i.e., a large number of workers and many possible resource configurations), to optimize for both training throughput and cost.

Fig. 1(b) compares the performance of training an AmoebaNet-D with the model partition and serverless resource allocation configurations found by FuncPipe and two existing algorithms (denoted by B1 and B2). We can see that an efficient configuration can greatly improve the overall performance. Training with configuration found by FuncPipe decreases 52%/70% iteration time/cost compared to B1, and reduces cost by 80% compared to B2 with only 8% time overhead. However, it is nontrivial to identify these effective configurations for different models because the decisions for model partition and resource allocation are *tightly-coupled*. The optimal model partition depends on the allocated resources, and the training performance achieved by the model partition determines whether the resource allocation is cost-effective. Therefore, a joint decision of the two aspects is required, making the optimization problem more challenging.

In short, we make the following main contributions.

- We design and implement FuncPipe; a novel pipelined serverless framework that enables fast and cost-efficient training of DL models with layered structures. FuncPipe provides user-friendly Python APIs that require minimal changes to user code. We make the source code of FuncPipe publicly available [1].
- We propose a novel pipelined scatter-reduce algorithm that utilizes uplink and downlink bandwidth during model synchronization. Our algorithm reduces the synchronization time by

---

[1] https://github.com/liu445126256/FunPipe

6%-26% and the overall iteration time by 2%-18% compared to the non-pipelined scatter-reduce used in LambdaML [35].
- We formulate a co-optimization problem of model partition and resource allocation using Mixed-Integer Quadratic Programming (MIQP). Our optimization approach finds configurations that achieve an average of 80% faster training speed or 55% lower cost compared to existing approaches [10, 63].
- We conduct an extensive evaluation of FuncPipe on two popular serverless platforms with representative DL models. FuncPipe achieves 1.3X-2.2X training speedup and 7%-77% cost reduction, compared to LambdaML, the state-of-the-art serverless training framework [35].

2 BACKGROUND

2.1 Serverless Computing

Serverless computing provides a new paradigm for deploying applications. To use serverless computing on major platforms such as AWS Lambda [6], users upload their applications (including code and dependencies) and execute them as stateless serverless functions. Though serverless users can execute the functions and obtain the computation results without managing the underlying computing infrastructures, users need to configure the functions with the proper amount of resources. The task of resource configuration in today's serverless platforms amounts to deciding the memory allocation; given a memory allocation, other resources like CPU and network bandwidth are allocated accordingly by the cloud providers. Further, users are charged proportionally to the allocated memory and the actual runtime of their applications.

Serverless computing makes it easy to launch many instances of the same serverless function (up to thousands) concurrently; each function instance is often ready to run within seconds or even milliseconds [44, 66]. Serverless provides the true *pay-as-you-go* pricing models and has garnered interests from both industry and academia [1, 16, 25, 54] to run event-driven workloads such as in-memory caching [56, 57, 67] and workloads that benefit from a high degree of parallelism, including distributed training [11, 35, 68, 72]. While prior work focuses on *enabling* distributed DL training on the serverless platforms, this work improves the training speed and cost-efficiency with approaches including pipelining and co-optimization of model partition and resource allocation.

2.2 Distributed Training

Distributed training refers to training a machine learning model with multiple workers that communicate over different networks [13, 76]. When using distributed training, DL practitioners need to make two major decisions, i.e., determining how to divide tasks among workers (parallelism) and how to communicate progress (synchronization).

Parallelism. *Data parallelism* [42, 58] is a widely adopted type of parallelism where each worker maintains a replica of the entire model and a portion of the dataset. In a *training iteration*, i.e., the processing of one batch of data, workers calculate gradients on their local data and then communicate the gradients to update the model parameters. Another type is *model parallelism* [8, 12, 28] where the model is partitioned across workers. Rather than compute the gradients for the entire model, each worker will only compute the data batch on the assigned partition and then communicate the output to the worker that holds the next partition. Consequently, model parallelism often leads to reduced memory consumption and communication data size on each worker, as the total size of transferred data is usually much smaller than that of model gradients in data parallelism. Model parallelism can be further combined with data parallelism by having multiple replicas for each model partition [7, 20, 34]. In such a case, workers working on the same partition need to communicate gradients. Similar to model parallelism, such hybrid parallelism

reduces communication as only gradients for partitions are exchanged when compared to data parallelism. Our work falls under the general hybrid parallelism as we leverage *pipeline parallelism*, detailed in §2.3, where we partition the model and allow data parallelism for each partition.

Synchronization. Distributed training can either use synchronous [17, 27, 29, 39] or asynchronous protocols [43, 49, 75] to instruct when workers can proceed to work on the next data batch. Synchronous protocol, in essence, ensures that workers work on the same version of model parameters by aggregating gradients from all workers to update the model at the end of every iteration. Therefore, it is not subject to potential accuracy convergence issues faced by asynchronous training. In this work, we focus on synchronous training to avoid impact on converged accuracy.

Serverless-based distributed training. In serverless-based training, each worker is mapped to a running serverless function. Existing serverless-based training frameworks [11, 35, 68, 72] are based on data parallelism and differ mainly in their communication designs. There are two major communication architectures, i.e., centralized and decentralized. Parameter Server (PS) is a typical centralized architecture where workers upload their gradients to a central server, and from whom fetch the latest updated model parameters [40, 41]. A recently proposed serverless-based training framework Cirrus [11] adopts such an architecture. In decentralized architecture, workers communicate with each other following the steps of specific communication algorithms, such as all-reduce [52, 53, 55, 64] and scatter-reduce [64]. The state-of-the-art serverless-based training framework LambdaML [35] adopts decentralized architecture and proposes a storage-based scatter-reduce method to combat the performance degradation due to indirect communication via the intermediary storage. Our work also uses decentralized architecture as it is shown to have better scalability in general [35]. The major difference between our work and LambdaML is that our work explores more complicated pipeline parallelism as the key to addressing the performance bottleneck of serverless-based training. We focus on combining pipeline parallelism with serverless design and optimizing the performance of serverless-based pipeline training.

2.3 Pipeline Training and Model Partition

Pipeline parallelism has been explored to improve the resource utilization of traditional server-based model parallel distributed training [17, 27, 39, 49, 60]. At a high level, pipeline parallelism divides a data batch into micro-batches and treats each model partition as a stage in the pipeline. During the training, micro-batches will be scheduled to go through the model partitions in a pipelined fashion to simultaneously utilize resources of different stages. As such, pipeline parallelism can address model parallelism's low resource utilization problem by reducing worker idle time.

One of the key designs to ensure the efficiency of pipeline training is model partition. Although model partition is a well-studied topic in model parallelism [9, 24, 31, 47, 48], the proposed algorithms usually achieve sub-optimal performance when applied to pipeline training due to mismatched goals. The goal for model parallelism is to minimize the processing time of one batch, while in pipeline training, the goal is to minimize the processing time of multiple micro-batches in an iteration. Prior work on server-based pipeline training has proposed several model partition strategies to improve pipeline training throughput [17, 49, 63]. However, they often assume static training resources, i.e., a fixed number of workers with a fixed amount of resources. In serverless computing, we are presented with the flexibility to scale up to many workers and to easily configure workers with different amounts of resources. Such flexibility is a double-edged sword: it gives us more knobs to improve the performance and reduce the monetary cost, while it also makes the problem of configuring pipeline parallelism more difficult. In this work, we tackle the challenge of effectively partitioning the model in pipeline training and configuring resources in a serverless environment to achieve high training throughput and incur low cloud bills.
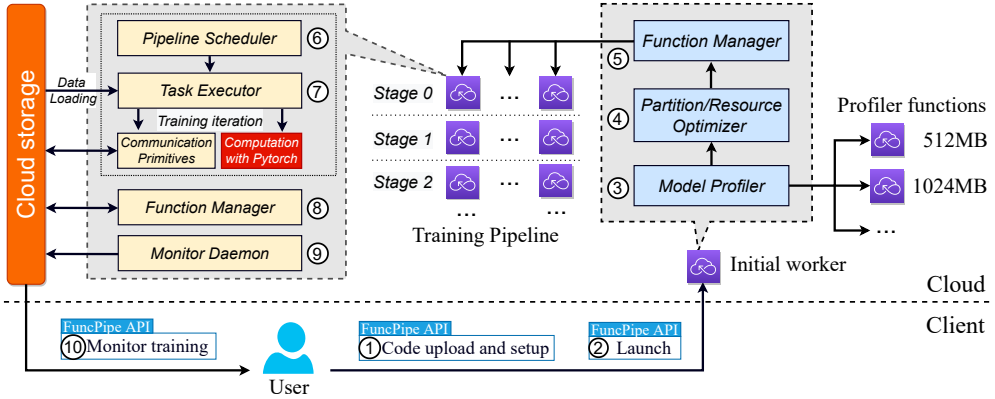
Fig. 2. **FUNCPIPE system architecture and workflow.** The two gray boxes enclose FUNCPIPE components. The blue blocks are the startup components active in the initial worker, and the yellow blocks are the runtime components in a training worker.

3 FUNCPIPE DESIGN

In this section, we present FUNCPIPE, a novel pipelined serverless framework for efficiently training DL models. §3.1 provides an overview of the system architecture and workflow. §3.2 and §3.3 give detailed designs of the training pipeline and pipelined scatter-reduce, respectively. §3.4 presents our co-optimization approach for model partition and resource allocation.

3.1 Overview

System architecture. As shown in Fig. 2, FUNCPIPE consists of three parts, startup components, runtime components, and client-side APIs. The startup and runtime components are displayed in the two gray boxes in the figure, represented by blue and yellow boxes, respectively. Those components run on the serverless platform and interact with cloud storage and client-side APIs. The client-side APIs enable the users to set up, launch, and monitor the training with minimum effort. Our choice of cloud storage as function-to-function communication channel follows the recent work LambdaML [35]. Specifically, we choose object storage, e.g., AWS S3, for its low monetary cost. Even though in-memory storage like Elasticache and DynamoDB provides lower access latency, they are often more costly. Plus, latency has little impact on the performance of serverless-based training, whose communication bottleneck is often caused by the limited function bandwidth.

Workflow. The workflow of FUNCPIPE is shown in Fig. 2. The user first prepares the training code using FUNCPIPE APIs and then sets up and launches training from the client side (① and ②). In the beginning, an initial worker with the startup components performs the preparation work: ③ *Model Profiler* profiles *model layers*, i.e. network topologies in the architecture of the deep learning model such as convolutional layers and fully connected layers, on serverless functions with different memory allocations; ④ With the gathered layer-wise information, e.g., computation time, parameter and activation size, *Partition/Resource Optimizer* finds the optimal model partition and the best resource allocation based on our MIQP formulation (§3.4); ⑤ *Function Manager* configures resources and launches all training workers to start the pipeline.

When the pipeline training starts, micro-batches are scheduled to traverse the pipeline with the help of the following components. ⑥ Each worker runs a *Pipeline Scheduler* and the scheduler decides the processing order of the micro-batches. ⑦ *Task Executor* handles the processing tasks
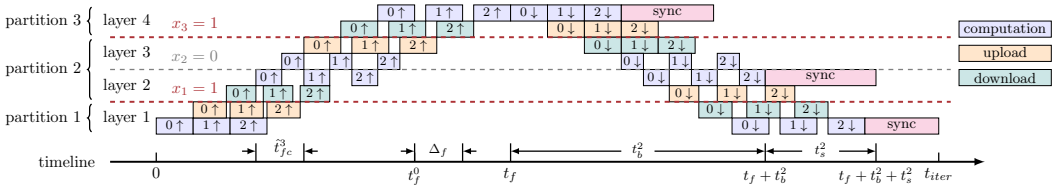
Fig. 3. **Training pipeline of FuncPipe.** Each block represents a processing task. The labeled index indicates the micro-batch that each block corresponds to, and the labeled up/down arrows represent the forward/backward processes respectively. Blocks in the vertical direction can overlap each other in execution. The notations on the timeline are used in the formulation in §3.4.

by interacting with underlying storage-based *Communication Primitives* and *Pytorch*. It properly overlaps communication and computation. ⑧ Each worker runs a *Function Manager*, and the managers exchange information during training to ensure the health of the pipeline. As serverless functions have a limited lifetime, e.g., 15 minutes in AWS Lambda, *Function Manager* checkpoints and restarts the worker at a designated time interval to avoid function timeout. The same procedure is adopted by prior work [11, 35]. Finally, ⑨ *Monitor Daemon* gathers and uploads training information that users can access using client-side API (⑩).

3.2 Training Pipeline

We illustrate the pipeline design of FuncPipe through the example in Fig. 3. FuncPipe uses the pipeline to perform synchronous training that avoids potential convergence and accuracy issues. FuncPipe partitions the model and places each partition on a serverless worker. In a training iteration, the data batch is divided into micro-batches, and they are scheduled to traverse the partitions in the following order: *(i)* all micro-batches go through each partition to perform forward computation; *(ii)* after all forward computations have finished, the micro-batches go in a reversed order for backward computation, i.e., backpropagation.

Each worker in our pipeline generally handles two types of tasks, computation and communication. Communication tasks are further divided into *upload*, *download*, and *sync*. The output of the partitions is communicated through *upload* and *download* to/from the cloud storage; *sync* is required at the end of a training iteration if multiple workers are configured for a partition (i.e., data parallelism). It can be performed once the backward computation of the partition is completed.

Our micro-batch scheduling policy is similar to the one used by GPipe [27], which was designed for server and GPU-based training. Our scheduling policy has two differences: it treats communication tasks as a pipeline stage and overlaps it with the computation task, and it uses a pipelined scatter-reduce algorithm (§3.3) to utilize both uplink and downlink bandwidth for the *sync* task. Our communication-oriented optimization is driven by the key difference between serverless and server-based pipelines, i.e., the proportion of communication time in the overall training time. For example, in the server-based case, communication time is usually negligible as its workers can have large bandwidth, e.g., 100Gb RDMA or 300GB NVlink. In the serverless case, however, communication can take up a large proportion as serverless functions have limited bandwidth. More concretely, both *upload* and *download* times can be comparable to *computation* time. The *sync* time can even be significantly longer depending on the degree of data parallelism.

Other micro-batch scheduling policies [17, 29, 39] could also be used but will lead to more complex pipeline structures and therefore introduce additional complexity in developing the co-optimization approach (§3.4). In other words, we choose the current scheduling policy for its simplicity, and we leave exploring other scheduling methods as future work.
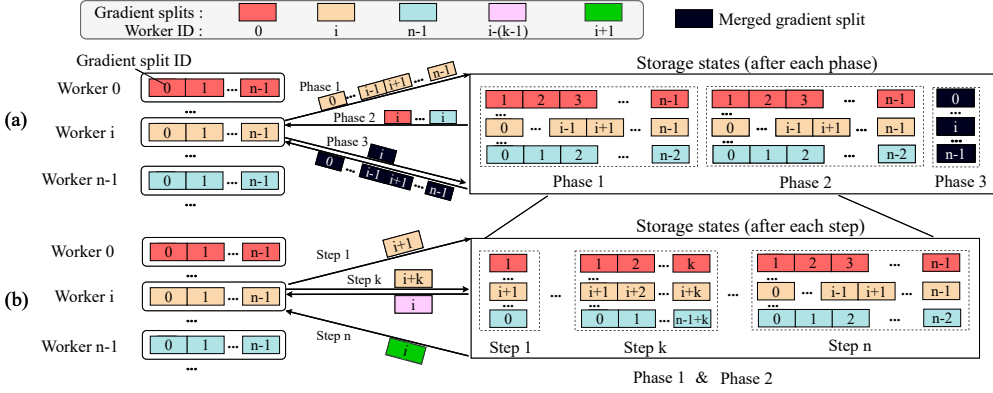
Fig. 4. **Illustration of our pipelined scatter-reduce. (a)** The scatter-reduce in LambdaML [35] has three phases where download and upload are performed serially. **(b)** Our pipelined scatter-reduce performs download and upload in duplex in *phase 1* and *phase 2*.

3.3 Pipelined Scatter-Reduce

We identify one of the causes for the low communication efficiency in existing serverless-based training frameworks [35] as that the current storage-based synchronization design fails to make efficient use of the network bandwidth. To address the problem, we propose a pipelined storage-based scatter-reduce method that simultaneously utilizes downlink and uplink bandwidth. Fig. 4(a) displays the state-of-the-art storage-based scatter-reduce method proposed in LambdaML [35]. It utilizes the computation resource of all workers for gradient aggregation by dividing the gradients as n splits, where n is the number of workers, and each worker is in charge of merging one split. The scatter-reduce process can be divided into three phases: in *phase 1*, each worker uploads the $n-1$ gradient splits that other workers are in charge of to the storage. In *phase 2*, the i-th worker retrieves all the i-th splits uploaded by other workers and computes the merged gradients. In *phase 3*, each worker uploads its merged split and retrieves all other merged splits. The communication time of *phase 1* and *phase 2* are both $\frac{s_{grad}(n-1)}{n \cdot w} + t_{lat}$, where s_{grad} is the size of the gradients, w is the bandwidth of a worker and t_{lat} is the latency for accessing the storage. The communication time of *phase 3* is $\frac{s_{grad}}{w} + 2t_{lat}$, and the total synchronization time is

$$3 \cdot \frac{s_{grad}}{w} - \frac{2s_{grad}}{n \cdot w} + 4t_{lat}. \tag{1}$$

As the upload in *phase 1* and the download in *phase 2* are performed serially, the network resource is not efficiently utilized.

Our scatter-reduce further pipelines *phase 1* and *phase 2* to improve communication efficiency. The pipelined phase includes a total of n steps, as shown in Fig. 4(b):

- In step 1: worker i uploads gradient split $i + 1$ to storage.
- In step k, for $2 \le k \le n-1$: worker i uploads gradient split $i + k$ to storage while downloading split i uploaded by worker $i - (k - 1)$.
- In step n: worker i downloads gradient split i uploaded by worker $i + 1$.

We use arithmetic modulo n in the above. The communication time of each of the above steps is $\frac{s_{grad}}{n \cdot w} + t_{lat}$, and the time for n steps is $\frac{s_{grad}}{w} + n \cdot t_{lat}$. The total synchronization time is

$$2 \cdot \frac{s_{grad}}{w} + (2 + n)t_{lat}. \tag{2}$$

Comparison of (1) and (2) shows that the pipelined scatter-reduce achieves a noticeable reduction in the transfer time, i.e. from $3\frac{s_{grad}}{w} - \frac{2s_{grad}}{n \cdot w}$ to $2\frac{s_{grad}}{w}$. For example, for an AWS Lambda function with 70MB/s bandwidth, the data transfer time of synchronizing a 280MB model among 8 workers can be reduced by 27%, from 11s to 8s. Although our design can suffer higher latency with the increase of workers, the latency is much smaller than the data transfer time, e.g., the measured t_{lat} is less than 40ms for AWS Lambda.

3.4 Co-optimization of Model Partition and Resource Allocation

To make the training pipeline fast and cost-efficient, we need to optimize the partition plan that splits model layers into different pipeline stages and the resource allocation for each stage. This plan includes the number of workers used for intra-stage data parallelism as well as the memory size of each worker. A major challenge here is the strong coupling between model partitioning and resource allocation, which defies most existing solutions that optimize only one aspect [17, 49, 63]. In this section, we formulate the co-optimization of model partition and resource allocation as a mixed-integer quadratic program.

3.4.1 Formulation of Optimization Problem. Consider a model with L layers. Let $\mathcal{D} = \{D_1, \ldots, D_K\}$ be the set of possible degrees of data parallelism, where $D_1 = 1$, meaning no data parallelism. Let $\mathcal{M} = \{M_1, \ldots, M_J\}$ be the set of different memory sizes for serverless workers. We use a binary variable x_i to indicate whether the model is partitioned after layer i. Let $d \in \mathcal{D}$ be the degree of data parallelism. We enforce the same degree of data parallelism for all stages to reduce the problem complexity. Let $m_i \in \mathcal{M}$ be the memory size of workers holding layer i. We parameterize d and m_i as $d = \sum_{k=1}^{K} y_k D_k$ and $m_i = \sum_{j=1}^{J} z_{i,j} M_j$ with binary variables y_k and $z_{i,j}$, where $y_k = 1$ if $d = D_k$ and $z_{i,j} = 1$ if $m_i = M_j$. The number of micro-batches per worker is given by $\mu = \frac{M}{d} = \sum_{k=1}^{K} y_k \frac{M}{D_k}$, where M is the total number of micro-batches. Other notations will be introduced as needed; see Appendix A for a full table of notations.

Our goal is to choose (x_i), (y_k) and $(z_{i,j})$ to minimize the cost c_{iter} and time t_{iter} per training iteration. We formulate it as the *nonlinear binary integer program* in (3), which we explain below.

$$\min \quad \alpha_1 \cdot c_{iter} + \alpha_2 \cdot t_{iter} \tag{3a}$$

$$s.t. \quad \mu \hat{a}_i + \hat{s}_i(4 - 2y_1) + s_0 \le m_i, \qquad 1 \le i \le L; \tag{3b}$$

$$|m_i - m_{i-1}| \le x_{i-1} \cdot M_{\max}, \qquad 2 \le i \le L; \tag{3c}$$

$$\sum_{k=1}^{K} y_k = 1, \quad \sum_{j=1}^{J} z_{i,j} = 1, \qquad 1 \le i \le L; \tag{3d}$$

$$x_i, y_k, z_{i,j} \in \{0, 1\}, \qquad \forall i, j, k. \tag{3e}$$

The expressions for c_{iter} and t_{iter} will be given in §3.4.2. We combine the two objectives into a single objective in (3a) using the weighted sum method. Each pair of weights (α_1, α_2) yields a Pareto optimal solution. As the weights vary, the solutions will trace out the Pareto Frontier [51].

To explain the constraints, we first introduce the hat operator. Given any sequence $u_1, u_2, \ldots, u_L$ where u_i is a quantity associated with layer i, we define

$$\hat{u}_1 = u_1, \quad \hat{u}_i = u_i + \hat{u}_{i-1}(1 - x_{i-1}), \quad 2 \le i \le L, \tag{4}$$

where x_i are our decision variables for model partition. The hat operator accumulates quantities forwardly in each partition. Let $\mathcal{H}$ denote the set of the highest layers of the partitions. For the example in Fig. 3, $\mathcal{H} = \{1, 3, 4\}$. For $i \in \mathcal{H}$, $\hat{u}_i$ is the sum of the quantity u_j over the partition containing layer i. In Fig. 3, $\hat{u}_3 = u_2 + u_3$ is the sum over partition 2.

The constraints (3b) specify that the memory consumption of each partition does not exceed the allocated memory of the corresponding worker. Let s_i denote the parameter size and a_i the activation size per micro-batch at layer i. For $i \in \mathcal{H}$, $\mu \hat{a}_i$ is the memory for activations of μ micro-batches in the partition that layer i belongs to; $\hat{s}_i(4 - 2y_1)$ comprises three parts of memory consumption, $\hat{s}_i$ for parameters, $\hat{s}_i$ for gradients, and $2(1 - y_1)\hat{s}_i$ for serialized data during model synchronization. Note synchronization is needed only if $y_1 = 0$. The quantity s_0 is the basic memory consumption of a serverless worker, e.g., memory consumed by the framework. We only need the constraints for $i \in \mathcal{H}$, as the others are redundant. The constraints (3c) enforce consistency of the memory allocation for adjacent layers if they belong to the same partition, as they actually share the same workers, i.e. $m_i = m_{i-1}$ if $x_{i-1} = 0$. With $M_{\max} = \max_{1 \leq j \leq J} M_J$ being the maximum memory available, the constraint for i becomes vacuous when the model is partitioned after $i - 1$, i.e. $x_{i-1} = 1$. The constraints (3d) and (3e) specify that we choose exactly one degree of data parallelism and exactly one memory configuration for each layer.

To solve (3), we convert it into an MIQP using standard linearization techniques, which is then solved by off-the-shelf solvers, e.g., Gurobi[22]. The details for linearization is in Appendix C.

3.4.2 Performance Model.

Iteration cost. Recall the memory allocated for layer i workers is $m_i = \sum_{j=1}^{J} z_{i,j} M_j$. Since layers of the same partition are assigned to the same workers, we only count the layers in $\mathcal{H}$, and the total memory of all workers is

$$c_{mem} = d \sum_{i \in \mathcal{H}} m_i = d \left(\sum_{i=1}^{L-1} x_i m_i + m_L \right) \tag{5}$$

The cost of serverless functions is proportional to the product of their running time and memory allocation, so the iteration cost c_{iter} is

$$c_{iter} = P \cdot t_{iter} \cdot c_{mem} \tag{6}$$

where P is the unit price specified by the service provider.

Iteration time. As shown in Fig. 3, the iteration time t_{iter} is given by

$$t_{iter} = t_f + \max_{1 \leq i \leq L} (t_b^i + t_s^i), \tag{7}$$

where t_f is the forward time. When layer i is the lowest layer of a partition (e.g., layer 2 in Fig. 3), t_b^i is the backward computation completion time of that partition, and t_s^i the corresponding model synchronization time. For other layers (e.g., layer 3 in Fig. 3), their sum $t_b^i + t_s^i$ will be dominated by that of the lowest layer of the same partition (e.g., layer 2), and hence their inclusion in (7) does not affect t_{iter}. Next we introduce the formulas for t_f, t_b^i and t_s^i in detail.

Forward and backward time. We only show the calculation of the forward time t_f. The calculation of the backward time t_b^i is similar and relegated to Appendix B. The forward time t_f is

$$t_f = t_f^0 + (\mu - 1)\Delta_f,$$

where t_f^0 is the time for the first micro-batch to traverse the forward pipeline, Δ_f the lag between consecutive micro-batches at the end of the forward pipeline, and μ the number of micro-batches

per worker. The time t_f^0 is given by

$$t_f^0 = \sum_{i=1}^{L} t_{fc}^i + \sum_{i=1}^{L-1} (t_{fu}^i + t_{fd}^i),$$

where t_{fc}^i is the forward computation time of layer i, t_{fu}^i the upload time of the output of layer i to the storage, and t_{fd}^i the download time of the output of layer i from the storage to layer $i + 1$. The individual terms are related to $(z_{i,j})$ by

$$
\begin{aligned}
t_{fc}^i &= \beta \sum_{j=1}^{J} z_{i,j} T_{fc}^{i,j}, & 1 \le i \le L, \\
t_{fu}^i &= x_i \left(\sum_{j=1}^{J} z_{i,j} \frac{o_i}{W_j} + t_{lat} \right), & 1 \le i \le L - 1, \\
t_{fd}^i &= x_i \left(\sum_{j=1}^{J} z_{(i+1),j} \frac{o_i}{W_j} + t_{lat} \right), & 1 \le i \le L - 1,
\end{aligned}
\tag{8}
$$

where $T_{fc}^{i,j}$ is the forward computation time of layer i by a worker with memory M_j, $\beta \ge 1$ is the average slowdown factor due to resource contention when we overlap computation and communication, o_i is the output size of layer i, W_j is the bandwidth of a worker with memory M_j, and t_{lat} is the measured latency to storage. The values of $T_{fc}^{i,j}$, β, W_j and t_{lat} are measured by the *Model Profiler* during initial profiling. Note that communication times t_{fu}^i and t_{fd}^i are nonzero only if $x_i = 1$, i.e. there is a partition boundary after layer i.

The lag Δ_f is the maximum time of all stages, i.e.

$$\Delta_f = \max \left\{ \hat{t}_{fc}^{1:L}, t_{fu}^{1:(L-1)}, t_{fd}^{1:(L-1)} \right\},$$

where $t^{i_1:i_2}$ denotes the set of variables t^i for $i_1 \le i \le i_2$, and $\hat{t}_{fc}^i$ is related to t_{fc}^i by (4). For $i \in \mathcal{H}$, $\hat{t}_{fc}^i$ is the computation time for the stage containing layer i. For the example in Fig. 3, $\hat{t}_{fc}^3$ is the time for the second computation stage, consisting of layer 2 and layer 3. Note we only need to include $\hat{t}_{fc}^i$ for $i \in \mathcal{H}$, but the inclusion of the other i gets rid of $\mathcal{H}$.

Synchronization time. When i is the lowest layer of a partition, e.g., layer 2 for partition 2 in Fig. 3, the synchronization time of that partition is

$$t_s^i = (1 - y_1) \left(\sum_{j=1}^{J} z_{ij} \frac{\tilde{s}_i}{W_j} \cdot \gamma + t_{lat} \cdot \delta \right),
\tag{9}$$

where γ and δ are parameters that depend on the synchronization algorithm. For the *pipelined scatter-reduce*, we have $\gamma = 2$ and $\delta = 2 + d$ by (2). The tilde operator is similar to the hat operator in (4), except that it accumulates the quantities backwardly so that $\tilde{s}_i$ of the lowest layer equals the size of the partition. The model update time is negligible and hence not included. Note t_s^i is positive only if the degree of data parallelism is more than 1, i.e. $y_1 \ne 1$. When i is not the lowest layer of a partition, we also define t_s^i by (9). The inclusion of those quantities do not affect the value in (7), since $t_s^i \ge t_s^{i'}$ if $i' \ge i$ and layers i and i' belong to the same partition.

4 IMPLEMENTATION

FuncPipe is implemented on top of *Pytorch* with 4012 lines of Python code. It provides easy-to-use APIs and requires minimal changes to legacy training code on the user side. A code example for using FuncPipe is given in Appendix D. FuncPipe currently supports two serverless platforms, AWS Lambda and Alibaba Cloud Function Compute, and can be easily extended to other platforms as the platform API design in FuncPipe is decoupled from the underlying SDK implementations, e.g., *boto3* for AWS Lambda and *fc2* for Alibaba Cloud Function Compute.

Pipeline task overlap. The different tasks, *upload*, *download*, and *computation* have internal dependencies and different resource requirements, i.e., downlink bandwidth, uplink bandwidth, and CPU. These tasks are organized as Directed Acyclic Graphs (DAGs) and handled by different threads in the *Task Executor*. Tasks of different types are processed in parallel; each is assigned a unique ID and contains a set of IDs representing its dependencies. A task is immediately processed once its dependencies are satisfied.

Communication collectives. FuncPipe performs storage-based communications, including *send-and-receive* between different partitions and *scatter-reduce* among partition replicas. The data communicated are serialized with the python library *pickle* and uploaded to the storage bucket as files. Metadata information is included in the file name to distinguish different pairs and types of communication. Workers periodically query the cloud storage bucket to check for *download*.

MIQP solution. For models with over a hundred layers, solving the MIQP problem can take hours or even days, limiting its practical usage. As many model layers can have small memory consumption and short computation time, they can be merged with other layers to reduce the value of L, i.e. the total number of layers in optimization. By merging the layers, our method ensures a minute-level solution time. Currently, we provide three options for the merging criterion, computation time, parameter size, or activation size. For all the tested models, merging by balancing the computation time achieves better performance and is adopted in our experiments.

FuncPipe provides two implementations for *Partition/Resource Optimizer*. The first one solves the MIQP optimization using serverless functions. However, off-the-shelf solvers can have licence limits that require additional support in order to be used in the serverless environment. For example, Gurobi requires user to have a Gurobi token server that grants temporary license [23]. For users that want to avoid such effort, we provide a second implementation that solves the MIQP optimization at the client side. The information obtained by *Model Profiler* is retrieved by the client for optimization and the results are uploaded back to the initial worker.

Limitation discussion. Currently FuncPipe does not support training models that contains a layer that exceeds the maximum memory for a serverless function. A solution to this problem, and also a possible direction for further optimization is to use tensor parallelism [33, 34, 50, 59, 60]. Tensor parallelism partitions a tensor along specific dimensions, for example, Megatron [60] partitions transformer layer by splitting its weight matrix and FlexFlow [34] partitions CNN layer in terms of both channels and input length. The major benefit we expect from using tensor parallelism is more fined-grained partition decision, which can potentially lead to more cost-efficient resource choices and the avoid of memory overflow caused by super large-sized layers. However, using tensor parallelism increases the complexity of the proposed co-optimization approach, as the extra decision dimensions greatly expand the search space. In such case, techniques like approximation may be required to make the solution of the MIQP practical. We leave extending FuncPipe to tensor parallelism as future work.

Model name	Parameter size (MB)	Activation size per sample (MB)
ResNet101	170	198
AmoebaNet-D18	476	432
AmoebaNet-D36	900	697
BERT-Large	1153	263

Table 1. **Models used for evaluation.** *AmoebaNet-D18* and *AmoebaNet-D36* are two *AmoebaNet-D* models with 18 and 36 normal cell layers, respectively. Both have a filter size of 256.

5 EVALUATION

This section first presents the overall performance of FuncPipe by comparing it with state-of-the-art serverless-based training designs (§5.2) and discusses its system scalability (§5.4). We then validate the effectiveness of FuncPipe's designs with component-wise study, including the evaluation of our pipelined scatter-reduce algorithm (§5.5) and co-optimization of model partition and resource allocation (§5.6). Next, we discuss the effect of resource availability on different serverless platforms (§5.7). Finally, we evaluate the performance of FuncPipe with increased network bandwidth (§5.8).

5.1 Methodology

Cloud serverless testbed. Our evaluation uses two of the mainstream serverless platforms, AWS Lambda [6] and Alibaba Cloud Function Compute [2], that provide different resource options. AWS Lambda provides a maximum of 10 GBs of memory allocation for each serverless function. Its corresponding cloud storage service, *S3*, grants unlimited bandwidth to concurrent access. Alibaba Cloud Function Compute has different resource availability compared with AWS Lambda. It allows a maximum memory allocation of 32 GBs, and its cloud storage *OSS* puts a limit on the concurrent bandwidth, e.g., a total of 10 Gb/s for a normal customer. Most of our evaluations are on AWS Lambda, and we leverage Alibaba Cloud Function Compute to study the impact of resource availability on different serverless platforms.

Models and datasets. The DL models used for our evaluation are in Table 1. *ResNet101*, *AmoebaNet-D18*, and *AmoebaNet-D36* are popular Convolution Neural Network (CNN) models for computer vision tasks. *BERT-Large* is a transformer model for natural language processing. We use the popular image classification dataset *CIFAR-10* to train the CNN models. To train *BERT-Large*, we run masked language modeling on the dataset *Wikitext-2*. We use synchronous Stochastic Gradient Descent (SGD) optimizer with the same global batch size (further explained in §5.2) for all tested designs in the evaluation, and we report the average per-iteration training time and cost.

Baselines. We compare FuncPipe with existing serverless-based training designs with two different structures: the pure serverless-based structure and the hybrid PS structure (as introduced in §2.2). LamdaML [35] is the state-of-the-art pure serverless-based training framework, and it also includes an implementation of the hybrid design exemplified by Cirrus [11], an end-to-end serverless framework for ML training. These two baselines are referred to as *LambdaML* and *HybridPS*. We further integrate *gradient accumulation*, a commonly adopted technique for reducing the memory consumption in training [14, 61, 62]. The resulting baselines are referred to as *LambdaML-GA* and *HybridPS-GA*, both serving as baselines that have reduced worker memory allocation and better-balanced computation to communication time ratio. The baselines and their resource allocation strategies are summarized as follows.

- **LambdaML** follows a pure serverless-based training design. It uses the maximum memory allocation and maximum local batch size within the memory limit for each worker. This strategy reduces the number of workers used for training with a given global batch size.

- **HybridPS** follows a hybrid PS training design and requires the use of parameter servers. We select the instance with the lowest cost that can perform our tasks without incurring CPU or memory bottleneck at the parameter server, i.e., a *c5.9xlarge* instance on AWS and a *r7.2xlarge* instance on Alibaba. The resource allocation of workers follows the same strategy in LambdaML [35] for a fair comparison. Note that we replace the data serialization API in the implementation of [35] with the python `pickle` module to utilize the worker network bandwidth better. We observe that our modification improves training speed and cost. For example, before this modification, HybridPS could only achieve a throughput of about 20 MB/s; the current implementation can fully utilize the bandwidth at about 70 MB/s.
- **LambdaML-GA** applies gradient accumulation to the LambdaML baseline. It uses the same number of workers as LambdaML but allocates *the minimum memory* required after performing gradient accumulation for each worker. We use a batch size of 1 for each accumulation step to minimize memory consumption.
- **HybridPS-GA** uses a similar resource allocation strategy and the same batch size for each accumulation step as LambdaML-GA.

To validate the effectiveness of our co-optimization on model partition and resource allocation, we compare it with two existing algorithms.

- **TPDMP** is the latest graph-based model partition algorithm for server-based pipeline training [63]. It maximizes the pipeline training throughput with a fixed amount of resources. To apply TPDMP to the serverless scenario, we perform a grid search on the resource allocation and optimize the model partition with TPDMP for each allocation. We select the configuration that minimizes the objective function in (3).
- **Bayes** is a black-box optimization method that has been proved effective in deciding cloud configurations [5]. It generates a configuration, measures its performance, and iteratively refines the decision. Bayes can be used to optimize the model partition and resource allocation jointly. However, with the large search space of our problem, Bayes can require many rounds of optimization just to find a feasible configuration. For example, it fails to find configurations that do not cause out-of-memory (OOM) errors for over half of our training tasks within 20 rounds of optimization. To reduce the prohibitive time cost of real-world measurement, we evaluate each configuration with our performance model, which has a high accuracy of 88% as shown in Appendix E. Using performance models in place of the actual measurement is recently proposed and demonstrated to produce good optimization performance [69, 77]. We run a total of 100 rounds of optimization to minimize the objective function in (3).

FuncPipe settings. For the evaluation, we use 8 discrete memory allocation choices, i.e., *[512MB, 1024MB, 2048MB, 3072MB, 4096MB, 6144MB, 8192MB, 10240MB]*. We empirically set the micro-batch size to 4 as it achieves a generally better performance on the evaluation models. We use four pairs of weights of (α_1, α_2), i.e., $[(1, 0), (1, 2^{16}), (1, 2^{19}), (1, 2^{22})]$, to locate the corresponding points on the Pareto Frontier. These weights are chosen empirically because they can generate results that represent very distinct speed and cost trade-offs. The same weights are used for the baseline algorithms TPDMP and Bayes.

Recommendation. FuncPipe also recommends a configuration out of the optimized results, labeled as *Recommendation* in subsequent figures. Denote by t_{mc} and c_{mc} the training time and cost of the minimum cost configuration obtained using weights $(1, 0)$. Assume the training time and cost of a given configuration as t_p and c_p. We use $\delta = (\frac{t_{mc}}{t_p} - 1)/(\frac{c_p}{c_{mc}} - 1)$ to represent how efficient a configuration is by comparing its speedup with its cost increase over the cheapest configuration. In our evaluation FuncPipe recommends the fastest configuration that satisfies $\delta \geq 0.8$.

5.2 Overall Performance

The training performance of FuncPipe and its comparison with existing serverless-based training designs are shown in Fig. 5. Generally, FuncPipe achieves better performance in both training speed and cost over existing designs in most of the test cases (comparable or faster performance in other cases). And the performance improvement increases with the model size and global batch size. The results are obtained with three commonly adopted global batch size of 16, 64, and 256. The performance of each baseline method is represented as a single point in the figure, and for FuncPipe, it is a curve consisting of the points corresponding to the configurations obtained using the four pairs of weights. Note that there can be fewer than four points on a curve as different weights may lead to the same configuration. The configuration recommended by FuncPipe is also highlighted in this figure. We make the following key observations.

First, FuncPipe achieves **1.3X-2.2X** training speedup and **7%-77%** cost reduction compared with the best-performing baseline LambdaML when training *AmoebaNet-D18*, *AmoebaNet-D36* and *BERT-Large* with global batch sizes of 64 and 256. The **2.2X** speedup and **77%** cost reduction is achieved when training *BERT-Large* with global batch size 256. The improved training speed and cost-efficiency come from the reduced communication time and increased computation to communication ratio, as further illustrated in §5.3.

Second, when training on a single worker is feasible, i.e., training with batch size 16, existing designs can achieve cost-efficiency similar to that of FuncPipe since no communication overhead exists. However, our follow-up experiments show that their training speed cannot be further improved given more resources. This is because more resources change the training from a single worker to multiple workers, which incurs prohibitive communication costs for the existing designs. In contrast, FuncPipe can achieve up to 1.6X speedup (training *BERT-Large*) over the best-performing serverless-based training baseline (LambdaML) when given more resources (2.4X cost).

Third, the hybrid design, HybridPS, achieves comparable or even better performance than LambdaML when training *ResNet101*. However, with the increase in model size and global batch size (leading to the use of more workers), the server node in this centralized structure can be heavily burdened. As a result, we can observe noticeable performance gap between HybridPS and LambdaML when training *AmoebaNet-D36* and *BERT-Large* in Fig. 5(c). In addition, we see that the use of gradient accumulation (LambdaML-GA and HybridPS-GA) can reduce the training cost at the price of a longer training time. However, the reduction is neither significant nor guaranteed to exist. We attribute it to the use of gradient accumulation, which reduces the memory allocation but may incur higher costs due to the increased runtime.

5.3 Training Time Breakdown

Fig. 6(a) displays the time breakdown for training *BERT-Large* (Fig. 5(a)). The small batch size allows the baseline methods to train the model on a single worker (no communication time) and thus fully utilize the computation resource and achieve cost-efficient training. However, their training speed cannot be improved any further. As their workers already have the maximum memory allocation, increasing the resource usage means using more workers. Such scaling up incurs high communication costs and stalls the training—synchronizing *BERT-Large* (1153MB) with 70MB/s bandwidth can take tens of seconds, which is longer than the total computation time. This shows that FuncPipe can be faster than existing designs even when training with a small batch size.

Fig. 6(b) shows the time breakdown of training *ResNet101* with batch size 64 (i.e., Fig. 5(b)). The improvement in training speed achieved by FuncPipe is relatively smaller than in Fig. 6(c) and Fig. 6(d). This is because when the model size is small, the synchronization time of LambdaML and HybridPS can be close to the sum of pipeline flush time and intra-stage model synchronization
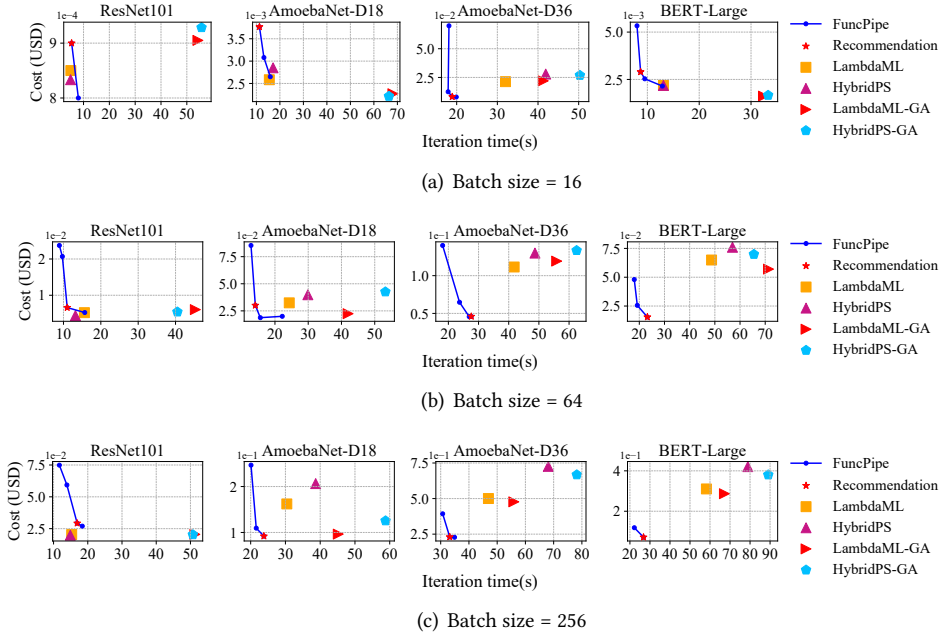
Fig. 5. **Overall performance.** FuncPipe outperforms existing designs in both training speed and cost in most of the test cases and achieves comparable or faster performance in other cases.
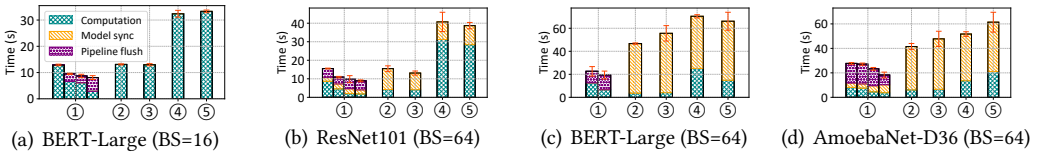

Fig. 6. **Training time breakdown.** Labels: ①FuncPipe, ②LambdaML, ③HybridPS, ④LambdaML-GA, ⑤HybridPS-GA. The multiple bars of FuncPipe correspond to different configurations on the Pareto Frontier. Legend shared across figures.

time in FuncPipe. This suggests that we expect small improvement or comparable performance from FuncPipe with small models.

Figs. 6(c) and 6(d) show the time breakdown for training *BERT-Large* and *AmoebaNet-D36*, respectively (i.e., Fig. 5(b)). The breakdown shows that the performance improvement of FuncPipe in Fig. 5(b) can be largely attributed to the reduced communication time, i.e., its pipeline flush time and intra-stage model synchronization time are much lower than the synchronization time of LambdaML. We can also see that FuncPipe has a larger computation to communication time ratio compared with the baseline methods, making FuncPipe more cost-efficient.

5.4 System Scalability

Next, we evaluate the scalability of FuncPipe by comparing its performance to the best-performing design, i.e., LambdaML, based on observations from §5.2. For this experiment, we use the total amount of allocated memory to denote the system resource. Further, we use the global batch size to
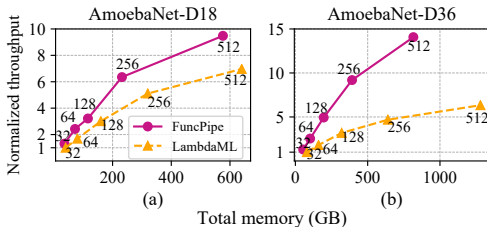
Fig. 7. **System scalability test.** FUNCPIPE achieves higher throughput and is more robust to bandwidth contention. Each data point is annotated with the global batch size.
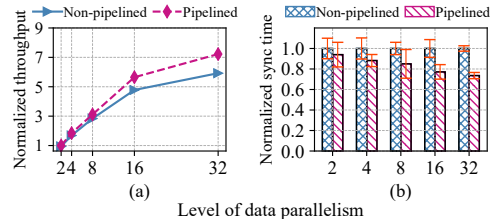
Fig. 8. **Performance of our pipelined scatter-reduce method.** (a) Our design achieves 2%-22% higher training throughput and (b) 6%-26% lower synchronization time.

specify the amount of work. As such, we are evaluating both FUNCPIPE and LambdaML's ability to handle more work (i.e., increased global batch size) given more resources (i.e., total memory). For LambdaML, we increase the global batch size and resource usage by adding more workers. Each worker is allocated the maximum memory and uses the maximum local batch size according to the resource strategy of LambdaML. For FUNCPIPE, we increase the global batch size and use the recommended configuration.

Fig. 7 reports the average training throughput, i.e., number of processed samples per second, on model *AmoebaNet-D18* and *AmoebaNet-D36*. The training throughput is normalized to that of LambdaML with global batch size 32. We first observe that FUNCPIPE achieves higher training throughput than LambdaML when given the same resource allocation. For example, when training the *AmoebaNet-D36* model, the throughput is 180% higher when both use 800 GB total memory. Second, both FUNCPIPE and LambdaML exhibit a sublinear scaling up performance with FUNCPIPE scaling better than LambdaML. We find that reduced per-worker network bandwidth causes the sublinear scaling up performance. The per-worker bandwidth reduction was also observed in prior work [65], and we suspect that it is because the serverless platforms schedule different serverless functions to the same machine, and thus they share a bandwidth capacity. Additionally, we see that FUNCPIPE is less affected by the bandwidth reduction than LambdaML, possibly due to the effectiveness of FUNCPIPE's designs in reducing the overall communication burden.

5.5 Scatter-Reduce Communication Efficiency

We compare our pipelined scatter-reduce design with LambdaML's non-pipelined scatter-reduce [35]. To perform the comparison, we use the recommended configuration for training *AmoebaNet-D18* with a global batch size of 32. The configuration divides the model into three stages, each with a data parallelism of 2. We gradually increase the level of data parallelism (the global batch size is increased proportionally) from 2 to 32 and compare the training throughput. As shown in Fig. 8(a), the two scatter-reduce methods achieve similar performance with small data parallel levels at the beginning (pipelined scatter-reduce has 2% higher throughput). As the data parallel level increases, we observe a growing performance gap, and pipelined scatter-reduce achieves a 22% higher training throughput than non-pipelined scatter-reduce. This increased performance gap can be understood in two ways. First, the increased data parallelism level increases the difference in transfer time of the two algorithms, as seen by comparing (1) and (2). Theoretically, a reduction of up to 33% in transfer time can be achieved. Fig. 8(b) shows that the gap between the synchronization time gradually increases from 6% and reaches 26%. Second, the increased data parallelism level uses more workers. Based on our observation in AWS Lambda, more workers can reduce the available
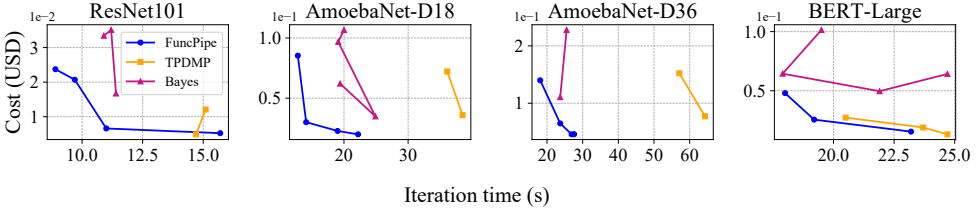
Fig. 9. **Co-optimization performance evaluation.** The global batch size is 64. The performances with other batch sizes are similar.
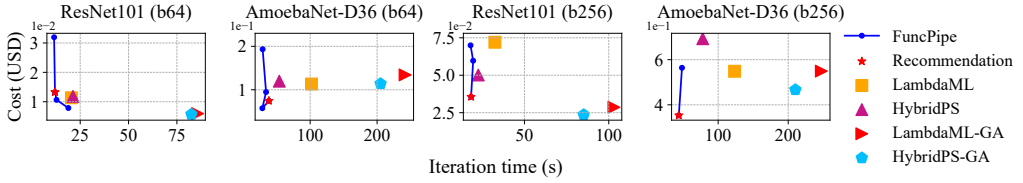

Fig. 10. **Performance on Alibaba Cloud.** With the same limit on total communication bandwidth, FuncPipe achieves up to **1.8X** speedup and **49%** cost reduction compared with the best-performing baseline HybridPS.

bandwidth per worker. As such, the communication time can take up a larger proportion of the overall training time, thus emphasizing the benefit of communication optimization. In summary, our pipelined scatter-reduce can effectively improve communication efficiency.

5.6 Co-optimization Performance

We evaluate the performance of our co-optimization design by comparing it with existing model partition/resource allocation algorithms in terms of training performance and solution time.

Training performance. Fig. 9 compares the model partition and resource allocation policies found by our co-optimization method and those by two existing algorithms [10, 63]. Note that some methods in the figure contain fewer points as they generate the same configuration for different pairs of weights. The results show that our design achieves the best overall performance. Compared to *TPDMP*, our design has a comparable average training cost (within 3% difference) but an average speedup of **1.8X** when optimized for the same objective function. The performance gap between our design and *TPDMP* suggests the benefit of co-optimizing the model partition and resource allocation. Compared to *Bayes*, our co-optimization method achieves **7%** higher average training speed and **55%** lower average cost. We observe that the policies generated by *Bayes* often have higher monetary costs; we attribute *Bayes*'s cost-inefficiency to its tendency to over-provision the resource to avoid infeasible solutions, i.e., policies that lead to OOM error.

Solution time. We evaluate the algorithms on the client side using an *Intel(R) Core(TM) i5-10210U CPU*. The average solution time for each configuration in Fig. 9 is **274s**, **603s**, **45s** for FuncPipe, TPDMP and Bayes respectively. The results show that FuncPipe achieves the best performance with a reasonable solution cost, i.e., minute level. When the optimization problem is solved on the client side, it incurs no cloud bills; when it is solved in the cloud, such minute-level solution cost is negligible to the training cost.

5.7 Impact of Different Resource Availability

We evaluate the performance of FuncPipe on the Alibaba Cloud to understand the potential impact of different resource availability. The major difference between AWS and Alibaba cloud is that the bandwidth of Alibaba Cloud storage OSS [4] has a total limit of 10Gb/s. The same bandwidth limit exists for the VM server used by the HybridPS baseline. We study how the same bandwidth bottleneck affects the performance of these methods. Due to the space limit, we only show the results of training *ResNet101* and *AmoebaNet-D36* with global batch size 64 and 256 in Fig. 10. Overall, we find that FuncPipe demonstrates similar benefits in Alibaba Cloud to AWS: comparable performance or small improvement on small-sized models and better performance in both training speed and cost as the model size and global batch size increase, with up to **1.8X** speedup and **49%** cost reduction compared with the best-performing baseline HybridPS. Note that the best baseline differs from AWS Lambda, as Alibaba cloud functions achieve higher throughput communicating with VM than with the object storage as we observe. This result shows that FuncPipe can alleviate the effect of the limited bandwidth.

Other platforms [45] may have similar limits on the storage-side bandwidth, e.g., Azure Storage has a total limit of 25Gb/s [46]. Such storage-side bandwidth bottleneck may limit the ability of FuncPipe to scale out, and FuncPipe may eventually be outperformed by HybridPS as the bandwidth of the latter can be increased by scaling up the parameter server. One solution for this is to use a VM-based storage design, like Pocket [37], and the total bandwidth can be increased the same way as HybridPS. In this case, we expect FuncPipe to achieve better performance than HybridPS, as the evaluation has demonstrated the performance benefits of FuncPipe under the same bandwidth. Extending FuncPipe to VM-based storage and further comparing to the HybridPS design is left as future work.

5.8 Impact of Increased Function Bandwidth

As our breakdown analysis in §5.3 shows that the improvement achieved by FuncPipe mostly comes from the reduced communication time, we are interested in the performance of FuncPipe when the network bandwidth increases. We simulate the performance of FuncPipe with the performance model proposed in §3.4 by changing the value of bandwidth W. We compare the performance with that of the best-performing baseline LambdaML, which is simulated using its analytical model [35]. Fig. 11 reports the training speed and cost as we gradually increase the bandwidth to 20x of the current function bandwidth in AWS Lambda, i.e., from about 0.5 Gb/s to 10 Gb/s, which is a common bandwidth for a VM.

Generally, as the bandwidth increases, the performances of FuncPipe and LambdaML improve. The performance improvement of LambdaML is larger than that of FuncPipe as LambdaML has a higher communication cost. The relatively mild performance improvement of FuncPipe with the increase of bandwidth suggests that FuncPipe is more robust to different network settings. With 20X the bandwidth, compared with LambdaML, FuncPipe achieves comparable performances on *ResNet101* and *BERT-Large*, i.e., 12.2% higher speed but 7.0% higher cost when training *ResNet101*, 12.9% higher speed but 6.3% higher cost when training *BERT-Large*. The trade-offs in speed and cost are caused by the small differences in the tendencies of the policies of FuncPipe and LambdaML. When training *AmoebaNet-D18/AmoebaNet-D36*, FuncPipe improves the training speed by 6.8%/14.0% while reducing the cost by 6.4%/38.6%. Such improvements are mostly attributed to FuncPipe's optimized function memory allocation. This shows that even with the communication bottleneck removed, the memory allocation policy of FuncPipe can still benefit serverless-based training, although by a smaller margin.
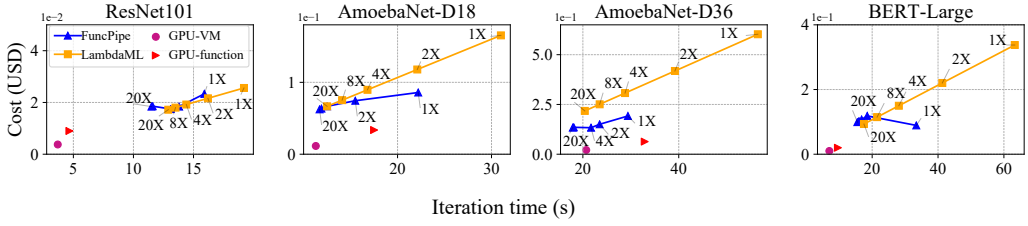
Fig. 11. **Iteration time and cost with the increase of network bandwidth.** We gradually increase the bandwidth to 20X of the current function bandwidth. Note that each curve contains 5 points (some points are overlapped), and they correspond to the results with 1X, 2X, 4X, 8X, and 20X bandwidth, respectively. We also include a point of VM GPU-based training to demonstrate the performance gap with serverless CPU-based training. Such gap can be greatly narrowed once GPU is enabled for serverless function.

Despite the improvements that FuncPipe achieves over existing serverless-based frameworks, a performance gap still exists between training with FuncPipe and GPU-enabled VM instances due to the lack of GPU support in serverless function. We conduct preliminary comparison by training the models on a popular p3.2xlarge AWS instance (equipped with a V100 GPU). As none of the models can be trained on the single GPU without causing memory overflow, we adopt gradient accumulation to reduce the memory consumption. The micro-batch size used for gradient accumulation is 4, the same as the micro-batch size in FuncPipe. The results reported in Fig. 11 show that GPU-based training can greatly outperform serverless CPU-based training in terms of cost, i.e. up to 90% cost reduction. The cause is that the per data sample processing cost of a vCPU can be tens of times higher than that of a GPU. Fortunately, some of the serverless platforms, e.g. Alibaba Cloud, are recently equipping their serverless functions with GPU [3]. Similarly, we report the performance of training with a single serverless GPU function in Fig. 11. Note that as GPU function has yet not been made fully available to users, we evaluate the training speed on a GPU of the same type as the GPU function and obtain the cost with the announced GPU function price. The results show that GPU function greatly narrows gap in the per data sample processing cost with VM GPU instance. It is our next step of work to extend FuncPipe to such GPU function, evaluate its distributed training performance against VM GPUs and explore further optimization.

6 RELATED WORK

Pipeline in serverless-based training. Dorylus [65] is a pipelined framework with a hybrid structure, i.e., CPU servers with serverless functions, for training Graph Neural Network (GNN) models. It exploits the inherent features of GNN to separate the computation tasks and uses serverless functions only for lightweight linear algebra operations. In contrast, our work exploits a serverless-based pipeline for training DNN models, which cannot be easily separated and trained the same way using Dorylus because they require much heavier computation and communication. Hydrozoa [21] proposes a pipelined framework that enables distributed training on GPU-enabled container instances. As serverless function has more stringent resource limits than container instance, our work focuses on providing more efficient communication design and careful co-optimization of model partition and resource allocation to tackle such resource challenges. Note that our optimization designs have the potential to benefit distributed training in other environments like the GPU-enabled container instances, as our preliminary experiments have demonstrated the benefits of our co-optimized model partition and resource allocation policy in GPU-based training.

Serverless communication. Feng et al. propose two centralized storage-based methods for model synchronization [18]. However, such a design is generally of low efficiency due to the bandwidth bottleneck of the central nodes. LambdaML proposes a more efficient decentralized scatter-reduce method, but it fails to utilize the available bandwidth fully [35]. In parallel, other works focus on improving the performance of storage systems for higher communication efficiency. Pocket proposes a distributed data store that provides better elasticity and latency [37]. Shredder designs a low-latency cloud store that supports in-storage computing [74]. This line of work could be integrated with our pipelined storage-based communication approach to improving the network performance potentially. Another choice is to use common *NAT-traversal* techniques to enable direct communication among functions [19, 71]. Direct communication can allow existing communication algorithms, e.g., ringAllreduce [55], to be used. However, NAT-traversal usually requires external servers that can cause communication bottlenecks. The performance of using existing communication designs for serverless-based training with NAT-traversal remains unclear.

Model partition and resource allocation in serverless. Recent works have studied the model partition and resource allocation problem for serverless-based inference serving [30, 73]. These works aim at satisfying Service Level Objectives in latency while minimizing cost or further improving throughput. Gillis fixes the per-function memory allocation and optimizes model partition to lower inference cost with a reinforcement learning approach [73]. AMPS fixes the number of functions/partitions and co-optimizes the partition and memory allocation with a MIP formulation [30]. Compared with inference, the optimization for distributed training, which is the focus of this work, includes more decision factors such as inner-stage data parallelism and synchronization cost and makes it more challenging to generate efficient model partition and resource configuration.

7 CONCLUSION

In this paper, we presented the design and implementation of a novel pipelined serverless training framework called FUNCPIPE. With the ever increasing interests in truly taking advantage of serverless computing, many researchers have looked at utilizing serverless functions to build scalable applications and improving serverless platforms [56, 57, 65, 67]. Our key goal can be simply boiled down to understand *how to allow DL practitioners to train models on serverless platforms* in a fast and low-cost manner, regardless of model size and training hyperparameters such as batch size that impact memory consumption. With three key designs—*(i)* the pipeline parallelism for model partitions, *(ii)* the communication-efficient scatter-reduce, and *(iii)* the co-optimization of partition and resource allocation policy, FUNCPIPE was able to overcome the memory and bandwidth limitations of serverless platforms. We demonstrated the benefits of FUNCPIPE, i.e. **1.3X-2.2X** training speedup and **7%-77%** cost reduction compared to state-of-the-art serverless training frameworks [35], by testing with four commonly used models and on two popular serverless providers in numerous settings. Interestingly, we observed that the benefits of FUNCPIPE remain even if the bandwidth of serverless functions increases to a level comparable to today's VM bandwidth. This observation suggests the relevance of FUNCPIPE techniques even as the cloud providers continue to improve serverless infrastructure.

REFERENCES

[1] 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.

[2] Alibaba. 2022. Alibaba Cloud Function Compute. https://www.aliyun.com/product/fc.

[3] Alibaba. 2022. Alibaba Cloud Function Compute Instance Type. https://help.aliyun.com/document_detail/179379.html.

[4] Alibaba. 2022. Alibaba Cloud Object Storage Service. https://www.aliyun.com/product/oss.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[6] Amazon. 2022. AWS Lambda. https://www.aliyun.com/product/fc.

[7] Ammar Ahmad Awan, Arpan Jain, Quentin Anthony, Hari Subramoni, and Dhabaleswar K Panda. 2020. HyPar-Flow: exploiting MPI and Keras for scalable hybrid-parallel DNN training with tensorflow. In *International Conference on High Performance Computing*. 83–103.

[8] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. 2021. Maximizing Parallelism in Distributed Training for Huge Neural Networks. *arXiv preprint arXiv:2105.14450* (2021).

[9] Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, Mohammad Alizadeh, et al. 2019. Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. *Advances in Neural Information Processing Systems* 32 (2019).

[10] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[11] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[12] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[13] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K Panda. 2020. Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[14] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029* (2017).

[15] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*.

[16] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.

[17] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[18] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring serverless computing for neural network training. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE, 334–341.

[19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 475–488.

[20] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Horizontal or vertical? a hybrid approach to large-scale distributed machine learning. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*. 1–4.

[21] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. 2022. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. *Proceedings of Machine Learning and Systems* 4 (2022), 779–794.

[22] Gurobi. 2022. Gurobi - The Fastest Solver. https://www.gurobi.com.

[23] Gurobi. 2022. Setting up and using a Floating license. https://www.gurobi.com/documentation/9.5/quickstart_mac/setting_up_and_using_a_flo.html.

[24] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. 2021. Towards optimal placement and scheduling of DNN operations with Pesto. In *Proceedings of the 22nd International Middleware Conference*. 39–51.

[25] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, and Venkataramani. 2016. Serverless Computation with Open-Lambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[26] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[27] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[28] Arpan Jain, Ammar Ahmad Awan, Asmaa M Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G Anthony, Hari Subramoni, Dhableswar K Panda, Raghu Machiraju, and Anil Parwani. 2020. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[29] Arpan Jain, Ammar Ahmad Awan, Asmaa M Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G Anthony, Hari Subramoni, Dhableswar K Panda, Raghu Machiraju, and Anil Parwani. 2020. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[30] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency. In *50th International Conference on Parallel Processing*. 1–12.

[31] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 416–430.

[32] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[33] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. 2279–2288.

[34] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[35] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.

[36] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 789–794.

[37] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[38] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *Proceedings of Machine Learning and Systems*, Vol. 2. 230–246.

[39] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[40] Shijian Li, Oren Mangoubi, Lijie Xu, and Tian Guo. 2021. Sync-Switch: Hybrid Parameter Synchronization for Distributed Deep Learning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*.

[41] Shijian Li, Robert J. Walls, and Tian Guo. 2020. Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*.

[42] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018.

[43] Ryan McDonald, Keith Hall, and Gideon Mann. 2010. Distributed training strategies for the structured perceptron. In *Human language technologies: The 2010 annual conference of the North American chapter of the association for computational linguistics*. 456–464.

[44] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[45] Microsoft. 2022. Microsoft Azure Cloud Computing. https://azure.microsoft.com/.

[46] Microsoft. 2022. Microsoft Azure Storage. https://azure.microsoft.com/services/storage.

[47] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. Hierarchical Planning for Device Placement. In *ICLR*.

[48] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*. PMLR, 2430–2439.

[49] 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.

[50] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[51] Patrick Ngatchou, Anahita Zarei, and A El-Sharkawi. 2005. Pareto multi objective optimization. In *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*. IEEE, 84–91.

[52] Pitch Patarasuk and Xin Yuan. 2007. Bandwidth efficient all-reduce operation on tree topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[53] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[54] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. 2019. Towards a serverless platform for edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[55] Baidu Research. 2017. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce.

[56] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. (2021).

[57] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) (*SoCC'21*).

[58] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research* 20, 112 (2019), 1–49.

[59] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems* 31 (2018).

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[61] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631* (2019).

[62] Liuyihan Song, Pan Pan, Kang Zhao, Hao Yang, Yiming Chen, Yingya Zhang, Yinghui Xu, and Rong Jin. 2020. Large-scale training system for 100-million classification at alibaba. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2909–2930.

[63] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems* 33 (2020), 15451–15463.

[64] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[65] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 495–514.

[66] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. 2020. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 1–7.

[67] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. usenix.org, 267–281.

[68] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[69] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. 2020. Learning Search Space Partition for Black-box Optimization using Monte Carlo Tree Search. In *Advances in Neural Information Processing Systems (NeurIPS), 2020* (online).

[70] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[71] Ingo Wawrzoniak, Mike and Fraga Barcelos Paulus Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research*.

[72] 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.

[73] Minchen Yu, Zhifeng Jiang, et al. 2021. Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 138–148.

[74] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–12.

[75] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-Aware Async-SGD for Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. 2350–2356.

[76] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is network the bottleneck of distributed training?. In *Proceedings of the Workshop on Network Meets AI & ML*. 8–13.

[77] Yiyang Zhao, Linnan Wang, Kevin Yang, Tianjun Zhang, Tian Guo, and Yuandong Tian. 2022. Multi-objective Optimization by Learning Space Partitions. *10th International Conference on Learning Representations, ICLR* (2022).

A NOTATIONS IN §3.4

Notation	Definition
s_0	basic memory consumption of a serverless worker
M	total number of micro-batches
L	number of model layers.
P	unit price of serverless function
t_{lat}	latency from serverless worker to cloud storage
s_i	model size of layer i
a_i	size of activations of layer i per micro-batch
o_i	size of output of layer i per micro-batch
g_i	size of gradients from layer i to layer $i-1$ per micro-batch
β	slowdown factor for computation due to resource contention
K	number of data parallelism options
D_k	value of k-th data parallelism option
J	number of resource allocation options
M_j	memory size of j-th resource option
W_j	bandwidth of j-th resource option
$T_{fc}^{i,j}$	forward computation time of layer i with j-th resource option
$T_{bc}^{i,j}$	backward computation time of layer i with j-th resource option
x_i	$\{0, 1\}$, 1 means model is partitioned between layers i and $i+1$
y_k	$\{0, 1\}$, 1 means the k-th data parallelism option D_j is chosen
$z_{i,j}$	$\{0, 1\}$, 1 means layer i workers have j-th memory size M_j
t_{iter}	iteration time
c_{iter}	iteration cost
t_f	forward time for full forward pipeline
t_f^0	time for one micro-batch traverse forward pipeline
Δ_f	lag between micro-batches at end of forward pipeline
t_b^i	backward time until layer i completes computation
t_s^i	model synchronizing time at layer i
t_{fu}^i	time for layer i to upload its output to storage.
t_{fd}^i	time for layer $i+1$ to download input from storage.
t_{bu}^i	time for i to upload gradient output to storage.
t_{bd}^i	time for layer $i-1$ to be download gradient from storage.
d	degree of data parallelism, $d = \sum_{i=1}^{K} y_k D_k$
μ	number of micro-batches per worker, $\mu = M/d$
m_i	memory size of layer i worker, $m_i = \sum_{j=1}^{J} z_{i,j} M_j$
w_i	bandwidth of layer i worker, $w_i = \sum_{j=1}^{J} z_{i,j} W_j$
$\hat{a}_i$	accumulated activation size at layer i (accumulated forwardly).
$\hat{s}_i$	accumulated model size at layer i (accumulated forwardly).
$\tilde{s}_i$	accumulated model size at layer i (accumulated backwardly).
$\hat{t}_{fc}^i$	accumulated forward computation time at layer i (accumulated forwardly).
$\tilde{t}_{bc}^i$	accumulated backward computation time at layer i (accumulated backwardly).

Table 2. Notations in §3.4

B   BACKWARD TIME

The backward computation time t_{bc}^i of layer i, the upload and download time t_{bu}^i, t_{bd}^i between layers i and $i - 1$ are given by

$$t_{bc}^i = \beta \sum_{j=1}^{J} z_{i,j} T_{bc}^{i,j}, \qquad\qquad 1 \le i \le L,$$

$$t_{bu}^i = x_{i-1} \left(\sum_{j=1}^{J} z_{i,j} \frac{g_i}{W_j} + t_{lat} \right), \qquad 2 \le i \le L,$$

$$t_{bd}^i = x_{i-1} \left(\sum_{j=1}^{J} z_{(i-1),j} \frac{g_i}{W_j} + t_{lat} \right), \quad 2 \le i \le L,$$

where g_i is the gradient size from layer i to layer $i - 1$. We introduce a tilde operator similar to the hat operator in (4), except that it accumulates the quantities backwardly. The cumulative backward computation time $\tilde{t}_{bc}^i$ from the previous partition boundary down to layer i is given by

$$\tilde{t}_{bc}^L = t_{bc}^L, \quad \tilde{t}_{bc}^i = t_{bc}^i + \tilde{t}_{bc}^{i+1}(1 - x_i), \quad 1 \le i \le L - 1. \tag{10}$$

For each $1 \le i \le L$, define

$$t_b^i = \sum_{k=i}^{L} t_{bc}^k + \sum_{k=i+1}^{L} (t_{bu}^k + t_{bd}^k) + (\mu - 1)\Delta_b^i, \tag{11}$$

where

$$\Delta_b^i = \max \left\{ \tilde{t}_{bc}^{i:L}, t_{bu}^{(i+1):L}, t_{bd}^{(i+1):L} \right\}.$$

When i is the lowest layer of a partition, t_b^i is the computation completion time of that partition, and Δ_b^i is the corresponding lag between consecutive micro-batches. Note that $t_b^i \ge t_b^{i'}$ if $i' \ge i$ and layers i and i' belong to the same partition.

C   LINEARIZATION

First we present the major linearization techniques used to convert the non-linear binary integer programming to MIQP:

Technique 1: Linearizing the multiplication of two binary variables. $x, y \in \{0, 1\}$, xy can be linearized as follows:

$$f = xy$$
$$f \le x$$
$$f \le y$$
$$f \ge x + y - 1$$
$$f \in \{0, 1\}$$

Technique 2: Linearizing the multiplication of a continuous variable and a binary variable. $x \in \{0, 1\}$, $y \in [a, b]$ is a continuous variable, xy can be linearized as follows:

$$f = xy$$
$$f \le y$$
$$f \ge y - b(1 - x)$$
$$ax \le y \le bx$$

Technique 3: Linearizing of the max operator. x, y, z are continuous variables, $\max\{x, y, z\}$ can be linearized as follows:

$$f = \max\{x, y, z\}$$
$$x \leq f, y \leq f, z \leq f$$
$$x \geq f - H(1 - l_1)$$
$$y \geq f - H(1 - l_2)$$
$$z \geq f - H(1 - l_3)$$
$$l_1 + l_2 + l_3 \geq 1$$
$$l_1, l_2, l_3 \in \{0, 1\}$$

where H is a large constant. Next we introduce how we linearize the formulation in detail.

(1) *Linearizing the equality constraint for the cumulative values $\hat{t}^i_{fc}$, $\tilde{t}^i_{bc}$, $\hat{s}_i$, $\tilde{s}_i$ and $\hat{a}_i$.* We introduce $\hat{t}^i_{fc}$, $\tilde{t}^i_{bc}$, $\hat{s}_i$, $\tilde{s}_i$ and $\hat{a}_i$ as continuous variables and linearize their equality constraints. We use $\tilde{t}^i_{bc}$ in (10) as an example and it is similar with the others. We can write $\tilde{t}^i_{bc}$ as:

$$r_i = 1 - x_i$$
$$\tilde{t}^i_{bc} = t^i_{bc} + \tilde{t}^{i+1}_{bc} r_i$$
$$= \sum_{q=i}^{L} t^q_{bc} \prod_{p=i}^{q-1} r_p$$

Since r_i is a binary variable, $\prod_{p=i}^{q-1} r_p$ can be converted to a new binary variable $\dot{r}_{i,q}$ by recursively performing linearization with **Technique 1**. Then continuous variable $\tilde{t}^i_{bc}$ satisfies the following constraint

$$\tilde{t}^i_{bc} = \sum_{q=i}^{L} t^i_{bc} \dot{r}_{i,q}$$
$$= \beta \sum_{q=i}^{L} \sum_{j=1}^{J} z_{i,j} \dot{r}_{i,q} T^{i,j}_{bc}$$

$z_{i,j}$ and $\dot{r}_{i,q}$ are both binary variables, thus $z_{i,j} \dot{r}_{i,q}$ can be linearized applying **Technique 1**.

(2) *Linearizing the equality constraint for t^i_{fu}, t^i_{fd}, t^i_{bu} and t^i_{bd}.* We introduce t^i_{fu}, t^i_{fd}, t^i_{bu} and t^i_{bd} as continuous variables and linearize their equality constraints. We use t^i_{fu} as an example and it is similar with the others. We can write t^i_{fu} in (8) as:

$$t^i_{fu} = \sum_{j=1}^{J} x_i z_{i,j} \frac{o_i}{W_j} + x_i t_{lat}$$

x_i and $z_{i,k}$ are both binary variables, thus $x_i z_{i,j}$ can be linearized applying **Technique 1**.

(3) *Linearizing forward time t_f and backward time t^i_b.* We use t^i_b as an example and it is similar with t_f. Linearizing t^i_b in (11) is equal to linearizing $(\mu - 1)\Delta^i_b$. Since Δ^i_b is the max of a set of continuous variables, it can be presented as a continuous variable with linear constraints using **Technique 3**. Expand $(\mu - 1)\Delta^i_b$, we have

$$(\mu - 1)\Delta^i_b = \sum_{k=1}^{K} \Delta^i_b y_k \frac{M}{D_k} - \Delta^i_b$$

Since Δ_b^i is a continuous variable and y_k is a binary variable, $\Delta_b^i y_k$ can be linearized applying **Technique 2**.

(4) *Linearizing t_s^i.* Expand (9), we have

$$t_s^i = \sum_{j=1}^{J}(z_{ij}\tilde{s}_i - y_1 z_{ij}\tilde{s}_i)\frac{\gamma}{W_j} + (1 - y_1)t_{lat} \cdot \delta,$$

z_{ij} and y_1 are binary variables, $\tilde{s}_i$ is a continuous variable, thus we can first linearize $z_{ij}\tilde{s}_i$ using **Technique 2** and then further linearize $y_1 z_{ij}\tilde{s}_i$ by applying **Technique 2** again.

(5) *Linearizing full iteration time t_{iter}.* So far we have linearized t_f, t_b^i and t_s^i in (7). We can further remove the max operator using **Technique 3**.

(6) *Linearizing total memory allocation c_{mem}.* Expand (5), we have

$$c_{mem} = \sum_{i=1}^{L-1}\sum_{k=1}^{K}\sum_{j=1}^{J} x_i y_k z_{i,j} D_k M_j + \sum_{k=1}^{K}\sum_{j=1}^{J} y_k z_{L,j} D_k M_j$$

Since x_i, y_k, and $z_{i,j}$ are all binary variables, $x_i y_k z_{i,j}$ and $y_k z_{L,j}$ can be linearized using **Technique 1**.

(7) *Linearizing memory constraint.* At last, we linearize the memory constraint, the first constraint in (3). Expand the constraint, we have

$$\sum_{k=1}^{K} \hat{a}_i y_k \frac{M}{D_k} + 4\hat{s}_i - 2\hat{s}_i y_1 + s_0 \le \sum_{j=1}^{J} z_{i,j} M_j$$

$\hat{a}_i y_k$ and $\hat{s}_i y_1$ can both be linearized with **Technique 2**.

After linearization, t_{iter}, c_{mem} and the constraints in (3) are all in linear form. c_{iter} ((6)) and the objective function are quadratic. The formulation becomes a mixed-integer quadratic program. It has a total of $\max\{o(JL^2), o(JKL)\}$ integer variables, $\max\{o(JL), o(KL)\}$ continuous variables and $\max\{o(JL^2), o(JKL)\}$ linear constraints.

D A FUNCPIPE FUNCTION EXAMPLE

Below is a code example for training with FUNCPIPE. As highlighted in orange, only minimal changes to the Pytorch training code are required.

```python
# Step1: get input training configurations
batch_size = int(event['batch_size'])
loss_func = ...
...

# Step2: build user-defined model and dataloader (Pytorch code)
model = ...
data_loader = ...

# Step3: wrap the model with FuncPipe API
Platform.use(platform_type) # Choose serverless platform
model = FuncPipe(model, loss_func=loss_func, ...) # Config training
model.init(event) # Initialize pipeline

# Step4: start training
for epoch_id in range(epochs):
    for batch_id, (inputs, targets) in enumerate(data_loader):
        model.pipeline_train(inputs, targets)
```

E PERFORMANCE MODEL ACCURACY

Table 3 displays the prediction error in training time for the measured points of FuncPipe in Fig. 5. The results show that our performance model achieves an average prediction error of less than 12%. The largest error happens when training *Amoebanet-D36* with a global batch size of 256. We note that this error is mainly caused by the unexpected bandwidth variation; other model training is less impacted as they use fewer serverless workers and are less subject to the performance interference among workers. We leave the consideration of such interference in our performance model as part of future work.

Model \ Batchsize	16	64	256	Average
ResNet101	5.9%	11.2%	15.4%	10.8%
Amoebanet-D18	13.3%	9.0%	10.6%	11.0%
Amoebanet-D36	10.8%	4.0%	18.1%	11.0%
Bert-large	9.8%	11.0%	16.4%	12.4%
Average	9.9%	8.8%	15.1%	11.3%

Table 3. **Prediction error of FuncPipe training tasks.** Our performance model achieves an average prediction error of less than 12%.