Asteroid: Resource-Efficient Hybrid Pipeline Parallelism for Collaborative DNN Training on Heterogeneous Edge Devices

Shengyuan Ye[♦*], Liekang Zeng[♦*], Xiaowen Chu[▲], Guoliang Xing[◊], Xu Chen[♦†]
[♦]School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
[▲]Data Science and Analytics Thrust, HKUST (GZ), Guangzhou, China
[◊]The Chinese University of Hong Kong, Hong Kong SAR, China
{yeshy8,zenglk3}@mail2.sysu.edu.cn,xwchu@ust.hk,glxing@cuhk.edu.hk,chenxu35@mail.sysu.edu.cn

ABSTRACT

On-device Deep Neural Network (DNN) training has been recognized as crucial for privacy-preserving machine learning at the edge. However, the intensive training workload and limited onboard computing resources pose significant challenges to the availability and efficiency of model training. While existing works address these challenges through native resource management optimization, we instead leverage our observation that edge environments usually comprise a rich set of accompanying trusted edge devices with idle resources beyond a single terminal. We propose Asteroid, a distributed edge training system that breaks the resource walls across heterogeneous edge devices for efficient model training acceleration. Asteroid adopts a hybrid pipeline parallelism to orchestrate distributed training, along with a judicious parallelism planning for maximizing throughput under certain resource constraints. Furthermore, a fault-tolerant yet lightweight pipeline replay mechanism is developed to tame the device-level dynamics for training robustness and performance stability. We implement Asteroid on heterogeneous edge devices with both vision and language models, demonstrating up to 12.2× faster training than conventional parallelism methods and 2.1× faster than state-of-the-art hybrid parallelism methods through evaluations. Furthermore, Asteroid can recover training pipeline 14× faster than baseline methods while preserving comparable throughput despite unexpected device exiting and failure.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computing methodologies** → **Distributed artificial intelligence**.

KEYWORDS

Edge intelligence, distributed machine learning, data parallelism, pipeline parallelism, hybrid parallelism

ACM Reference Format:
Shengyuan Ye[♦*], Liekang Zeng[♦*], Xiaowen Chu[▲], Guoliang Xing[◊], Xu Chen[♦†]. 2024. Asteroid: Resource-Efficient Hybrid Pipeline Parallelism for Collaborative DNN Training on Heterogeneous Edge Devices. In *The 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), September 30-October 4, 2024, Washington D.C., DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3649363

1 INTRODUCTION

Deep Neural Networks (DNNs) have driven diverse intelligence in today's smart applications, ranging from voice assistance, smart robotics to city surveillance, etc. While existing works have extensively studied the inference aspect of DNN models, the growing proliferation of human-in-the-loop intelligent services urgently emphasizes the necessity for privacy-preserving personalization and continuous model refinement, raising the need for advanced on-device learning ability. For instance, in Federated Learning [9, 38], user devices are required to provision a local model training task in order to contribute to and share the learning procedure. In Continual Learning [7, 58], user devices periodically retrain their local models with newly-collected data so as to adapt the model performance to the contextual factors.

Despite the increasing demand, efficient in-situ learning still suffers from its prohibitively long training time and vulnerable convergence stability. As we will empirically show

∗: Equal contributions. †: Corresponding author.

ACM MobiCom '24, September 30-October 4, 2024, Washington D.C., DC, USA

ACM ISBN 979-8-4007-0489-5/24/09…$15.00
https://doi.org/10.1145/3636534.3649363

in §2.1, even training a mobile-oriented compact DNN model in typical edge devices (i.e., Jetson Nano) takes 160× longer epoch time than that in a GPU server, showing the fundamental contradiction between intensive training workload and constrained on-board resources. Moreover, insufficient memory capacity can be a real game-stopper for on-device learning. Towards alleviating these issues, existing wisdom has explored extensive optimizations from various aspects. For example, a number of works adopt model compression techniques (e.g., pruning, sparsification, and quantization) or manually designed lightweight model architectures to reduce the computation complexity for DNN training [8, 27, 32, 45, 59]. Other leading research works have explored to design sophisticated management mechanisms (e.g., tensor rematerialization [12, 23, 43], memory budget adapting [17, 53]) on native resources, but are still bottlenecked by the intrinsic deficiency of physical resource shortage.

In this paper, we alternatively observe that prevalent edge scenarios like smart homes and smart factories usually comprise a group of trusted idle devices beyond a single terminal (e.g., pads, laptops, and smart-home devices owned by the same user or family) [61, 62, 64]. These accompanying devices are typically in physical proximity to the primary one running on-device learning tasks and can be associated as a resource augmentation for in-situ DNN training acceleration. This motivates us to regard adjacent available devices as a resource pool and collaborate with them in a distributed manner to render expedited model training at the edge.

We note that distributed training acceleration has been comprehensively studied in cloud datacenters for years, but is much less understood with edge devices. Nevertheless, scheduling efficient distributed training over edge devices is non-trivial given the unique challenges inherent in edge environments: (1) In contrast to accelerator clusters in cloud, edge devices are extremely limited in terms of computing power, memory capacity, and communication bandwidth. (2) Edge devices are much more heterogeneous compared to cloud server configurations, which necessitates a heterogeneity-aware strategy to maximize the utilization of computing potential. (3) Edge devices frequently exhibit more potential dynamics than dedicated cloud environments, due to the devices' mobility and accessibility. Unfortunately, no existing work can address all of the aforementioned challenges.

To this end, we propose **Asteroid**, a general distributed training system that is able to orchestrate multiple heterogeneous edge devices for expedited, resource-efficient, and fault-tolerant model training. Asteroid's contribution goes beyond merely leveraging distributed edge devices for training acceleration, instead it addresses the above challenges in three levels. First, from a parallelism perspective, a hybrid pipeline parallelism (HPP) is employed as a principle to manage the distributed training workflow, which combines the best of data parallelism and pipeline parallelism and allows significantly larger optimization space for parallelism planning in heterogeneous edge environments. Second, to maximize resource utilization of HPP among heterogeneous edge devices, a novel dynamic-programming based parallelism planning algorithm is designed, as well as a memory-efficient batch ingestion strategy for multidimensional resources optimization including memory budget, limited communication capacity, and resource heterogeneity. Finally, to adapt to dynamic participants during the runtime, a fault-tolerant mechanism is applied through lightweight coarse-granularity workload migration and topology-driven model replication. We implement Asteroid in four realistic testbeds with each consisting of at least 4 heterogeneous edge devices. Extensive evaluations on both vision and language models show that Asteroid is able to deliver up to 12.2× speedup over conventional parallel training counterparts, and achieves up to 2.1× throughput improvement over the state-of-the-art hybrid parallelism methods. Besides, Asteroid can well adapt to device-level dynamics (i.e., device exiting or failure) in agile pipeline recovering time (14× faster than baseline) with minimal throughput sacrifice.

The main contributions are summarized as follows.

- Through extensive measurement studies on on-device and parallel training performance, we employ a hybrid pipeline parallelism as a principled tool to collaborate trusted edge devices for model training acceleration.
- We design a novel planning algorithm tailored for hybrid parallelism mechanism, which comprehensively considers memory budget, limited communication capacity, and resource heterogeneity of edge devices.
- We propose Asteroid, a general distributed edge training system that is able to orchestrate resource-efficient, expedited training across heterogeneous edge devices with fault-tolerant robustness. Asteroid reveals another path towards efficient in-situ model training at the edge.
- We implement Asteroid and evaluate it in realistic heterogeneous testbeds. Experimental results show up to 12.2× throughput improvement over baselines and strong robustness to device dynamics with swift pipeline recovering.

2 MOTIVATION AND PRELIMINARIES

2.1 DNN Training on Resource-Constrained Edge Devices

On-device training can leverage locally collected data to improve model performance while fully preserving data in-situ, making it a widely utilized approach in privacy-sensitive edge applications [7, 38, 41, 48, 66]. However, the resource-intensive and computation-demanding nature of DNN training presents significant challenges for resource-constrained edge devices [17, 32, 53, 57].

Table 1: Elapsed time of a training epoch on devices.

DNN Model	Average Epoch Time		
	A100	Jetson TX2	Jetson Nano
EfficientNet-B1	10sec	11.2min	26.7min
MobileNetV2	9.4sec	8.5min	22min
ResNet50	65sec	1.14hour	3.48hour

To gain insights, we conduct several experiments to analyze how limited computation resource affects on-device DNN training. Specifically, we perform on-device learning for three widely-used DNN architectures on off-the-shelve edge devices and Nvidia GPU platform for comparison, as shown in Table 1. We observe that the average time of a training epoch exhibits a huge gap between A100 and Jetson boards, e.g., 160× and 67× slowdown for Jetson Nano and TX2 when comparing with A100 on MobileNetV2 [45], a representative compact DNN architecture specific for mobile platforms. Memory capacity is another crucial knob of DNN training. However, contemporary edge platforms are incapable of accommodating the memory demands of prominent state-of-the-art DNN models despite their latest release [53].

To mitigate resource constraints, existing works [11, 13, 28, 37, 59] use compression techniques like pruning, quantization, and knowledge distillation to build crafted models dedicated to edge platforms. While these techniques reduce computation, they compromise model accuracy. Other works, such as compilation optimization [17] or run-time memory management [43, 53], can better utilize device resources, but are still restricted by the physical resources of a single device.

Alternatively, in typical edge scenarios such as smart homes, there are usually multiple trusted edge devices in physical proximity that are owned by the same user or family, such that mutually-trustworthy computation resource sharing among these can be achieved. Some pioneering research works have dived into collaborative edge computing to break the resource walls across edge devices [54, 60, 62–64]. However, most works focus only on model inference and few of them manage to address model training.

2.2 Edge Collaborative Training with Data Parallelism and Pipeline Parallelism

Data Parallelism. The most common way to train DNN models in parallel is *data parallelism* (**DP**) [18, 31]. In DP, inputs are partitioned across workers, and each worker maintains a replica of the entire model and performs training on its local data while periodically synchronizing gradients with other workers (i.e., AllReduce). The simplicity of its workload induces better scalability with multiple devices. However, due to the loose and varying edge connections, the communication overhead caused by synchronization can usually dominate training time, as shown in Fig. 1(Left).

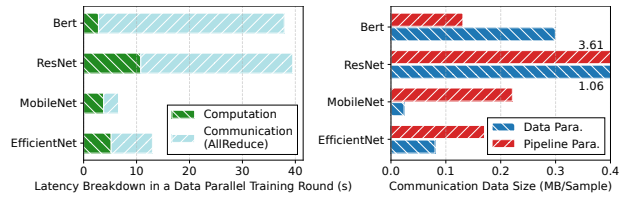


Figure 1: Left: The training latency breakdown in DP. Right: Bytes communicated per sample in DP and PP. Both experiments are conducted on a three-Jetson Nano edge environment with 100Mbps D2D bandwidth.

Pipeline Parallelism. Another widely-used parallelism is *pipeline parallelism* (**PP**). PP is an advanced model parallelism-based training strategy that executes the DNN model in a pipelined manner across multiple workers [22]. Specifically, in PP, the DNN model is partitioned into multiple stages and each stage is mapped to a separate processor for stage-wise forward/backward pass execution. The partitioning ensures each stage has an approximately equal workload, optimizing parallel efficiency. For language model (e.g., Bert-small [14]) or crafted edge model with tiny activations, PP is far more communication-efficient than DP, since each worker only needs to exchange a subset of output activations with neighboring workers. Nonetheless, pipeline parallelism is also followed with shortcomings: (1) *Weak scalability*. The straight-forward implementation of PP for edge clusters can create too many stages, which amplifies the impact of inter-stage communication latency. (2) *Unoverlappable inter-stage communication*. When communication occurs between layers with huge intermediate activations, PP can not effectively overlap the inter-stage communication with forward and backward execution. Our experiment, implemented with Gpipe [22] for PP, shows that the inter-stage communication latency can be up to 24× longer than the stage execution time, which dominates the entire training process. As shown in Fig. 1(Right), for CNN-based models, the per-sample data size communicated in PP even surpasses the communication volume necessitated by DP.

2.3 Key Insight: Combining Data Parallelism with Pipeline Parallelism

The above analysis motivates us to employ a hybrid parallelism architecture that incorporates the best of both DP and PP, so as to achieve superior performance in complex and heterogeneous edge environments. *Hybrid Data Parallelism* (**HDP**), as adopted by Hetpipe [42], organizes devices into groups for inter-group DP and intra-group PP, necessitating a centralized parameter server (PS) for full gradient exchange. Alternatively, another hybrid approach, *Hybrid Pipeline Parallelism* (**HPP**), utilized by PipeDream [39] and Dapple [16], arranges devices into groups for inter-group PP and intra-group DP, as depicted in Fig. 2.

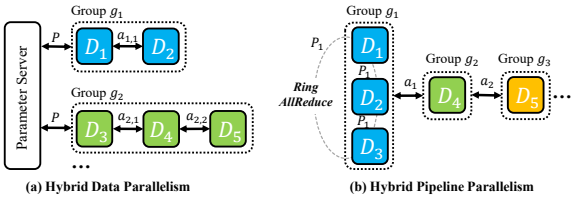


Figure 2: Illustration of HDP and HPP.

To better understand the communication efficiency of these two architectures, we quantitatively analyze their communication volume across devices in a formal way. We first focus on HDP. Assume that each edge device can be and only be divided into a specific group g_i and the considered architecture counts G device groups in total. Let P be the gradient size of the global model, the communication volume for the parameter server bidirectional synchronization is $2GP$. The pipelined communication data size within a group is $2\beta_i \sum_{j=1}^{|g_i|-1} a_{i,j}$, where β_i is the batch size handled by group g_i and $a_{i,j}$ is the size of j-th intermediate tensor in g_i. When there are multiple device groups (e.g., in Fig. 2(a)), we can summarize the above analysis to derive the case of $G > 1$ in Eq. (1). If there is exactly one device group ($G = 1$), however, PS synchronization is free and thus only intra-group communication volume is charged.

$$\mathcal{V}_{\text{HDP}} = \begin{cases} 2GP + \sum_{i=1}^{G} \left(2\beta_i \sum_{j=1}^{|g_i|-1} a_{i,j}\right), & G > 1, \\ 2\beta_1 \sum_{j=1}^{|g_1|-1} a_{1,j}, & G = 1. \end{cases} \tag{1}$$

We next target at HPP. Different from HDP which shares the global model across groups, each group in HPP may take charge of only a part of the model, whose size is denoted as P_i for group g_i. For example, group g_1 in Fig. 2(b) takes a model segmentation of size P_1. Given β as the global mini-batch size and a_i as the size of the intermediate tensor exported by group g_i, each group in HPP requires $2(|g_i| - 1)$ round of ring AllReduce of sub-model P_i in group g_i and the total intra-group communication is thus $\sum_{i=1}^{G} [2(|g_i| - 1)P_i]$. With the inter-group pipelined communication data size as $2\beta \sum_{j=1}^{G-1} a_j$, the total communication volume of HPP can be derived in the case of $G > 1$ of Eq. (2). When $G = 1$, inter-group communications are eliminated and the formula can be simplified into the case of $G = 1$ of Eq. (2).

$$\mathcal{V}_{\text{HPP}} = \begin{cases} \sum_{i=1}^{G} [2(|g_i| - 1)P_i] + 2\beta \sum_{j=1}^{G-1} a_j, & G > 1, \\ 2(|g_1| - 1)P, & G = 1. \end{cases} \tag{2}$$

Upon the above formulas, we further empirically evaluate representative models in Table 1 in an edge environment with five Jetson Nano devices. For HDP, we adopted Hetpipe's allocation recommendations, and for HPP, we adopt Asteroid's planning method (refer to §3.3). The results in Table 2 reveal HDP's communication volume exceeds HPP's by 1.9× - 2.7×. This is because HDP employed by HetPipe necessitates full parameter exchange between groups once the number of groups surpasses one. Conversely, HPP's architecture, through parallelism planning, confines AllReduce operations to initial parameter-light convolution layers, thereby circumventing the final parameter-dense layers.

Table 2: Comparison of communication volume for training a global mini-batch employing HDP and HPP.

Volume	EfficientNet-B1	MobileNetV2	ResNet50
$\mathcal{V}_{\text{HDP}}$ (MB)	171.4	98.0	576.2
$\mathcal{V}_{\text{HPP}}$ (MB)	76.2	52.1	212.4

Opportunities with HPP across Edge Devices. In light of the foregoing analysis, Asteroid employs HPP to facilitate collaboration with edge devices. Beyond breaking the resource wall of a single device, employing HPP offers the following benefits: (1) Each device stores only a subset of the entire model, leading to a smaller memory footprint, which is particularly advantageous for models with huge parameters. (2) HPP offers a highly flexible parallelism architecture that can effectively minimize communication volume by preventing AllReduce in parameter-dense layers. (3) Through layer-wise planning, HPP can avoid inter-stage communication between layers with huge intermediate tensors. By fully overlapping computation and communication, HPP conceals the limitations of network capacity in edge environments. (4) HPP provides higher scheduling flexibility and expands a larger optimization space of parallelism planning in our considered complex and heterogeneous edge environments.

2.4 Technical Challenges

Despite the benefits, realizing HPP in complex edge environments still suffers from a set of challenges.

Scarce Memory and Network Capacity. Edge devices, constrained by limited memory and bandwidth through links such as WiFi and cabled network [21], require meticulous planning for the HPP architecture to prevent out-of-memory issues and maximize bandwidth efficiency. Successfully leveraging the architecture's flexibility to address these constraints presents a significant challenge.

Heterogeneous Computing Resource. Edge environments are typically heterogeneous [66], with facilities ranging from small devices and gateways [10] to much powerful cloudlets [46]. Efficiently applying HPP in such highly heterogeneous settings is a particularly challenging aim of maximizing its resource utilization, which requires judiciously matching workload distribution to diverse edge resources.

Dynamic Training Participants. Further complicating the problem is the inherent, unpredictable dynamics of available edge resources, due to devices moving across networks and multitasking [51, 53]. To render stable, reliable training performance with HPP therefore poses significant challenges in designing a robust scheduling such that training participants' failure is tolerable.

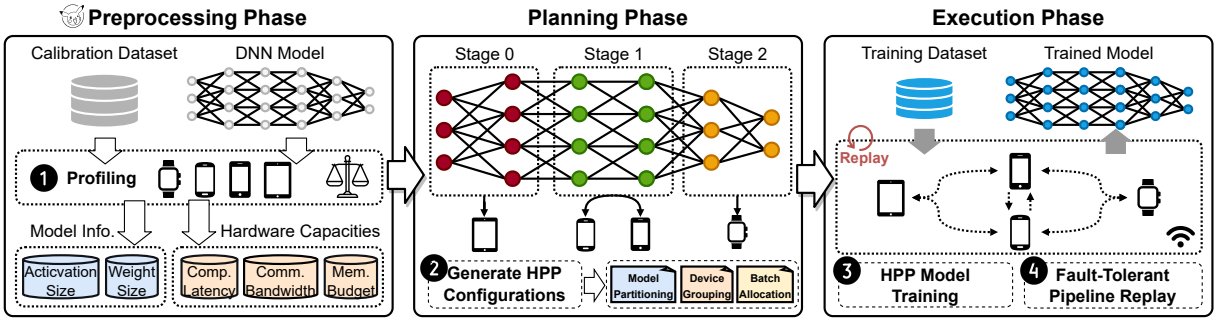
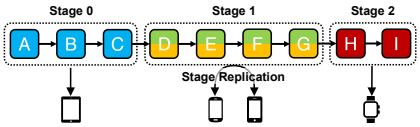
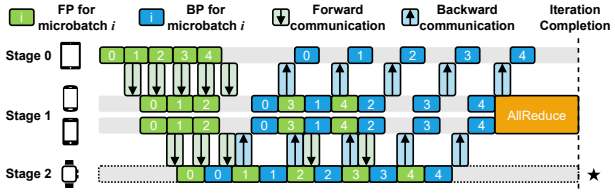


Figure 3: Asteroid Overview: A three-phase workflow includes Preprocessing, Planning, and Execution Phase.



(a) The computation graph is partitioned into three stages, where Stage 1 is replicated on a device group with two devices for intra-stage data parallelism.



(b) Training pipeline of 5 micro-batches. The numbers in the cells represent micro-batch ids. AllReduce is performed in Stage 1 for model synchronization. The star marks the dominant step (will be introduced in §3.3).

Figure 4: An instance of HPP with four edge devices.

3 ASTEROID SYSTEM DESIGN

3.1 System Overview

Fig. 3 depicts an overview of our proposed Asteroid which features three primary phases: *Preprocessing Phase*, *Planning Phase* and *Execution Phase*. *Preprocessing Phase* is an offline procedure that runs once before deployment. *Asteroid Profiler* performs a training process using calibration data as input on the physical edge devices to record the runtime profile necessary for parallelism planning (step ❶). During *Planning Phase*, *Asteroid Planner* takes profiling results as input to generate planning configurations that include DNN model partitioning points, device grouping strategies, and micro-batch allocations within groups (step ❷). These configurations comprehensively addresses challenges including memory budget, limited communication capacity, and resource heterogeneity, and is subsequently applied to target DNN models and training participants. In *Execution Phase*, *Asteroid Worker* which is deployed on each participant will be responsible for model execution, intermediate output exchange and gradient synchronization (step ❸). To account for the challenge of runtime dynamics, a fault-tolerant pipeline replay

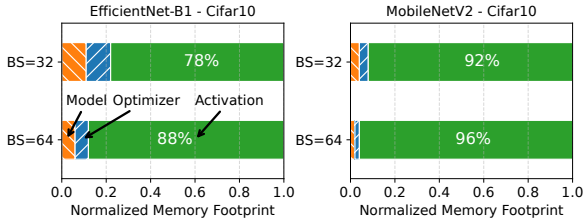


Figure 5: Breakdown of the memory footprint during DNN training, profiled on a Jetson NX.

mechanism will be applied (step ❹). A central coordinator (a user-specified device) is required to apply planning configuration and detect device failure.

3.2 Hybrid Pipeline Parallelism in Asteroid

HPP Architecture and Workflow in Asteroid. As illustrated in Fig. 4(a), Asteroid's HPP first divides a DNN model into multiple **stages** where each contains a **stage model** composed of a set of consecutive network layers. Edge devices in the resource pool will be divided into a corresponding number of **device groups**, where each contains one or multiple devices. HPP combines pipeline parallelism across these groups with data parallelism within them. During training, a mini-batch will be split into M smaller **micro-batches** (with the size of B) and injected into the pipeline concurrently to increase parallelism. Micro-batches are broken up further if device groups contain multiple devices. Each device performs the *forward pass* (**FP**) and *backward pass* (**BP**) for the stage model it takes in charge and accumulates gradients for all micro-batches in each mini-batch. At the end of a mini-batch, gradients in each device group are synchronized using **AllReduce** and then applied to stage model parameters. The entire mini-batch training process is called an **HPP-Round**. Fig. 4(b) shows a well-designed scheduling arrangement for Asteroid HPP training that models inter-stage network communication, which cannot be neglected due to the significant latency under low-speed links in edge environments. **Bubbles** are idle periods when a stage waits for data from the previous one, represented by gray blocks.

Memory-efficient 1F1B Micro-batch Scheduling. Our memory breakdown experiment in Fig. 5 shows that the peak

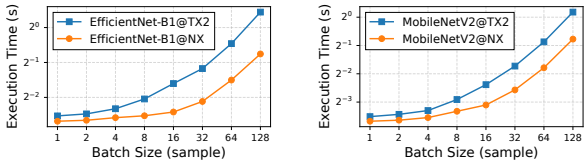


Figure 6: DNN execution time across diverse batch sizes, profiled on Jetson TX2 and Jetson NX.

memory footprint during DNN training can be classified into three categories: (1) model weight memory (including model parameters and accumulated gradients), (2) optimizer memory, and (3) activation memory (intermediate outputs of FP). HPP architecture enables each device to only store the corresponding stage model in memory. In stage p, the memory footprints for model weights, optimizer, and activations for a single micro-batch of size β are denoted as $\text{Mem}_p^{(\text{MOD})}$, $\text{Mem}_p^{(\text{OPT})}$, and $\text{Mem}_p^{(\text{ACT})}(\beta)$, respectively.

The experiment in Fig. 5 shows that the intermediate activations are the main contributor to memory footprint, especially for the CNN-based models which usually have large inter-layer feature maps. Gpipe [22] schedules micro-batches in a *backward-after-forward* manner, resulting in a peak memory demand that scales proportionally with the number of concurrently resident micro-batches ($O(M)$), which is memory-unfriendly for edge devices. Inspired by the idea of gradient accumulation, we adopt a fine-grained micro-batch scheduling that works in a *one-forward-one-backward* (**1F1B**) manner, which schedules the BP early to release the activation memory produced by FP for reuse. We propose performing K_p FP for each stage p before strictly enforcing 1F1B (as illustrated in Figure 4(b), where $K_0 = 5, K_1 = 3, K_2 = 1$), resulting in an activation memory requirement of $O(K_p)$ ($K_p < M$) for each stage p. Setting a smaller K_p can reduce memory footprint but compromise stage-level pipeline concurrency. Specifically, with $K_p = 1$ for all stages, only one stage will be activated concurrently. Our experiments in §5.4 reveal that setting $K_p = 2 \times (P - p) - 1$ (P is the total number of stages) can minimize the peak memory footprint of each stage without sacrificing parallelism efficiency. With the above modeling, the total memory footprint of stage p with a micro-batch size β in Asteroid is as follows:

$$\text{Mem}_p(\beta) = \text{Mem}_p^{(\text{MOD})} + \text{Mem}_p^{(\text{OPT})} + K_p \times \text{Mem}_p^{(\text{ACT})}(\beta). \tag{3}$$

3.3 Parallelism Planning

DNN Model and Asteroid Profiler. We consider a DNN model as a directed acyclic graph. The graph nodes represents modules like Conv, MaxPool, Attention, etc while the graph edge encodes the data dependency between modules. In order to split the model into multiple sequential stages, we topologically sort the graph nodes and transform the DNN model into layers sequence. Asteroid profiler precisely collects the total output size and weight parameters in bytes for each layer. We denote the output activations (and corresponding input gradients) and weight parameters of layer l as a_l, and w_l, respectively.

Table 3: Table of Notations for Parallelism Planning

Notation	Definition
L	Total number of layers in the DNN model.
N	Total number of edge devices involved.
M	Number of micro-batches processed concurrently in a single HPP-Round.
B	Size of each micro-batch.
P	Total number of pipeline stages in the HPP.
S	Total number of communication and execution steps in the HPP.
a_l	Size of output activations for the layer l.
w_l	Size of weight parameters for the layer l.
u_d	Memory budget available on device d.
v_d	Computational capacity of device d.
$b_{d,d'}$	D2D bandwidth between devices d and d'.
$t_f^{d,l}(\beta)$ / $t_b^{d,l}(\beta)$	Time to perform FP / BP for layer l on device d with a batch size of β.
T_w^s / T_e^s / T_a^s	Execution time for the Waiting / Execution / AllReduce phases in step s.
E_f^s / E_b^s	Time taken for FP / BP in step s.
$\mathcal{G}_s$	Device group associate with step s.
$\mathcal{D}_s$	Sub-model associate with step s.
$\mathcal{Y}_s$	Allocation of micro-batch samples across devices in the group $\mathcal{G}_s$.

We further profile the FP and BP execution time of each layer. Existing works [25, 35] simply assume a linear relationship between batch size and execution time. However, our experiment reveals that smaller batch size fails to fully leverage the parallelism capacity of GPUs, leading to a non-linear correlation between them, as shown in Fig. 6. Therefore, Asteroid profiler measures the execution time of each layer on realistic edge hardware for various batch sizes. We denote $t_f^{d,l}(\beta)$ and $t_b^{d,l}(\beta)$ as the FP and BP execution time for layer l on device d using a batch size of β. We profile the D2D bandwidth between devices d and d' as $b_{d,d'}$.

Optimization Objective Formulation. As illustrated in Fig. 7, to emphasize the non-negligible communication latency in edge environments, we abstract the process into separate **steps**, categorizing them as either **execution steps** or **communication steps** for inter-stage communication and stage model execution. We divide the training process on each step in an HPP-Round into three phases: *Waiting Phase*, *Execution Phase*, and *AllReduce Phase*, with corresponding times denoted as T_w^s, T_e^s, and T_a^s. Our optimization objective is to minimize the *HPP-Round Latency*, which is determined by the step with the largest total latency of three phases:

$$\text{HPP-Round Latency} = \max_{s \in \{0,1,\ldots,S-1\}} \left(T_w^s + T_e^s + T_a^s\right), \tag{4}$$

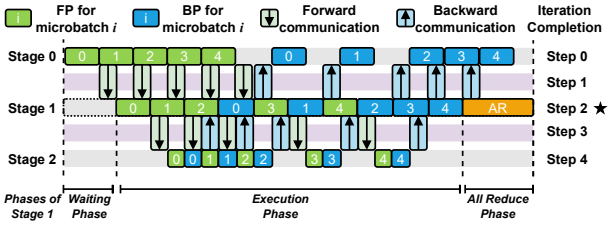


Figure 7: In this example, Step 2 is the dominant step, which is marked with a star.

where S denotes the total steps in pipeline. For notation simplicity, we denote the forward (backward) execution or communication time of a step s as E_f^s (E_b^s), which is associated with a device group $\mathcal{G}_s$ and a sub-model $\mathcal{D}_s$ depending on the pipeline stage of step s ($\mathcal{G}_s$ and $\mathcal{D}_s$ are empty sets for communication steps). The time of *Waiting Phase* and *AllReduce Phase* can be estimated by:

$$T_w^s = \sum_{i=0}^{s-1} E_f^i, \quad T_a^s = \frac{2\,(|\mathcal{G}_s|-1)\cdot \sum_{l\in\mathcal{D}_s} w_l}{|\mathcal{G}_s|\cdot \min_{d,d'\in\mathcal{G}_s} b_{d,d'}}. \tag{5}$$

T_w^s is the sum of FP time till step $s-1$. The size of communication volume per device in $\mathcal{G}_s$ during AllReduce is $\frac{2(|\mathcal{G}_s|-1)}{\mathcal{G}_s}(\sum_{l\in\mathcal{D}_s} w_l)$. The time taken by the AllReduce operation, denoted by T_a^s, is further decided by the minimum connection bandwidth among all devices [35, 47].

A remaining hurdle is how to estimate T_e^s of each step. As illustrated in Fig. 4, the *Execution Phases* of all steps always form a trapezoid shape due to data dependency. Therefore, with an accurate latency estimation of one step, we can infer the T_e^s of the other steps by considering the shift in FP and BP time before and after it. We observe that there always exists a step such that the FP and BP are compactly injected in its *Execution Phases*, which is the dominant factor in estimating *Execution Phase* latency. Thus, we define the step with the fewest number of bubbles during *Execution Phase* as **dominant step** and its *Execution Phases* latency can be well approximated as the accumulated time of FP and BP of M micro-batches. Although the dominant step may contain a small fraction of bubbles, leading to an approximation to the true pipeline latency, it has been practically effective in all our evaluations. As shown in Fig. 7, step 2 is the dominant step without bubble. Conversely, the remaining steps contain scattered bubbles that cannot be accurately estimated, rendering them unsuitable as the dominant step. Assuming that the index of the dominant step for a pipeline is dm, the T_e^s of other steps can be estimated by:

$$T_e^s = M \times \left(E_f^{dm} + E_b^{dm}\right) + \begin{cases} \sum_{i=s}^{dm-1}\left(E_f^i + E_b^i\right), & s < dm, \\ -\sum_{i=dm}^{s-1}\left(E_f^i + E_b^i\right), & s \geq dm. \end{cases} \tag{6}$$

Based on the aforementioned formulation, once E_f^s and E_b^s are determined, we can calculate the HPP-Round Latency according to Eq. (4). In the following, we first derive and formulate algorithms to estimate E_f^s and E_b^s. We then design an innovative dynamic programming algorithm to identify the optimal HPP architecture from all possible configurations, effectively minimizing HPP-Round Latency.

Algorithm 1: Allocation of a micro-batch's samples

Input: $\mathcal{G}_s$: Device Group of step s. B: Micro-batch Size.
Output: $\mathcal{Y}_s = \{y_d | d \in \mathcal{G}_s\}$: Micro-batch allocation of step s.

Function `MemoryAwareBalancing`$(\mathcal{G}, \beta)$**:**
  if $|\mathcal{G}| == 0$ *and* $\beta > 0$ **then**
    Exit with Fail; ▷ Set $T(i \to j, \mathcal{G})$ as ∞
  else if $\beta == 0$ **then**
    Return;
  foreach $d \in \mathcal{G}$ **do**
    $bs_d \leftarrow$ Maximum batch size of d under budget u_d;
    $y_d \leftarrow y_d + \min(\frac{v_d}{\sum_{d\in\mathcal{G}} v_d}\beta, bs_d)$;
    Update u_d with remaining memory budget;
  $\mathcal{G}' \leftarrow$ Devices in $\mathcal{G}$ with remaining memory;
  $\beta' \leftarrow$ Unallocated batch size in β;
  `MemoryAwareBalancing`$(\mathcal{G}', \beta')$;
Function `StragglerWorkloadOffloading`**():**
  Sort devices within $\mathcal{G}_s$ in ascending order based on their execution latency with $\mathcal{Y}_s$;
  do
    old_straggler $\leftarrow$ The slowest device in $\mathcal{G}_s$;
    Transfer a block of samples from the straggler to the fastest device with sufficient memory;
    Reorder devices within $\mathcal{G}_s$ in ascending;
    new_straggler $\leftarrow$ The slowest device in $\mathcal{G}_s$;
  while *new_straggler faster than old_straggler*;
`MemoryAwareBalancing`$(\mathcal{G}_s, B)$; ▷ Phase 1
`StragglerWorkloadOffloading`(); ▷ Phase 2

Estimating E_f^s and E_b^s for step s. For the communication steps, E_f^s and E_b^s can be estimated by the size of the intermediate tensors required for transmission between stages and the profiled D2D communication bandwidth. In the following, we concentrate on estimating E_f^s and E_b^s for execution steps, with a key objective being the optimal allocation of a micro-batch's samples among resource-diverse devices. This allocation seeks to minimize data parallel execution time within the memory budget of each device. We denote $T(i \to j, \mathcal{G}_s)$ as the optimal time taken by device group $\mathcal{G}_s$ spanning layers i through j for both FP and BP. The optimization target can be formulated as follows:

$$T(i \to j, \mathcal{G}_s) = \min_{y_d\in\mathcal{Y}_s} \max_{d\in\mathcal{G}_s} \sum_{l=i}^{j} [t_f^{d,l}(y_d) + t_b^{d,l}(y_d)], \\ s.t. \sum_{d\in\mathcal{G}_s} y_d = \mathrm{B}, \quad \mathrm{Mem}_s(y_d) \leq u_d, \tag{7}$$

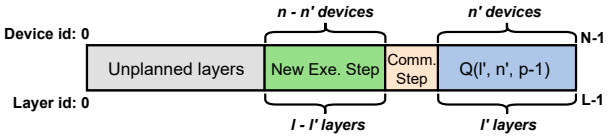


Figure 8: An instance of dynamic programming planning process for $\widetilde{Q}(l, n, l', n', p-1)$.

where $\mathcal{Y}_s = \{y_d | d \in \mathcal{G}_s\}$ defines the set of the allocation of a micro-batch across devices in step s, with each element y_d representing the number of samples allocated to device d. The peak memory footprint is given by $\text{Mem}_s(y_d)$ (refer to Eq. (3)). u_d denotes the memory budget of device d. E_f^s and E_b^s can be estimated as follows once the $\mathcal{Y}_s$ is determined.

$$E_f^s = \max_{d \in \mathcal{G}_s} \sum_{l=i}^{j} t_f^{d,l}(y_d), \quad E_b^s = \max_{d \in \mathcal{G}_s} \sum_{l=i}^{j} t_b^{d,l}(y_d). \tag{8}$$

To efficiently solve the aforementioned constrained optimization problem, we propose an iterative planning algorithm as outlined in Algorithm 1. The algorithm can be divided into two distinct phases: *memory-aware workload balancing phase* and *workload offloading phase*. In the first phase, we recursively distribute the workload among devices in a manner that balances the workload based on their computing capacities, while strictly adhering to the memory budgets (line 1-12). A device's computing capacity v_d is defined as the inverse of the sum of FP and BP execution latency with a micro-batch:

$$v_d = \left(\sum_{l=i}^{j} [t_f^{d,l}(B) + t_b^{d,l}(B)] \right)^{-1}. \tag{9}$$

As previously stated, our experiment reveals the nonlinear relationship between batch size and execution time. Consequently, relying solely on the first phase will lead to suboptimal outcomes. In the second phase, we iteratively transfer the sample workload (one block at a time) from the straggler (the slowest device) to the fastest device with sufficient memory. The iteration terminates when the workload offloading results in a slower straggler (line 13-20). We can adjust the *block size* according to the micro-batch size to trade-off between planning overhead and balancing.

Dynamic Programming HPP Planning. Our dynamic programming algorithm searches for optimal HPP configurations to minimize HPP-Round Latency. To reduce the searching complexity for orchestration across heterogeneous devices, we sort the devices in descending order by their memory capacity and map stages accordingly. This sorting is inspired by the observation in §5.4 that the earlier stages in Asteroid typically require more storage space for activations compared to the later stages.

We consider a DNN model consisting of L layers and aim to leverage the computing resource from an edge resource pool with N edge devices to efficiently perform HPP training.

Algorithm 2: Dynamic Programming HPP Planning

```
for p from 1 to min(L, N) do
    for n from 1 to N do
        for l from 1 to L do
            for n' from 0 to n do
                for l' from 0 to l do
                    Get E_f^s and E_b^s with Alg. 1 and Eq. (8);
                    Update Dominant Step with Eq. (11);
                    Get T_w^s, T_e^s and T_a^s with Eq. (5) and (6);
                    Get HPP-Round Latency with Eq. (4);
            Update Q(l, n, p) with Eq. (10);
```

To achieve this, a novel *dynamic programming algorithm* is devised which facilitates optimal parallelism planning. We denote $Q(l, n, p)$ as the *HPP-Round Latency* of the optimal pipeline planning when slicing the last l consecutive layers into p stages and putting them onto the last n devices. The goal of our algorithm is to calculate $\min_p Q(L, N, p)$. $Q(l, n, p)$ is updated as follows:

$$Q(l, n, p) = \min_{l', n'} \widetilde{Q}(l, n, l', n', p-1), \tag{10}$$

where $\widetilde{Q}(l, n, l', n', p-1)$ represents the *HPP-Round Latency* of a pipeline which comprises two parts (see Fig. 8 as illustration): (1) an optimal sub-pipeline $Q(l', n', p-1)$ consisting of the last l' layers with $p-1$ stages cross the last n' devices. (2) a new single stage (execution step) with layers $L-l$ to $L-l'$ replicated over remaining $n-n'$ devices using DP.

To obtain $\widetilde{Q}(l, n, l', n', p-1)$, its dominant step must first be determined. Finding the step with the fewest bubbles during *Execution Phase* is equivalent to finding the step with the largest total FP and BP execution time after alignment. Consequently, we align and compare the total execution or communication time of the dominant step of sub-pipeline $Q(l', n', p-1)$, the new execution step at the head, and the communication step between the first step and $Q(l', n', p-1)$. The largest one contains the fewest bubbles during *Execution Phase* will be updated as the new dominant step:

$$\max \begin{cases} M \times \left(E_f^{dm^*} + E_b^{dm^*} \right) + \sum_{i=0}^{dm^*+1} \left(E_f^i + E_b^i \right), \\ M \times \left(E_f^{ne} + E_b^{ne} \right), \\ M \times \left(E_f^{nc} + E_b^{nc} \right) + (E_f^{ne} + E_b^{ne}), \end{cases} \tag{11}$$

where dm^* denotes the original index of dominant step in sub-pipeline $Q(l', n', p-1)$, and ne and nc represent the index of the new execution step and new communication step, respectively. After determining the dominant step, we can estimate T_w, T_e and T_a for each step according to Eq. (5) and 6, and then infer the *HPP-Round Latency* by Eq. (4).

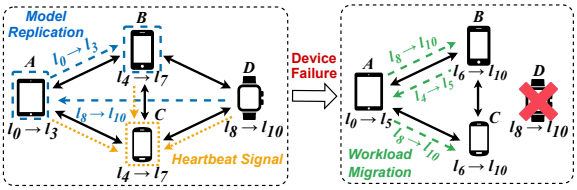


Figure 9: An instance of fault-tolerant pipeline replay.

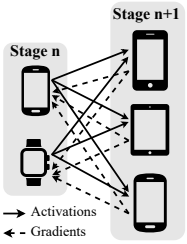


Figure 10: A case of stage replication.

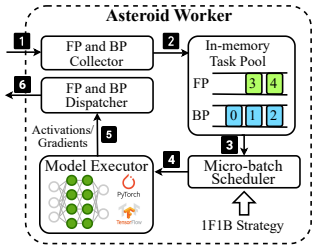


Figure 11: An instance of Asteroid Worker.

We summarize our dynamic programming planning in Algorithm 2. During the execution of algorithm, we concurrently record the parallel configurations of $Q(l, n, p)$, which includes model partitioning points, device grouping strategies, and micro-batch allocations within groups. This process identifies the optimal parallel configurations for HPP upon completion. In §5.7, we demonstrate the efficiency and lightweight nature of our HPP planning, illustrating its suitability for deployment in edge environments.

3.4 Fault-Tolerant Pipeline Replay

Devices at the edge exhibit strong dynamics, as they may leave training at any time, or disconnect due to energy depletion or network anomalies. Such single-device failures can cause the following issues: (1) The device departing can result in the loss of the trained weights. (2) An abnormal device in pipeline can lead to blockages, which necessitate pipeline re-planning. A straw-man proposal is to aggregate stage models to coordinator, rerun the planning algorithm, and redistribute weights based on the new configuration. However, this *heavy rescheduling* may induce considerable latency in the re-planning and model transmission. To tackle these issues, Asteroid incorporates an on-the-fly fault-tolerant pipeline-replay *lightweight rescheduling*, featuring three modules that efficiently respond to resource fluctuations:

1. Heartbeat-guided Failure Detection. As illustrated in Fig. 9(Left), each device periodically emits heartbeat signals to the central coordinator as proof of liveness. The absence of these signals within an expected timeframe may indicate a potential device failure. The coordinator will further dispatch a probe message specifically to the suspected device for confirmation of its operational status.

2. Topology-driven Model Replication. Asteroid adopts a topology-driven model replication to mitigate the loss of trained weights due to device failure. As depicted in Fig. 9(Left), single-device stage (A and D) periodically backs up its stage model to a dedicated device (*backup node*) in the next stage, with the last stage being backed up to the first stage. The model weights can be restored from backup node when device failure occurs. In the presence of device failure in multi-device stages (B and C), model weights can be restored from other surviving devices within the same stage. We note that our topology-driven replication can complement other fault-tolerance mechanisms to address more dynamic scenarios, such as the simultaneous failure of multiple devices.

3. Layer-wise Lightweight Pipeline Re-planning. To expedite the pipeline re-planning, we employ a layer-wise lightweight approach as a substitute for rerunning the entire planning algorithm. We quantify the workload by profiling the FLOPs of all model layers. In the case of a device failure, the associated workload is reallocated among the remaining stages based on their computing capacity ($\sum_{d \in G_s} v_d$). This can be achieved by making a minor adjustment to the layer partitioning points of the current planning configuration. As illustrated in Fig. 9(Right), our mechanism facilitates concurrent layer migration between adjacent stages according to the updated configuration, maximizing bandwidth utilization while minimizing redundant weights transmission.

Putting It All Together. An illustrative example with four devices is presented in Fig. 9. Devices *A*, *B*, and *D* emit periodic heartbeat signals to coordinator *C* to confirm their liveness. Device *A* periodically checkpoints its stage model to backup node *B*, while Device *D* checkpoints its stage model to *A*. When detecting a failure in Device *D*, our lightweight FLOPs-based approach recalibrates the original layer partitioning. Subsequently, all stages concurrently migrate layers based on the new partitioning points. The layer weights initially on failed device *D* can be restored from backup node *A*. After re-planning, a refined two-stage pipeline involving three devices will take over the collaborative model training.

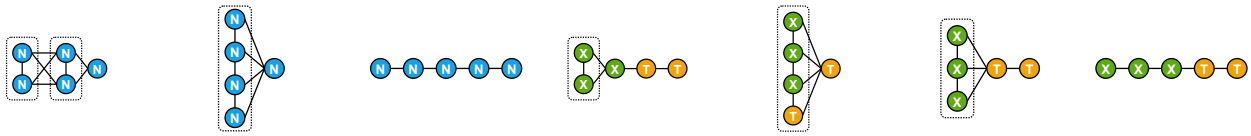
4 IMPLEMENTATION

We have fully implemented a prototype system of Asteroid with ~2,000 LOC in Go and Python in total atop PyTorch [4]. Although we use PyTorch for auto-differentiation and computation graph execution, Asteroid is extensible and can work well with other lightweight ML frameworks such as TF-Lite [6] and MNN [26].

Stage Replication. Fig. 10 illustrates an example where stage n is replicated onto two devices and stage $n+1$ spans three devices. During FP, 1/3 of the activations derived by each device on stage n is sent to each device on stage $n+1$. During BP, each device on stage $n+1$ splits its gradients into two sets and sends back to each device on stage n. We

Table 4: Summary of throughput results comparing Asteroid with on-device training, data parallelism (DP), and pipeline parallelism (PP). The pipeline configuration generated by Asteroid is visualized in Fig. 12. We select the most powerful device in each edge environment as the platform for on-device training.

Task	Model	Dataset	Input Size	Edge Environment	Asteroid Config.	Speedup over Device	Speedup over DP	Speedup over PP
Image Classification	EfficientNet-B1 [49]	Cifar-10 [15]	3 × 32 × 32	A (100Mbps)	❶	4.4×	2.1×	2.8×
				B (100Mbps)	❹	3.0×	4.8×	9.7×
				B (1000Mbps)	❹	3.7×	2.1×	1.4×
	MobileNetV2 [45]	Cifar-10 [15]	3 × 32 × 32	A (100Mbps)	❷	4.5×	1.5×	3.5×
				B (100Mbps)	❺	3.2×	2.3×	11.2×
				B (1000Mbps)	❻	3.8×	1.2×	1.3×
	ResNet50 [20]	Mini-ImageNet [52]	3 × 224 × 224	A (100Mbps)	❷	3.4×	3.6×	5.8×
				B (100Mbps)	❻	1.5×	6.1×	12.2×
				B (1000Mbps)	❹	3.7×	2.9×	3.1×
Language Model	Bert-small [14]	Synthetic Data	32 × 512	A (100Mbps)	❸	3.5×	6.4×	1×
				B (100Mbps)	❼	1.3×	6.8×	1×
				B (1000Mbps)	❼	3.9×	4.2×	1.3×



(a) Configuration ❶. (b) Configuration ❷. (c) Configuration ❸. (d) Configuration ❹. (e) Configuration ❺. (f) Configuration ❻. (g) Configuration ❼.

Figure 12: HPP configurations for the experiments in Table 4. The dashed box indicates a pipeline stage where the devices inside perform data parallelism. "N", "T", and "X" indicate Jetson Nano, TX2, and NX, respectively.

Table 5: Specifications of edge devices in experiments.

Edge Device	GPU Processor	Memory
Jetson Nano [2]	128-core NVIDIA Maxwell	4GB
Jetson TX2 [1]	256-core NVIDIA Pascal	8GB
Jetson NX [3]	384-core NVIDIA Volta	8GB

Table 6: Heterogeneous edge env. used in experiments.

ID	Devices	ID	Devices
A	5 × Nano	C	1 × NX, 2 × TX2, 3 × Nano
B	3 × NX, 2 × TX2	D	1 × TX2, 3 × Nano

ensure that FP and BP of each sample are co-located on one device. We use PyTorch's *Distributed-DataParallel* library [5] to synchronize parameters for data-parallel stages.

Micro-batch FP and BP Scheduling. Each device deploys an *Asteroid Worker* to manage micro-batch FP and BP scheduling, as depicted in Fig. 11. Workers asynchronously receive activations or gradients from the preceding stage (1), packaging them into FP and BP tasks and collecting in an in-memory task pool (2). Micro-batch scheduler fetches tasks for FP/BP execution (3), adhering to a predefined scheduling strategy (e.g., 1F1B) (4). Intermediate tensors derived from FP/BP execution are passed to the dispatcher (5) for asynchronous transmission to the next stage (6).

5 EVALUATION

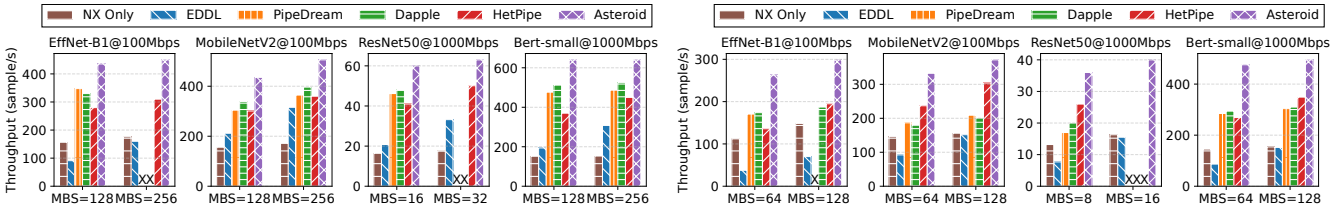
5.1 Experimental Setup

Models and Datasets. We evaluate Asteroid with 4 typical DNN models that are widely used in computer vision (i.e., EfficientNet-B1 [49], MobileNetV2 [45], ResNet-50 [20]) and natural language processing (i.e., Bert-small [14]). We use Cifar-10 [15] dataset with input size 3 × 32 × 32 for both EfficientNet-B1 and MobileNetV2, and Mini-ImageNet [52] dataset with input resized to 3 × 224 × 224. For Bert-small, we build a synthetic dataset with input shape 32 × 512.

Heterogeneous Edge Environment Setup. We use 3 heterogeneous off-the-shelf edge devices listed in Table 5 in our experiments. We analyze our system performance on four edge environments (Table 6) where each is a different combination of heterogeneous edge devices. We consider both 100Mbps and 1000Mbps intra-cluster network bandwidth setting to simulate different network conditions in real edge environments.

Baseline Methods. We implement and compare Asteroid with both the widely-used traditional baselines and the state-of-the-art parallel training methods:

- **Data Parallelism (DP)** [31] is a conventional parallel training method that distributes batch data across cluster devices for concurrent processing.
- **Pipeline Parallelism (PP)** [22] is a conventional parallel training approach that divides the DNN model into sequential stages and processes them in a pipeline manner.
- **EDDL** [19] is a collaborative edge training system that leverages data parallelism on edge device clusters.
- **PipeDream** [39] explores the HPP for asynchronous training in homogeneous accelerator clusters within



(a) Training throughput compared with existing approaches on Env. B.

(b) Training throughput compared with existing approaches on Env. C.

Figure 13: Training throughput comparison under various settings. × means out-of-memory error.

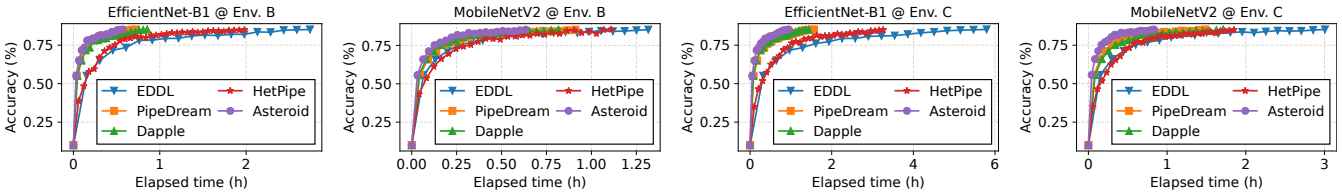


Figure 14: Training convergence of EfficientNet-B1 and MobileNetV2 on Env. B and C compared with baselines.

datacenters. We compare our planning algorithm with PipeDream's under synchronous training scenarios.

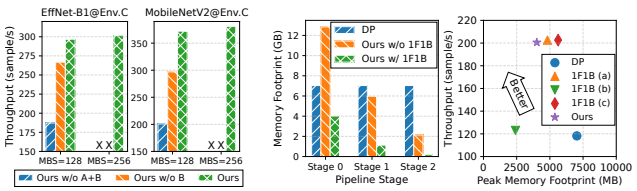
- **Dapple** [16] devises an HPP method dedicated to the synchronous DNN training in large-scale homogeneous accelerators cluster in datacenters.
- **HetPipe** [42] facilitates asynchronous parallel training of large DNN models on heterogeneous GPU clusters by treating sub-groups of GPUs as virtual workers and employing intra-worker PP and inter-worker DP.

5.2 Comparison with DP and PP

Table 4 summarizes the training throughput results comparing Asteroid with on-device, DP and PP training methods. To facilitate a better comparison, we implement heterogeneous workload balancing for both DP and PP, and further employ our 1F1B scheduling for PP. To evaluate Asteroid's performance across clusters with diverse computational capabilities and network bandwidths, we conduct experiments on three different edge environments for each model: Env. A, Env. B and Env. B with 1000Mbps D2D bandwidth. For a fair comparison, all methods for each model share a same global mini-batch size of 2048 for EfficientNet-B1, MobileNetV2, Bert-small and 256 for ResNet50. While prior works have focused on optimizing DP and PP, our evaluation results indicate that Asteroid's HPP is the best for complex edge environment. In CNN-based models, feature map size decreases as layers deepen, and most parameters are in the end's fully connected layers. To reduce inter-stage communication with large activations and minimize AllReduce overhead, Asteroid employs DP in earlier layers and PP in later layers. For Transformer-based language models with huge parameters and small inter-layer activations, Asteroid's planner suggests a straight pipeline. As shown in Table 4, Asteroid achieves a training speedup of 2.1×-6.8× and 1.3×-12.8× compared to DP and PP methods, respectively. Specifically, for ResNet50 on Env. B, Asteroid achieves up to 12.2× faster than PP. We investigate the underlying reasons and find that the inter-stage communication latency between the first and second stage is 24× greater than the execution time of the first stage, which dominates the entire training process. In particular, Asteroid achieves nearly linear scaling and is 3.9× faster than single Jetson NX on the BERT-small model.

5.3 Comparison with Existing Systems

Training Throughput Performance. We compare Asteroid with other state-of-the-art distributed training approaches. To showcase Asteroid's robustness in heterogeneous environments, we conducted experiments for all four models in two edge environments, B and C, each exhibiting different levels of heterogeneity. The results are demonstrated in Fig. 13. Our key observation is that Asteroid consistently and remarkably outperforms other existing parallelism under heterogeneous and resource-constrained edge environments. Specifically, Asteroid surpasses the DP method EDDL, achieving a throughput increase of 1.6×-6.9×. When compared to HPP methods PipeDream and Dapple, Asteroid attains throughput improvements of 1.3×-2.1× and 1.2×-1.8×, respectively. This is because both PipeDream and Dapple are designed for homogeneous accelerator clusters in datacenter, which results in unbalanced and suboptimal workload partition and device grouping for both inter-stage and intra-stage. When compare to HetPipe, Asteroid achieves a throughput increase of 1.2×-1.9×. HetPipe considers device heterogeneity but requires full model gradient exchange after each training iteration, leading to increased D2D communication volume. Also, HetPipe necessitates a centralized parameter server (PS) for asynchronous gradient exchange. Utilizing a bandwidth-limited edge device as a PS can become a bottleneck in the distributed system, and may also disrupt



(a) Ablation study of planning algorithm. A: inter-stage planning. B: intra-stage planning.

(b) Left: ablation study of 1F1B scheduling. Right: memory footprint and throughput results of different 1F1B policies.

Figure 15: Ablation study for each individual optimization technique in §3. × means out-of-memory error.

the communication essential for HPP training in which it participates. This experiment further reveals that our planning algorithm effectively balances training throughput and peak memory footprint. Conversely, PipeDream, Dapple, and HetPipe do not consider device memory budget in their parallelism planning algorithms, leading to OOM errors with their planning configurations.

Training Convergence Performance. We conducted training experiments on the EfficientNet-B1 and MobileNetV2 models using the CIFAR-10 image classification dataset in Env. B and C. We compared the time taken by Asteroid and baselines to achieve a target accuracy of 85%. The results are demonstrated in Fig. 14. Compared to synchronous methods, Asteroid achieves the target accuracy in a similar number of epochs as other baselines. HetPipe's asynchronous updates induce parameter staleness, thereby impairing training accuracy. Consequently, more epochs are required to attain the target accuracy [55, 56]. We observe that Asteroid achieves the target accuracy 1.2×-6.1× faster than all the baselines, attributed to either shorter per-epoch times or fewer required epochs.

5.4 Optimization Implication

This subsection investigates the performance boost of each individual optimization technique introduced in §3.

Asteroid Parallelism Planning. We conduct an ablation study using EfficientNet-B1 and MobileNetV2 on Env. C to assess the contributions of our inter-stage and intra-stage planning to the overall system performance, as depicted in Fig. 15(a). The naive approach without any planning optimization treated all devices as homogeneous and overlooked memory and bandwidth constraints. We observe that our inter-stage planning, which considers both computing heterogeneity between stages and gradient synchronization overhead, substantially boosts training throughput. Our intra-stage planning further enhances throughput by taking into account intra-stage heterogeneity and judiciously managing memory budgets to prevent run-time OOM issues.

1F1B Micro-batch Scheduling. Our analysis of the effectiveness of our 1F1B micro-batch scheduling revealed that, when applied to a 3-stage pipeline composed of three Jetson

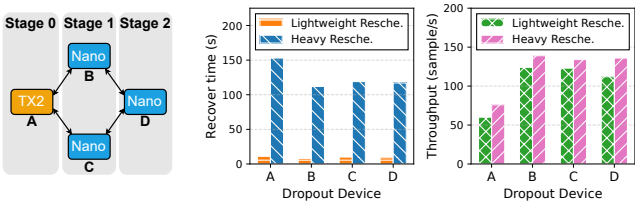


Figure 16: Device grouping and performance of fault-tolerant module across diverse dropout scenarios.

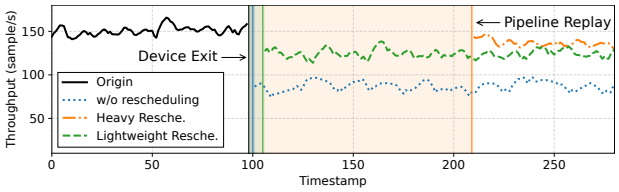


Figure 17: Throughput variation of different scheduling strategies when device B exits the training pipeline.

TX2 and used for training the EfficientNet-B1, our scheduling approach yields a much smaller per-stage memory footprint (estimated by Eq. (3)) compared to DP or conventional *backward after forward* scheduling (Ours w/o 1F1B), as shown in Fig. 15(b)(Left). To verify the superiority of our K_p selection for stage p, we compare four policies: (a): $K_p = 2\times(P-p)$. (b): $K_p = P-p$. (c): $K_p = 2\times(P-p)+1$. (Ours): $K_p = 2\times(P-p)-1$. We can observe from Fig. 15(b)(Right) that our policy is sufficient to achieve a comparable training throughput as Policy a and c, while having the smallest peak memory footprint.

5.5 Fault Tolerance and Pipeine Replay

We evaluate our lightweight fault-tolerant pipeline replay module with EfficientNet-B1 on Env. D, with device orchestration illustrated in Fig. 16(Left). We simulate the individual dropout of four devices, contrasting our lightweight approach with the heavy rescheduling. *Heavy rescheduling* involves aggregating stage models, rerunning the planning algorithm, and redistributing weights according to the new configuration. The planning algorithm is re-executed on the most powerful remaining device. As illustrated in Fig. 16(Mid) and 16(Right), our mechanism recovers significantly faster than the heavy rescheduling and can achieve a comparable system throughput across diverse scenarios. Specifically, Fig. 17 showcases the real-time training throughput over a time window, wherein we deliberately halt device B at the 100th timestamp. We observe that our lightweight mechanism can achieve 90% throughput of heavy rescheduling, while recovering 14× faster.

5.6 Scalability

We analyze the scalability of Asteroid on an 8-node homogeneous Jetson Nano cluster with a fixed micro-batch size of 32 per device (e.g. B=128 for 4 Jetson Nano) and conduct experiments on both EfficientNet-B1 and MobileNetV2 under 100Mbps network bandwidth. Fig. 18 shows that Asteroid's

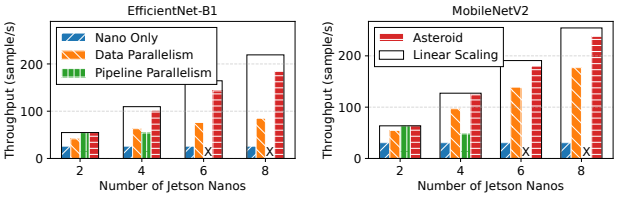


Figure 18: Throughput results with a varying number of Jetson Nanos. × means out-of-memory error.

Table 7: Planning for Env. C across diverse models.

Model	EffNet	MobileNet	ResNet	Bert
Planning time	480sec	261sec	192sec	69sec

Table 8: Total profiling times of four models on devices.

Edge device	Jetson Nano	Jetson TX2	Jetson NX
Profiling time	82min	51min	25min

hybrid parallelism exhibits substantial scalability even under a bandwidth-limited environment, attaining a throughput enhancement of 1.3× - 2.2× in comparison to DP when apply to EfficientNet-B1 and exhibiting near-linear scalability performance with MobileNetV2. Gpipe's partitioning algorithm overlooks the sizes of intermediate tensors at partition points, making inter-stage communication the bottleneck in PP. Consequently, a 4-stage pipeline suffers from lower throughput than its 2-stage counterpart. Moreover, despite employing the 1F1B memory optimization technique, OOM errors arise when scaling up to 6 devices.

5.7 System Overhead

Energy Consumption. We measure the energy consumption during the EfficientNet-B1 training process on Env. D. The experiment shows that Asteroid consumes ~0.13 J per training sample and achieves 2-fold reductions in energy consumption compared to DP. This improvement can be attributed to both the reduced on-device training time and network communication overhead.

Planning Overhead. We evaluate the overhead of Asteroid's parallelism planning on Env. C involving six edge devices. The planning time across diverse models using Jetson NX is detailed in Table 7. Specifically, *Asteroid Planner* takes 480 seconds to partition the 213-layer EfficientNet-B1 optimally, compared to 69 seconds for the 56-layer Bert-small. As the number of model layers increases, the planning time rises significantly. To mitigate this overhead in practical deployment, we can partition models at a coarser granularity (e.g., residual blocks), thereby narrowing the search space.

Profiling Overhead. Profiling is another major overhead in Asteroid. We profile models including EfficientNet-B1, MobileNetV2 and Bert-small with batch sizes ranging from 1 to 256, as well as ResNet50 with batch sizes from 1 to 32. The total profiling times for all models on each edge devices are detailed in Table 8. The profiling overhead can be linearly scaled down by concurrent profiling on more devices.

Properties	EDDL[19]	PipeDream[39]	Dapple[16]	Alpa[64]	HetPipe[42]	Ours
Combining DP with PP?		✓	✓	✓	✓	✓
Res. Hetero. Awareness?	✓				✓	✓
Mem. Constraint Awareness?				✓		✓
Comm. Modeling & Optimization?			✓			✓
Robust to Device-Level Dynamics?	✓					✓
Edge Device Implementation?	✓					✓

Figure 19: Comparing Asteroid with other systems.

Notably, both planning and profiling stages are one-shot offline processes and their outputs can be stored and reused. The overhead of these offline stages can be amortized across thousands of training iterations.

6 RELATED WORK

On-Device DNN Training. POET [43] manages to fine-tune a BERT model on an embedded device in both training and energy efficiency. Lin et al. [33] make on-device training possible with only 256KB of memory. Sage and Melon [17, 53] adopt hybrid memory managing and saving techniques such as operator fusion and dedicated memory pool to address the memory constraints. Mandheling [57] adopt mixed-precision training with DSP offloading for learning acceleration.

Collaborative Edge Computing for DNNs. BlastNet, CoDL and μLayer [24, 30, 34] perform a collaborative DL inference on CPU and GPU concurrently. CoEdge, DeepThings and MoDNN [36, 62, 64] distributed execution of CNN-based inference applications on resource-constrained edge clusters. EDDL [19] adopts DP training across embedded devices.

Parallel DNN Training in Datacenter. DP [18, 31, 44, 47] is the most extensively used distributed training method in datacenter. PP [22, 29] has been proposed to conquer the memory issues of training large-scale transformer-based models. HPP and HDP [16, 25, 35, 39, 40, 42, 50, 65] which combine merits of both DP and PP has been further proposed to address the resource efficiency and scalability. Fig. 19 provides a comparison of Asteroid with other existing works, emphasizing the distinctions between them.

7 CONCLUSION

This paper proposes Asteroid for collaborative DNN training across heterogeneous and resource-constrained edge devices. Asteroid addresses multiple challenges faced in edge environments and achieves 12.2× faster training than traditional methods and 2.1× faster than state-of-the-art HPP methods.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of China (No. U20A20159, No. 62272122); Guangdong Basic and Applied Basic Research Foundation (No. 2021B151520008); Guangzhou Basic and Applied Basic Research Program (No. 2024A04J6367); and the Research Grants Council of Hong Kong under grant GRF14212323.

REFERENCES

[1] 2017. Jetson-TX2. https://developer.nvidia.com/embedded/jetson-tx2.

[2] 2019. Jetson-Nano. https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[3] 2019. Jetson-NX. https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer.

[4] 2019. PyTorch. https://github.com/pytorch/pytorch.

[5] 2019. PyTorch DDP. https://pytorch.org/docs/stable/_modules/torch/nn/parallel/distributed.html.

[6] 2021. On-device training with tensorflow lite. https://www.tensorflow.org/lite/examples/on_device_training/overview.

[7] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 119–135.

[8] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. 176–189.

[9] Dongqi Cai, Yaozong Wu, Shangguang Wang, Felix Xiaozhu Lin, and Mengwei Xu. 2022. Autofednlp: An efficient fednlp framework. *arXiv preprint arXiv:2205.10162* (2022).

[10] Ching-Han Chen, Ming-Yi Lin, and Chung-Chi Liu. 2018. Edge computing gateway of the industrial internet of things using multiple collaborative microcontrollers. *IEEE Network* 32, 1 (2018), 24–32.

[11] Haowei Chen, Liekang Zeng, Shuai Yu, and Xu Chen. 2020. Knowledge distillation for mobile edge computation offloading. *arXiv preprint arXiv:2004.04366* (2020).

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[13] Wentao Chen, Hailong Qiu, Jian Zhuang, Chutong Zhang, Yu Hu, Qing Lu, Tianchen Wang, Yiyu Shi, Meiping Huang, and Xiaowe Xu. 2021. Quantization of Deep Neural Networks for Accurate Edge Computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17, 4 (2021), 1–11.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[15] Krizhevsky et. al. 2009. CIFAR-10. https://www.cs.toronto.edu/~kriz/cifar.html.

[16] 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.

[17] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 464–476.

[18] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[19] Pengzhan Hao and Yifan Zhang. 2021. EDDL: A Distributed Deep Learning System for Resource-limited Edge Computing Environment. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 1–13.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Kai Huang and Wei Gao. 2022. Real-time neural network inference on extremely weak devices: agile offloading with explainable AI. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 200–213.

[22] 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).

[23] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems* 2 (2020), 497–511.

[24] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. Association for Computing Machinery New York, NY, USA, 209–221.

[25] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. 2022. Whale: Efficient Giant Model Training over Heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 673–688.

[26] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. 2020. Mnn: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems* 2 (2020), 1–13.

[27] Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. 2022. Model pruning enables efficient federated learning on edge devices. *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[28] Yuang Jiang, Shiqiang Wang, Victor Valls, Bong Jun Ko, Wei-Han Lee, Kin K Leung, and Leandros Tassiulas. 2022. Model pruning enables efficient federated learning on edge devices. *TNNLS* (2022).

[29] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910* (2020).

[30] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[31] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems* 27 (2014).

[32] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *arXiv preprint arXiv:2206.15472* (2022).

[33] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[34] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2022. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *Proceedings of the Twentieth ACM Conference on Embedded Networked Sensor Systems*. 91–105.

[35] Ziyue Luo, Xiaodong Yi, Guoping Long, Shiqing Fan, Chuan Wu, Jun Yang, and Wei Lin. 2022. Efficient Pipeline Planning for Expedited Distributed DNN Training. *arXiv preprint arXiv:2204.10562* (2022).

[36] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1396–1401.

[37] Yoshitomo Matsubara, Davide Callegaro, Sabur Baidya, Marco Levorato, and Sameer Singh. 2020. Head network distillation: Splitting

distilled deep neural networks for resource-constrained edge computing systems. *IEEE Access* 8 (2020), 212177–212193.

[38] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.

[39] 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.

[40] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 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*. 1–15.

[41] Xiaomin Ouyang, Zhiyuan Xie, Jiayu Zhou, Jianwei Huang, and Guoliang Xing. 2021. Clusterfl: a similarity-aware federated learning system for human activity recognition. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 54–66.

[42] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 307–321.

[43] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *International Conference on Machine Learning*. PMLR, 17573–17583.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[45] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[46] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009), 14–23.

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[48] Xian Shuai, Yulin Shen, Siyang Jiang, Zhihe Zhao, Zhenyu Yan, and Guoliang Xing. 2022. BalanceFL: Addressing class imbalance in long-tail federated learning. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 271–284.

[49] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[50] DeepSpeed Team and Rangan Majumder. 2020. DeepSpeed: Extreme-scale model training for everyone.

[51] Stylianos I Venieris, Christos-Savvas Bouganis, and Nicholas D Lane. 2022. Multi-DNN Accelerators for Next-Generation AI Systems. *arXiv preprint arXiv:2205.09376* (2022).

[52] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. 2016. Matching networks for one shot learning. *Advances in neural information processing systems* 29 (2016).

[53] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the Memory Wall for Resource-Efficient On-Device Machine Learning. (2022).

[54] Yuanxin Wei, Shengyuan Ye, Jiazhi Jiang, Xu Chen, Dan Huang, Jiangsu Du, and Yutong Lu. 2024. Communication-Efficient Model Parallelism for Distributed In-situ Transformer Inference. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[55] Qiong Wu, Xu Chen, Tao Ouyang, Zhi Zhou, Xiaoxi Zhang, Shusen Yang, and Junshan Zhang. 2023. Hiflash: Communication-efficient hierarchical federated learning with adaptive staleness control and heterogeneity-aware client-edge association. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1560–1579.

[56] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2019. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934* (2019).

[57] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 214–227.

[58] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. 2018. Deeptype: On-device deep learning for input personalization service with minimal privacy concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 1–26.

[59] Zirui Xu, Zhao Yang, Jinjun Xiong, Janlei Yang, and Xiang Chen. 2019. Elfish: Resource-aware federated learning on heterogeneous edge devices. *Ratio* 2, r1 (2019), r2.

[60] Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. 2024. Galaxy: A Resource-Efficient Collaborative Edge AI System for In-situ Transformer Inference. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*.

[61] Shengyuan Ye, Liekang Zeng, Qiong Wu, Ke Luo, Qingze Fang, and Xu Chen. 2022. Eco-FL: Adaptive Federated Learning with Efficient Edge Collaborative Pipeline Training. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[62] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608.

[63] Liekang Zeng, Peng Huang, Ke Luo, Xiaoxi Zhang, Zhi Zhou, and Xu Chen. 2022. Fograph: Enabling Real-Time Deep Graph Inference with Fog Computing. In *Proceedings of the ACM Web Conference 2022*. 1774–1784.

[64] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[65] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. *arXiv preprint arXiv:2201.12023* (2022).

[66] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.