IceFrog: A Layer-Elastic Scheduling System for Deep Learning Training in GPU Clusters

Wei Gao, Zhuoyuan Ouyang, Peng Sun, Tianwei Zhang, *Member, IEEE*, and Yonggang Wen, *Fellow, IEEE*

***Abstract*—The high resource demand of deep learning training (DLT) workloads necessitates the design of efficient schedulers. While most existing schedulers expedite DLT workloads by considering GPU sharing and elastic training, they neglect *layer elasticity*, which dynamically freezes certain layers of a network. This technique has been shown to significantly speed up individual workloads. In this paper, we explore how to incorporate *layer elasticity* into DLT scheduler designs to achieve higher cluster-wide efficiency. A key factor that hinders the application of layer elasticity in GPU clusters is the potential loss in model accuracy, making users reluctant to enable layer elasticity for their workloads. It is necessary to have an efficient layer-elastic system, which can well balance training accuracy and speed for layer elasticity. We introduce IceFrog, the first scheduling system that utilizes layer elasticity to improve the efficiency of DLT workloads in GPU clusters. It achieves this goal with superior algorithmic designs and intelligent resource management. In particular, (1) we model the frozen penalty and layer-aware throughput to measure the effective progress metric of layer-elastic workloads. (2) We design a novel scheduler to further improve the efficiency of layer elasticity. We implement and deploy IceFrog in a physical cluster of 48 GPUs. Extensive evaluations and large-scale simulations show that IceFrog reduces average job completion times by 36-48% relative to state-of-the-art DL schedulers.**

***Index Terms*—Distributed systems, deep learning, GPU cluster scheduling.**

I. Introduction

THE proliferation of deep learning (DL) motivates many organizations to set up dedicated GPU clusters to manage numerous DL training (DLT) workloads. These workloads typically demand intensive GPU resources for the long term, leading to high resource oversubscription. This inspires the design of efficient schedulers to mediate resource sharing among DLT workloads and improve resource utilization.

Received 16 August 2024; revised 7 February 2025; accepted 9 March 2025. Date of publication 20 March 2025; date of current version 18 April 2025. The work was supported by the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative. Recommended for acceptance by A. Li. *(Corresponding author: Tianwei Zhang.)*

Wei Gao is with the College of Computational and Data Science, Nanyang Technological University, Singapore 639798, and also with S-Lab, Nanyang Technological University, Singapore 639798 (e-mail: gaow0007@ntu.edu.sg).

Zhuoyuan Ouyang, Tianwei Zhang, and Yonggang Wen are with the College of Computational and Data Science, Nanyang Technological University,, Singapore 639798 (e-mail: ouya0013@ntu.edu.sg; tianwei.zhang@ntu.edu.sg; ygwen@ntu.edu.sg).

Peng Sun is with the Shanghai AI Lab & Sensetime, Shanghai 200232, China (e-mail: sunpeng@pjlab.org.cn).

The code is available at https://zenodo.org/records/14830066.

Digital Object Identifier 10.1109/TPDS.2025.3553137

To this end, many DL schedulers [1], [2], [3], [4], [5], [6], [7] adopt various optimization techniques to accelerate the execution of DLT workloads. Two prominent advanced techniques stand out: (1) *GPU sharing* allows multiple jobs to share the GPU via the NVIDIA MPS or MIG techniques. (2) *Elastic training* dynamically adjusts allocated resources [3], [8] and batch sizes [4], [5] respectively. Here, we explore another promising speedup technique, *layer elasticity*[1] to improve DL scheduling performance. Extensive research efforts [9], [11], [12], [13], [14], [15] have been done to advance the layer elasticity for DLT workloads. These works have demonstrated that by freezing the training of certain **front** layers, the training speed can be improved with limited model accuracy degradation. Therefore, users can incorporate these layer-elastic optimization techniques into their workloads using the resources allocated by schedulers. While this can enhance efficiency, a question we seek to answer is: *can we design a more efficient scheduler to* **further***expedite DLT workloads with the awareness of layer elasticity?* This remains an important but unsolved problem with the following challenges.

First, *the trade-off between training speed and model accuracy has not been fully investigated in existing layer-elastic optimization approaches*. Early techniques [9], [10], [11] necessitate users to manually determine the number of frozen layers throughout the training. The high sensitivity of model accuracy to the number of frozen layers complicates the adjustment. Recent work [15] designs a metric to assess the convergence of each layer and determine the frozen layers on the fly. The computation of this metric requires generating a reference model with quantization techniques, which has significant overhead with multiple feed-forward processes and is error-prone [16]. Moreover, many optimization techniques typically assess the speed-accuracy trade-off by fixing the number of epochs. They ignore that model accuracy of layer-elastic workloads can be restored with more training iterations (Section III-C), thus leading to less optimal balance between training speed and model accuracy.

Second, *existing DL schedulers do not capture the behavior changes of DLT workloads caused by layer elasticity in a system's view*. First, layer elasticity can reduce GPU utilization and memory consumption [9], [10]. Previous DL schedulers [2],

[1] In this paper, layer elasticity refers to considering both resource scaling and layer scaling. Additionally, layer elasticity allows previously frozen layers to be unfrozen, which is different from the concept of layer freezing as described in prior literature [9], [10], [11].

[17], [18] point out that colocating jobs with low GPU utilization and memory consumption can improve cluster-wide efficiency. Such optimization opportunity is never considered in existing GPU sharing enabled schedulers [1], [2], [17]. Second, the effectiveness of elastic schedulers [3], [4], [5], [8], [19] depends on the accurate job throughput modeling for distributed data parallelism (DDP).[2] The introduction of layer elasticity brings a significant throughput improvement due to decreased gradient computation and communication [15]. However, existing elastic schedulers largely neglect such throughput change, leading to less efficient scheduling decisions. Besides, training large models demands various parallelization strategies (e.g., sharded data parallelism (SDP) [20], [21], [22], pipeline parallelism (PP) [23], [24]) to alleviate the GPU memory consumption. The lack of awareness of these parallelization strategies further complicates the job throughput modeling for large model training with layer elasticity. We provide motivational examples in Section II and empirical evaluation in Section V to detail the limitations of existing DL schedulers.

To address the above challenges, we present IceFrog, a layer-elastic scheduler to manage DLT workloads in GPU clusters. First, inspired by goodput in Pollux [4] and plasticity in Egeria [15], we introduce a light-weight metric called *effective progress* to measure the time-to-accuracy (TTA) of a layer-elastic job. Through maximizing *effective progress*, we effectively decide the number of frozen layers to balance the throughput improvement and accuracy loss. More importantly, the computation of this metric only requires gradient features and profiled system features, avoiding the heavy and error-prone computation of the reference model in [15].

Second, we propose a scheduling optimization objective to harness the advantageous aspects of layer elasticity to optimize the cluster-wide performance. This objective delivers a joint resource allocation optimization for elastic training and GPU sharing. For each job, this metric measures the ratio of the actual *effective progress* achieved with the allocated GPUs, to its potential maximum *effective progress* if all available GPU resources are allocated. Through this ratio, IceFrog effectively allocates resources for each job based on their system properties and resource availability to maximize the cluster-wide *effective progress* improvement.

We implement IceFrog as a customized scheduler atop Kubernetes, and evaluate it on a cluster of 12 GPU servers with 48 GPUs. We construct a diverse set of tasks [25], [26], [27], [28], [29], [30] and datasets [31], [32], [33] following a trace pattern in [34]. Compared with state-of-the-art GPU sharing enabled and elastic schedulers, IceFrog reduces the average job completion time (JCT) by up to 48% (Lucid [2]), 46% (Optimus [3]) and 36% (Pollux [4]). Also, we conduct large-scale simulation experiments on a 960-GPU cluster to confirm its scalability. We summarize our contributions as follows:

- We explore and exploit layer elasticity in GPU sharing and resource elasticity as well as large model training with sharded data parallelism, pipeline parallelism, and hybrid parallelism.
- We design a scheduling objective to leverage layer elasticity to automatically optimize layer-elastic configurations and resource allocations for each DLT workload.
- We present and implement IceFrog, a scheduler designed to optimize layer elasticity for DLT workloads, and demonstrate its efficiency through evaluation using representative DLT tasks.

[2]Distributed data parallelism presupposes that the model should be fit into the single GPU. This paper distinguishes between distributed data parallelism and sharded data parallelism [20], [21].

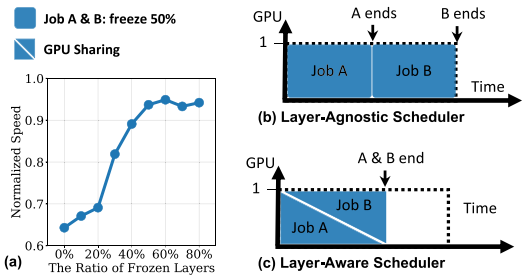


Fig. 1. The impact of layer elasticity on GPU sharing: (a) normalized speed of packing both MobileNetV2 training tasks on a single GPU with batch size 128 over different ratios of frozen layers; (b) layer-agnostic GPU sharing enabled scheduler; (c) layer-aware GPU sharing enabled scheduler.

II. Background and Motivation

We first review the background of layer elasticity. Next, we perform two motivational experiments on V100 GPUs to illustrate how we can exploit the knowledge of layer elasticity to improve the efficiency for GPU sharing enabled and elastic training schedulers.

A. Layer Elasticity

Layer elasticity is an approach to accelerate the DL training via freezing the training of certain front layers during the training progress. Substantial studies [11], [12], [13], [15], [35], [36] highlight layer elasticity as a promising technique to accelerate training jobs with minimal impact on accuracy.

For a DLT job, the adoption of layer elasticity necessitates the consideration of two key aspects. (1) **Job Throughput**: freezing the training of specific layers reduces the computational overhead and eliminates the gradient communication for these layers, thereby enhancing the throughput. (2) **Model Convergence**: while increasing the number of frozen layers can improve the job throughput, it may come at the expense of slow model convergence. Therefore, an efficient metric to quantify model convergence is essential.

B. GPU Sharing

We discuss how a GPU sharing enabled scheduler benefits from knowing a job's remaining time in job colocation scenarios through an example. Fig. 1(a) compares the normalized speed of packing two MobileNetV2 models on a single GPU over different numbers of frozen layers. Increasing the number of frozen layers alleviates the slowdown in speed caused by job colocation on a single GPU.

We denote two identical DL tasks with 50% frozen layers in Fig. 1(a) as job A and job B. Fig. 1(b) and (c) present how a GPU sharing enabled scheduler determines resource allocations for two jobs with a GPU. We also consider layer-agnostic and layer-aware schedulers. In Fig. 1(b), the layer-agnostic scheduler assesses whether both jobs can be packed together according to the profiled GPU utilization and memory consumption when neither job experiences layer freezing. Then, the layer-agnostic scheduler predicts the high interference of packing job A and job B on a single GPU, and decides both jobs should run sequentially. On the contrary, in Fig. 1(c), the layer-aware scheduler knows the normalized speedup caused by layer freezing and intelligently packs job A and job B on the same GPU. Despite job A experiencing a slight speed slowdown, the cluster-wide latency achieves nearly a 1.4× speedup.

Difference between layer elasticity and batch reduction: Reducing the batch size can unlock potential opportunities for GPU sharing, but it does not always lead to a reduction in end-to-end execution time. The reduction of the batch size does not result in a linear improvement in job throughput. For example, training MobileNetV2 on CIFAR10 takes 0.057 seconds using a V100 GPU with a batch size of 32 and 0.074 seconds with a batch size of 64. When two jobs with a batch size of 32 are packed onto a V100 GPU, the interference is negligible. However, sequentially executing two jobs with a batch size of 64 results in a total execution time of 4.81 hours, while colocating two jobs with a batch size of 32 extends the execution time to 4.94 hours. Reducing the batch size increases the number of training iterations, which in turn raises the overhead induced by model parameter access and kernel launch. In contrast, ICEFROG leverages layer freezing to reduce end-to-end execution time. By increasing the number of frozen layers, GPU utilization decreases, creating an opportunity for GPU sharing with negligible interference. Consequently, layer freezing prevents the extended end-to-end execution time that typically results from merely reducing the batch size.

C. Elastic Training

Similar to GPU sharing, we discuss how an elastic scheduler reduces the latency for layer-elastic workloads with the knowledge of their remaining time under given allocated resources. Fig. 2(a) compares the throughput of training ResNet50 [28] on CIFAR10 [31] across different numbers of GPUs without freezing and with 50% frozen layers. Freezing certain layers can mitigate the throughput plateau induced by the heavy communication overhead.

We denote the DL task without freezing as job A and the DL task with 50% frozen layers as job B. In Fig. 2(b) and (c), we investigate how an elastic scheduler minimizes the average JCT for two jobs competing for 24 GPUs. We consider two scenarios: one where layer elasticity is not considered (referred to as the layer-agnostic scheduler), and another where layer elasticity is taken into account (referred to as the layer-aware scheduler). In Fig. 2(b), the layer-agnostic scheduler adopts a layer-agnostic throughput model, predicting the same throughput for both jobs. This leads the layer-agnostic scheduler to allocate an equal

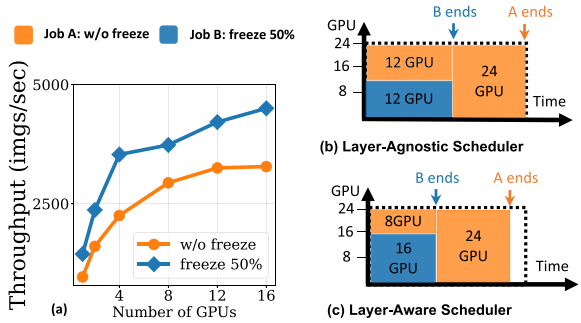


Fig. 2. The impact of layer elasticity on elastic training: (a) job throughput of training ResNet50 on CIFAR10 with batch size 512 over different number of NVIDIA V100 GPUs; (b) layer-agnostic elastic scheduler; (c) layer-aware elastic scheduler.

number of GPUs (12 each) to both jobs. Then, job B is completed faster due to layer freezing, and the remaining GPU resources are allocated to job A. On the other hand, in Fig. 2(c), the layer-aware scheduler knows different remaining times for job A and job B. It proactively allocates more resources to job B, even if job A experiences a slight throughput reduction. As job B is completed sooner than that in Fig. 2(b), job A can get all 24 GPUs earlier than that in Fig. 2(b), and also completes earlier. Thus, the layer-aware scheduler speeds up job A and job B. Overall, the scheduling decision that takes layer elasticity into account can benefit jobs with and without layer freezing.

III. PERFORMANCE MODELING OF LAYER ELASTICITY

In this section, we first illustrate how to model the TTA performance for a layer-elastic job. Next, we introduce the definition of *effective progress*. Last, we explain how *effective progress* is measured for a layer-elastic job. The empirical analysis of *effective progress* is conducted on A800 GPUs.

A. Modeling Time-to-Accuracy

For a layer-elastic workload, we can increase the number of frozen layers to improve its training speed but at the cost of model accuracy. Therefore, a desirable layer-elastic approach is to minimize the TTA. Here, for a workload, we denote its TTA as T^{acc}, and T^{acc} can be measured as follows:

$$T^{\text{acc}} = T^{\text{age}} + \frac{P}{\mathrm{E}}, \tag{1}$$

where T^{age} denotes the elapsed time since the submission of this workload. We denote P as the remaining number of processed samples to reach the target accuracy for this workload without enabling layer elasticity. Practically, P is computed as a product of the maximum training epochs and the number of samples processed in each epoch. E refers to *effective progress* (discussed in Section III-B), indicating the effective processed samples per time unit. Section IV discusses how ICEFROG exploits T^{acc} to allocate resources for layer-elastic workloads.

B. Definition of Effective Progress

We denote *effective progress* as the product of its *layer-aware throughput* and *frozen penalty* at the t-th training iteration:

$$\mathtt{E}_t(a, s, m, \ell) = \psi_t(\ell) \times \mathtt{T}(a, s, m, \ell), \tag{2}$$

where ψ_t, and $\mathtt{T}$ represents *frozen penalty*, and *layer-aware throughput* respectively. Moreover, a denotes the number of allocated GPUs to the job, s is whether to share GPUs with other jobs, m is the per-GPU batch size, and ℓ is the number of frozen layers. The global batch size $M(a, m)$ of this job can be written as $a \times m$, and is fixed for layer-elastic workload. We assume no gradient accumulation when modeling *effective progress* for brevity. We can utilize existing techniques [4], [20] to support the scenario with gradient accumulation.

The *layer-aware throughput* quantifies the number of processed examples per time unit, while the *frozen penalty* assesses the relative progress of processed examples when using layer elasticity compared to training with all layers. The product of *layer-aware throughput* and *frozen penalty* yields a balanced measure of model accuracy and training speed. Below, we explain how to measure *frozen penalty* and *layer-aware throughput* in detail.

C. Definition of Frozen Penalty

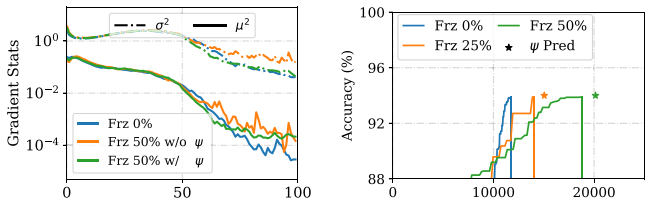
We introduce *frozen penalty* $\psi_t(\ell)$ to facilitate the computation of the additional number of iterations needed to recover the model accuracy as follows:

$$\psi_t(\ell) = \frac{\sigma_t^2[\ell+1:L_0]}{\sigma_{t_0}^2[1:\ell] + \sigma_t^2[\ell+1:L_0]}, \tag{3}$$

where $\sigma_{t_0}^2[1:\ell]$ represents the gradient variance from the first layer to the ℓ-th layer at the t_0-th training iteration when layer freezing is applied. $\sigma_t^2[\ell+1:L]$ denotes the gradient variance from the $(\ell+1)$-th layer to the final L-th layer at the current t-th training iteration. We use $\psi_t(\ell)$ to compute how many additional iterations needed when freezing the first ℓ layers to attain similar model convergence when training all layers.

Intuitive Explanation of (3): Eqn. (3) computes how many additional training iterations needed to recover the model accuracy. At t-step, ICEFROG desires to recover the gradient variance of ℓ frozen layers at frozen step t_0 by running $\frac{1}{\psi_t(\ell)} - 1$ additional iterations. At the t-step, the DL job produces the gradient variance $\sigma_t^2[1:L_0]$. If the DL job runs extra $\frac{\sigma_{t_0}^2[1:\ell]}{\sigma_t^2[1:L_0]}$ iterations, it is assumed that the gradient variance in each additional iteration is approximated as $\sigma_t^2[1:L_0]$. Consequently, the accumulated gradient variance is $\sigma_{t_0}^2[1:\ell]$, which compensates for the gradient variance loss of ℓ frozen layers at t_0-step. Given that the first ℓ layers are frozen at t-step and have zero gradient variance, we have $\sigma_t^2[1:L_0] = \sigma_t^2[\ell+1:L_0]$. This leads to the relationship $\frac{1}{\psi_t(\ell)} - 1 = \frac{\sigma_{t_0}^2[1:\ell]}{\sigma_t^2[\ell+1:L_0]}$, which is the basis for (3).

How (3) restores the model accuracy: To understand how *frozen penalty* contributes to restoring model accuracy, we examine the gradient statistics and model accuracy of a concrete layer-elastic workload. First, we uncover how *frozen penalty* recovers the gradient statistics. In Fig. 3(a), we present the gradient variance and square of freezing 0% layers (blue line) and freezing 50% layers from the 50_{th} epochs onwards (orange line) for ResNet18 on CIFAR10. We observe a significant discrepancy of gradient variance and square between the settings of freezing 50% layers and training all layers. We incorporate ψ into the training task with 50% layers frozen (green line), and the epoch (x-axis) is scaled by ψ. We compensate the training iterations using ψ instantaneously in each epoch. As such, the number of training examples in each epoch differs between green and orange lines. For example, the orange line experiences 196 iterations at 51_{th} epoch, while the green line experiences 204 iterations. Hence, with ψ, we observe that the gradient variance of the training task with 50% layers frozen can perfectly match that of training all layers. Also, ψ reduces the difference of gradient square between the settings of freezing 50% layers and training all layers. This demonstrates that ψ can recover the gradient statistics of a DL task with certain layers frozen. Similar phenomena are observed in other DL tasks as well.



(a) Epoch versus Gradient Stats. (b) Iteration versus Accuracy.

Fig. 3. Statistical information of training ResNet18 on CIFAR10 with batch size 256 on a single GPU. (a) The gradient variance σ^2 and square μ^2 (y-axis) of freezing 0%, freezing 50% layers from 50_{th} epochs w/ and w/o *frozen penalty* ψ change over the epochs. (b) The validation accuracy (y-axis) of freezing 0%, 25% and 50% layers from 50_{th} epochs onwards vary over the training iterations (x-axis) and the number of iterations to accuracy predicted by *frozen penalty*.

Second, we investigate whether *frozen penalty* can predict how many additional iterations are needed to reach the target accuracy. We set the target of top 1 validation accuracy of training ResNet18 as 94%. In Fig. 3(b), we compare the iteration versus validation accuracy of ResNet18 with different portions of frozen layers from the 50_{th} epoch onwards. Our observation is that the number of training iterations required for the task with all layers training (blue line) is not sufficient for the tasks with 25% (orange line) and 50% (green line) frozen layers to achieve the same target accuracy. We use ψ to predict how many training iterations to reach the target accuracy for tasks with certain layers frozen (star marker). We observe that their accuracy can be reached before the number of training iterations predicted by ψ. This indicates ψ is an appropriate metric to predict how many additional training iterations are needed to recover the model accuracy.

We also conduct a qualitative analysis for ψ. When the gradient variance of certain front layers converges to a small value, we can safeguard the increase of frozen layers. When the gradient variance σ_t^2 of the first ℓ layers converges to close to zero, $\psi_t(\ell)$ is close to 1. Freezing the first ℓ layers does not sacrifice the model convergence significantly but improves the throughput. The *effective progress* E can achieve the overall improvements by

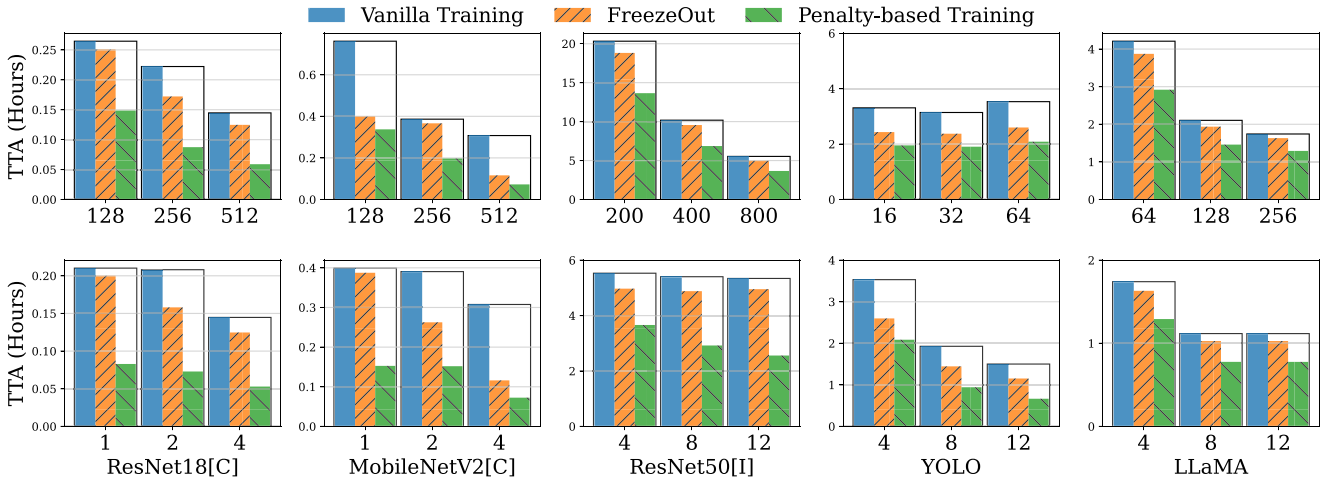


Fig. 4. Time-to-accuracy (TTA) performance of different training methods for common DL tasks. (First row): TTA (y-axis) between the scenarios of vanilla training, FreezeOut as well as our proposed penalty-based training over different batch sizes (x-axis). We evaluate them on a 4-GPU A800 node. (Second row): TTA (y-axis) between three training methods over different numbers of allocated GPUs. We fix the batch size as 512 for CIFAR tasks, 800 for ImageNet tasks, 64 for YOLO, and 256 for LLaMA-3B. [C] and [I] denote CIFAR10 and ImageNet datasets.

freezing these ℓ layers. As the training goes on, $\sigma^2_{t_0}[1:\ell]$ keeps fixed, while $\sigma^2_t[\ell+1:L_0]$ gradually decreases. This decreases $\psi_t(\ell)$, and prompts to unfreeze some layers to maximize E.

Empirical validation: We provide a comprehensive empirical analysis about *frozen penalty* in Fig. 4. For simplicity, we use vanilla training to represent training all layers. Penalty-based training refers to using *frozen penalty* to determine the number of frozen layers by maximizing (2) in given allocated resources and global batch size. Considering the complexity [15] and strong coupling of other freeze training implementations with transformer-based models [9], [10], we have opted for FreezeOut [11] as a freeze training baseline. We strengthen FreezeOut as a competitive baseline by optimizing hyperparameters that control the number of frozen layers during training and reporting the best TTA results.

Fig. 4 (First row) compares TTA results of vanilla training, FreezeOut [11], and penalty-based training with different batch sizes in fixed resource allocations. The target accuracy for different tasks can be found in Table VI. FreezeOut can decrease TTA compared to vanilla training in most cases. However, it cannot compete with penalty-based training in that it fails to trade off the model convergence and training speedup. Fig. 4 (Second row) presents TTA results of different training methods with a fixed batch size in a variety of resource allocations. When we increase the number of allocated GPUs, penalty-based training presents a much better performance in TTA compared to the other two baselines. Overall, penalty-based training aims to maximize (2), thus minimizing the TTA. This suggests that we can use *frozen penalty* to better configure the number of frozen layers. Note that with the same training epoch (scaled by *frozen penalty*), penalty-based training can bring an average of (1%) improvement in accuracy for CIFAR10 tasks. Other DL tasks (e.g., ResNet18, ResNet50, MobileNetV2 on ImageNet, and YOLO on PASCAL-VOC) almost suffer no performance loss.

TABLE I
SUMMARY OF NOTATIONS IN SECTION III-D

Sym.	Definition
T_{fwd}	The forward activation computation overhead
T_{bwd}	The backward gradient computation overhead
T_{sync}	The gradient synchronization overhead
γ	Learnable parameter to capture the overlap between T_{bwd} and T_{sync}
α_{flop}, β_{flop}, γ_{flop}	Learnable parameter to capture the impact of layer elasticity on T_{bwd}
θ_ℓ	The FLOPs ratio of non-frozen layers to total layers
α_{sync} , β_{sync}	Fitting parameters to capture the impact of layer elasticity on T_{sync}
ω_ℓ	The ratio of the sizes of unfrozen parameters to total parameters

D. Modeling Layer-Aware Throughput

Next, we compute the layer-aware throughput $\mathtt{T}(a,s,m,\ell)$ for distributed data parallelism (DDP). We summarize relevant notations in this section in Table I. For the ease of understanding, we also summarize the key insight for several equations discussed in this section. Following prior works [3], [4], [5], [8], we define the job throughput as the number of processed samples per time unit. Here, it is computed by dividing the global batch size M by the time cost per iteration T_{iter}, and then multiplying it by the slowdown factor $\lambda(s,m,\ell)$ caused by GPU sharing. Mathematically, it can be expressed as:

$$\mathtt{T}(a,s,m,\ell) = M(a,m)/T_{\text{iter}} \times \lambda(s,m,\ell). \tag{4}$$

Modeling T_{iter}: We initially consider throughput modeling without the slowdown brought by GPU sharing. A typical way to model T_{iter} is to explicitly decompose it into the forward activation computation overhead T_{fwd}, the backward gradient computation overhead T_{bwd}, and the gradient synchronization overhead T_{sync}. Existing schedulers [4], [5], [8], [37] adopt similar decomposition solutions. Particularly, introducing layer elasticity would primarily influence the backward gradient computation overhead T_{bwd} and gradient synchronization overhead T_{sync}. Without considering layer elasticity, cluster users have many techniques [3], [4], [5] to model $T_{\text{bwd}}(m,\ell=0)$ under any local batch sizes, and $T_{\text{sync}}(a,\ell=0)$ under any resource

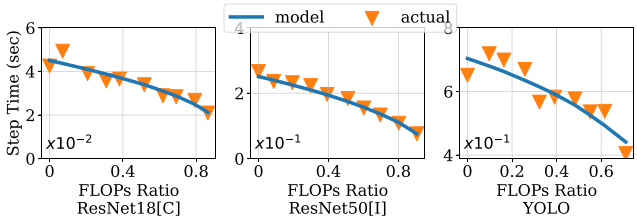
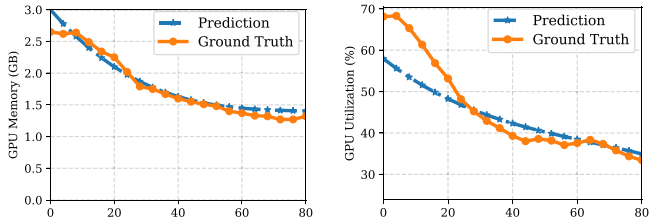


Fig. 5. The time cost per iteration (seconds) versus the FLOPs ratio of frozen layers for different DL tasks. The result is the average of 20 iterations to filter out the system noise on one A800 GPU. We fix m as 200 (ImageNet), 256 (CIFAR10) and 16 (YOLO) for different DL tasks.



(a) GPU Memory. (b) GPU Utilization.

Fig. 6. Prediction of GPU memory consumption (a) and utilization (b) over different numbers of frozen layers (x-axis) on MobleNetV2[C] with batch size 128.

allocations. Inspired by [4], [38], the throughput modeling for distributed data parallel jobs is formulated as

$$T_{\text{iter}}(a,m,\ell) = T_{\text{fwd}}(m) + \left(T^{\gamma}_{\text{bwd}}(m,\ell=0) + T^{\gamma}_{\text{sync}}(a,\ell=0)\right)^{1/\gamma}, \quad (5)$$

where $\gamma \geq 1$ is a learnable parameter to capture the overlap between T_{bwd} and T_{sync}. The layer elasticity brings minor impact on T_{fwd}. Next, we discuss how to model T_{bwd} and T_{sync} for layer elasticity.

Key Insight of (5): The throughput of a DLT job can be decomposed into three stages: forward activation computation, backward gradient computation, and gradient synchronization. Note that there exists an overlap between the last two stages. In conventional DLT workloads, layer elasticity primarily impacts the efficiency of these two stages.

First, we consider modeling $T_{\text{bwd}}(m,\ell)$ when $T_{\text{bwd}}(m,\ell=0)$ is known. Increasing ℓ will reduce the number of floating point operations (FLOPs) in the backward pass. Therefore, instead of scaling T_{bwd} with ℓ, we model the relationship between the FLOPs of frozen layers and T_{bwd}. We denote the FLOPs ratio of non-frozen layers to total layers as θ_ℓ. Fig. 5 shows that T_{bwd} changes with the FLOPs ratio of frozen layers (i.e., $1-\theta_\ell$). We observe a sub-linear scaling for ResNet18, ResNet50, and YOLO. This relationship needs to be considered in addition to θ_ℓ. Thus, T_{bwd} is formulated as

$$T_{\text{bwd}}(m,\ell) = \alpha_{\text{flop}} + \beta_{\text{flop}} \cdot \theta_\ell^{\gamma_{\text{flop}}} \cdot T_{\text{bwd}}(m,0), \quad (6)$$

where $\alpha_{\text{flop}}, \beta_{\text{flop}}, \gamma_{\text{flop}}$ are learnable parameters: α_{flop} models the kernel launch overhead; γ_{flop} fits the sub-linear scaling trend between θ_ℓ and T_{bwd}. When $\gamma_{\text{flop}} < 1$, $T_{\text{bwd}}(m,\ell)$ drops sub-linearly with the decrease of θ_l.

Key Insight of (6): The time cost of the backward gradient computation decreases sublinearly with the FLOPs ratio of the frozen layers.

Second, modeling T_{sync} necessitates consideration of the impacts of both the resource allocation a and the number of frozen layers ℓ. Typically, T_{sync} correlates linearly with the size of the gradients or parameters for communication, which can be computed via ℓ. We denote as ω_ℓ the ratio of the sizes of unfrozen parameters to total parameters when freezing the first ℓ layers. Then, we have

$$T_{\text{sync}}(a,\ell) = \alpha_{\text{sync}} + \beta_{\text{sync}} \cdot \omega_\ell \cdot T_{\text{sync}}(a,0). \quad (7)$$

When we fix ℓ (i.e., ω_ℓ), α_{sync} and β_{sync} are fitting parameters to model the launching and communication overheads of gradient synchronization. The time cost of gradient synchronization typically scales linearly with ω_ℓ [4], [37]. Hence, we utilize the product of the regression coefficient β_{sync} and ω_ℓ to represent the linear correlation between T_{sync} and the ratio of the sizes of unfrozen parameters to total parameters.

Key Insight of (7): The time cost of the backward gradient computation scales linearly with the proportion of unfrozen parameters relative to the total parameters.

Modeling $\lambda(s,m,\ell)$*:* We consider the scenario of packing multiple jobs on a single GPU. We leverage profiled features, including GPU utilization and GPU memory, to identify combinations of jobs that are not likely to suffer serious interference. Inspired from [2], [17], DL jobs primarily contend for GPU compute and memory resources. Therefore, we adhere to two rules to determine whether multiple jobs can be packed together: (1) The accumulative usage of GPU memory does not exceed the GPU memory to avoid out-of-memory issues; (2) The accumulative GPU utilization does not surpass 100%. Note that, we will evict packed jobs based on submission time if profiled GPU utilization is unstable during packing.

The next step is to estimate the GPU utilization and maximum memory usage for a job. We employ a linear regression model, using m, θ_ℓ and ω_ℓ as inputs, to estimate the GPU utilization and memory usage for a layer-elastic job. For reference, we will denote the parameters of the learnable linear regression model as θ_{util} and θ_{mry}. Fig. 6 illustrates the comparison between ground truth and prediction over different numbers of frozen layers when training MobileNetV2 on CIFAR10. When GPU memory consumption and utilization are high, inaccurate predictions still suggest that such jobs are unsuitable for GPU sharing. Conversely, in situations of low memory consumption and utilization, prediction errors do not adversely impact GPU sharing

TABLE II
R^2 SCORES FOR MEMORY AND UTILIZATION PREDICTION

DL Task	R^2	
	GPU Memory	GPU Utilization
ResNet18[C]	0.94	0.86
MobileNetV2[C]	0.90	0.94
ResNet50[C]	0.74	0.90
VGG19[C]	0.70	0.74
GoogLeNet[C]	0.85	0.95

performance either. The linear regression model demonstrates excellent accuracy when GPU utilization is around 50%.

Given that GPU sharing is applicable to relatively small tasks, we present the R^2 scores for a limited number of DL tasks in Table II. The higher R^2 (close to 1) indicates a more accurate prediction performance. Our adopted linear regression performs fairly well in estimating GPU memory consumption and utilization. Furthermore, we multiply a factor of 1.1 by the GPU memory prediction results to avoid potential failures induced by the estimation error of GPU memory.

With the predicted results for GPU memory and utilization, we apply the aforementioned rule to determine whether the two jobs are suitable for packing. Instead of precisely modeling the job slowdown caused by GPU sharing across different numbers of frozen layers, we treat this as a classification problem and formulate λ as follows:

$$\lambda(s,m,\ell) = \begin{cases} 0 & \text{if } s = 0 \\ 0 & \text{if } (m,\ell) \text{ not satisfy rules (1) and (2)} \\ 0.9 & \text{otherwise.} \end{cases} \quad (8)$$

When a job is classified as insensitive to GPU sharing, we empirically assign a decay factor of 0.9 to λ, otherwise, we assign 0 to λ. Although the actual slowdown factor varies across different job-packing pairs when the job is insensitive to GPU sharing, adopting a constant empirical value of 0.9 is motivated by three key reasons. First, accurately predicting the slowdown factor is inherently challenging and necessitates additional profiling resources and time. By setting a constant value, we aim to harness the potential benefits of GPU sharing without introducing excessive complexity. This strikes a balance between modeling complexity and practical effectiveness. Second, the primary purpose of the slowdown factor is to quickly and easily classify jobs as sensitive or insensitive to GPU sharing. Setting an appropriate constant value facilitates the effective co-location of jobs on the same GPU. Furthermore, when observed slowdowns are unsatisfactory, we can promptly disable GPU sharing to prevent further performance degradation, ensuring robust performance under dynamic conditions. Third, while setting a fixed value for the slowdown factor may constrain optimization, it still achieves desirable scheduling performance (as illustrated in Fig. 10(d) in Section V-D). We further investigate the effect of varying the constant value of the slowdown factor in Table IX. Our analysis reveals that the optimal scheduling efficiency is achieved when the slowdown factor is set to 0.9. Also, adjusting the slowdown factor does not result in substantial performance variance, underscoring the robustness of our approach.

TABLE III
PREDICTION ERROR OF LAYER-AWARE THROUGHPUT MODEL

Dataset	Model	APE	
		mean (%) ↓	max (%) ↓
CIFAR10	ResNet18	3.2	7.1
	VGG19	2.2	4.4
	MobileNetV2	3.0	10.1
	GoogLeNet	2.9	7.2
	ResNet50	2.2	6.3
ImageNet	ResNet18	2.8	8.6
	MobileNetV2	3.4	8.7
	ResNet50	2.2	5.5
PASCAL-VOC	YOLO	3.2	17.7
Naruto-Caption	DDPM	4.2	12.3
WikiText	BERT	2.2	6.1

TABLE IV
SUMMARY OF NOTATIONS IN SECTION III-E

Sym.	Definition
$T_{\text{bwd}}^{\text{grad}}$	The backward gradient computation of SDP
$T_{\text{bwd}}^{\text{ag}}$	`All-gather` sharded model parameters during backward propagation
γ_{lm}	Modeling the overlap between the backward gradient computation and the `All-gather` operation
p	The number of pipeline stages
T_{fwd}^{j}	The computation overhead of forward activation at stage j
T_{bwd}^{j}	The computation overhead of backward gradient at stage j
T_{act}^{j}	The activation transmission overhead from stage j and $j+1$
T_{grad}^{j}	The gradient transmission overhead from stage $j+1$ and j
d	The number of pipeline replicas
$\mathcal{F}(\ell, j)$	A function to compute how many layers are frozen in the stage j

Empirical validation: Table III presents the mean/maximum absolute percentage error (APE) of our proposed *layer-aware throughput* across different models, batch sizes, and resource allocations including GPU sharing. All models attain at most 10% APE, with the exception of YOLO and DDPM, which exhibit errors of up to 12.3% and 17.7%, respectively. Interestingly, YOLO and DDPM still benefits from our throughput model in the evaluation because our proposed *layer-aware throughput* can effectively capture the change of throughput with the number of frozen layers.

E. Layer-Aware Throughput for Large-Scale Parallelization Techniques

DL developers usually choose sharded data parallelization, pipeline parallelization, and hybrid parallelization strategies to realize large model training. Note that large-scale model training saturates GPU utilization even with most layers frozen, leaving no opportunity for GPU sharing. Next, we discuss the layer-aware throughput for these parallelization techniques without GPU sharing. We summarize relevant notations in this section in Table IV.

Sharded Data Parallelization: SDP [21] is to reduce the GPU memory consumption by partitioning the model weights, gradients and optimizer states across GPUs. PyTorch [39] provides seamless integration of SDP with no significant code modifications, thereby fostering wide adoption in large model training.

We use the same equation with (5) to model T_{iter} for SDP. The difference between DDP and SDP lies in the overhead modeling of forward activation computation and backward gradient computation. In the forward pass, SDP overlaps the forward

activation computation and `all-gather` operations to collect the sharded model parameters across GPUs. The slight reduction in GPU memory allocation overhead for activations of frozen layers caused by increasing frozen layers, taking only tens of milliseconds, is negligible compared to T_{iter}, so we disregard this impact.

In the backward pass, SDP performs the `all-gather` operations on model weights and backward gradients. T_{bwd} is influenced by the resource allocation a and the local batch size m. In consideration of the SDP's implementation in PyTorch [22] support the overlap between the `all-gather` operations and computational operations. Thus, the backward gradient computation overhead is expressed as:

$$T_{\text{bwd}}(a,m,0)=\left(T_{\text{bwd}}^{\text{grad}}(m,0)^{\gamma_{\text{lm}}}+T_{\text{bwd}}^{\text{para}}(a,0)^{\gamma_{\text{lm}}}\right)^{\frac{1}{\gamma_{\text{lm}}}}, \quad (9)$$

where $T_{\text{bwd}}^{\text{grad}}$ and $T_{\text{bwd}}^{\text{para}}$ indicate the backward gradient computation and the `all-gather` operations to collect the sharded model parameters. γ_{lm} indicates the overlapping between the backward gradient computation and the `all-gather` operations. The layer freezing skips the backward gradient computation and all-gather operations for frozen layers.

Key Insight of (9): In the context of SDP, the time cost of the backward gradient computation can be divided into two components: the gradient computation operation and the `all-gather` operations. Without the layer elasticity, the time cost of these two components relates to the local batch size and allocated GPUs respectively. Note that the overhead of these two components can be overlapped to enhance efficiency.

Eqn. (6) and (7) inspire to model the backward gradient computation with the number of frozen layers ℓ as

$$T_{\text{bwd}}(a,m,\ell)=\Big[\Big(\alpha_{\text{flop}}+\beta_{\text{flop}}\cdot\theta_{\ell}^{\gamma_{\text{flop}}}\cdot T_{\text{bwd}}^{\text{grad}}(m,0)\Big)^{\gamma_{\text{lm}}} + \big(\alpha_{\text{para}}+\beta_{\text{para}}\cdot\omega_{\ell}\cdot T_{\text{bwd}}^{\text{para}}(a,0)\big)^{\gamma_{\text{lm}}}\Big]^{\frac{1}{\gamma_{\text{lm}}}}, \quad (10)$$

where α_{flop} and α_{para} models the overhead of launching compute kernels and communication primitives (e.g., NCCL [40]), respectively. β_{flop} and β_{para} are learnable parameters to model the gradient computation and all-gather parameter communication.

Key Insight of (10): With the layer elasticity, the computational operations decrease sublinearly with the FLOPs ratio of the frozen layers. The `all-gather` operations scale linearly with the fraction of unfrozen parameters compared to the total parameter count.

Pipeline Parallelization: An alternative strategy for large model training is pipeline parallelism (PP) [23]. This parallelization strategy partitions a model into p pipeline stages distributed across GPUs. In the forward pass, stage j transmits the activation of its last layer to the first layer of its successor stage $j+1$. Conversely, in the backward pass, the last layer of stage j receives the gradient from the first layer of stage $j+1$.

Thus, without layer freezing, we use AMP's formula [41] to model the time cost per iteration for a pipeline parallel job T_{pp} as

$$T_{\text{pp}}(p,m,\ell=0)=\underbrace{\sum_{j=0}^{p-1}\Big(T_{\text{fwd}}^{j}(m)+T_{\text{act}}^{j}(m)\Big)}_{\text{forward pass}} + \underbrace{\sum_{j=0}^{p-1}\Big(T_{\text{bwd}}^{j}(m,\ell=0)+T_{\text{grad}}^{j}(m)\Big)}_{\text{backward pass}}, \quad (11)$$

where T_{fwd}^{j} and T_{bwd}^{j} is denoted as the computation overhead of forward activation and backward gradient at stage j, respectively. T_{act}^{j} and T_{grad}^{j} is denoted as the activation transmission overhead from stage j and $j+1$ and the gradient transmission overhead from stage $j+1$ and j, respectively. Note that T_{act}^{p-1} and T_{grad}^{p-1} are both set as zero because the last stage does not receive gradients from other stages and does not transmit any activations to other stages.

Key Insight of (11): In pipeline parallelism, the time cost per iteration includes forward computation, activation transmission, backward computation, and gradient transmission. Notably, they are affected by the local batch size. Additionally, the overhead of backward computation also depends on the number of frozen layers.

The performance modeling for PP in AMP [41] furnishes prior knowledge of T_{fwd}^{j}, T_{bwd}^{j}, T_{act}^{j} and T_{grad}^{j}. Layer freezing reduces the backward gradient computation and circumvents gradient transmission of certain stages. With the number of frozen layers ℓ, T_{pp} is formulated as

$$T_{\text{pp}}(p,m,\ell)=\underbrace{\sum_{j=0}^{p-1}\Big(T_{\text{fwd}}^{j}(m)+T_{\text{act}}^{j}(m)\Big)}_{\text{forward pass}} + \underbrace{\sum_{j=0}^{p-1}\Big(T_{\text{bwd}}^{j}(m,\mathcal{F}(\ell,j))+\mathbb{1}(\mathcal{F}(\ell,j+1)=0)\cdot T_{\text{grad}}^{j}(m)\Big)}_{\text{backward pass}}, \quad (12)$$

where we introduce a function $\mathcal{F}(\ell,j)$ to compute how many layers are frozen in the stage j and $\mathbb{1}$ is the indicator function. We follow the (6) to model the backward gradient computation with the number of frozen layers ℓ. Moreover, when there are frozen layers in stage $j+1$, no gradient transmission between stage j and $j+1$.

TABLE V
PREDICTION ERROR OF LAYER-AWARE THROUGHPUT MODEL FOR TRAINING LLAMA-3B AND LLAMA-7B UNDER VARIOUS PARALLELIZATION STRATEGIES

Model	Parallelization	APE	
		mean (%) ↓	max (%) ↓
LLaMA-3B	SDP	4.1	11.9
	PP	1.4	4.3
	HP	4.7	8.9
LLaMA-7B	SDP	4.4	13.4
	PP	3.4	6.1
	HP	4.7	9.9

Key Insight of (12): Layer freezing only affects the time cost of the backward pass. We just account for the overhead of the backward pass for unfrozen layers.

In the scenario where only PP is applied, there is a constraint where the number of allocated GPUs equals the number of pipeline stages (i.e., $T_{\text{iter}}(a, m, \ell) = T_{\text{pp}}(p, m, \ell)$).

Hybrid Parallelization: Hybrid parallelism (HP) is a hybrid of distributed data and pipeline parallelism for large model training. Given the number of allocated GPUs a, T_{iter} depends upon the number of pipeline replicas d and the number of pipeline stages p per replica (i.e., $a = d \times p$). We model the gradient synchronization overhead between pipeline replicas T_{sync} as

$$T_{\text{sync}}(p, d, \ell) = \sum_{j=0}^{p-1} T^{j}_{\text{sync}}(d, \mathcal{F}(\ell, j)). \tag{13}$$

Key Insight of (13): Layer freezing eliminates gradient synchronization for the frozen layers, thus we only consider the gradient synchronization overhead of the unfrozen layers.

To account for the overlap between gradient computation and gradient synchronization, we adopt the same technique in (5) to model this interaction.

Empirical Validation: Table V shows the mean/maximum absolute percentage error of various parallelization strategies for training LLaMA-3B [42] on SQuAD V2 dataset [43] and LLaMA-7B on SST2 dataset [44]. Considering the GPU memory constraint, the allocation unit is configured as four GPUs. The maximal estimation error of our designed throughput model for various parallelization strategies is 13.4% error rate. We validate that the dedicated throughput model for these parallelization strategies is more effective than that for DDP in Section V-D.

IV. IceFrog System Design

Fig. 7 illustrates the workflow of IceFrog. It has two key components: `Model Trainer` and `Cluster Scheduler`. Each user submits a DLT workload and the instantiation of `Model Trainer`, which specifies the ranges of allocated GPUs, ranges of the number of frozen layers (❶).

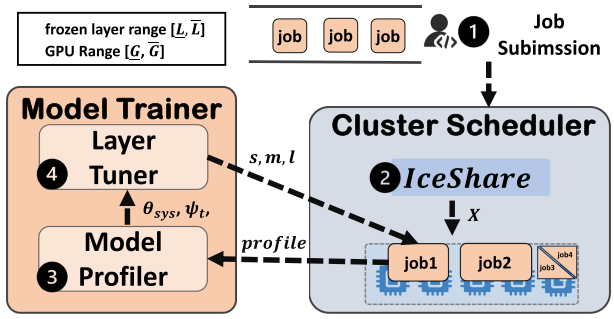


Fig. 7. The workflow of IceFrog. It consists of two key components: (1) `Model Trainer` aims to collect the profiled features and determines the layer-elastic hyper-parameters; (2) `Cluster Scheduler` utilizes profiled features to decide resource allocations.

In each scheduling interval, IceFrog determines the allocated resources of each workload by optimizing the scheduling objective `IceShare` in (18) (❷). During training, the *model profiler* in `Model Trainer` profiles and reports the job run-time, gradient statistics, GPU memory, and utilization to *layer tuner* (❸). The *layer tuner* determines the training configurations (e.g., ℓ, m) of DLT jobs to maximize *effective progress* (❹). Below we detail the mechanisms of `Model Trainer` and `Cluster Scheduler`.

A. Model Trainer

`Model Trainer` acts as an interface to automate layer freezing, and it consists of the following two components.

Model Profiler: This component collects the run-time, gradient statistics, GPU memory consumption, and GPU utilization of DLT jobs. Such information is required to accurately model the parameters θ_{sys} of *effective progress* $\mathtt{E}$:

$$\theta_{\text{sys}} = (\alpha_{\text{flop}}, \beta_{\text{flop}}, \gamma_{\text{flop}}, \alpha_{\text{sync}}, \beta_{\text{sync}}, \alpha_{\text{bwd}}, \beta_{\text{para}}, \gamma, \theta_{\text{util}}, \theta_{\text{mry}}). \tag{14}$$

Specifically, we randomly initialize θ_{sys} at the beginning. Collecting sufficient profiling information requires the DLT workload to span many different GPU resource allocations and numbers of frozen layers. During this process, *layer-aware throughput* $\mathtt{T}$ eventually becomes accurate.

Layer Tuner: This component determines the training configurations based on profiling information to minimize the TTA. For given resource allocations, *layer tuner* maximizes the *effective progress* to identify the most efficient per-GPU batch size and the number of frozen layers:

$$(m^*, \ \ell^*) = \arg\max_{m,\ell} \ \mathtt{E}(a, s, m, \ell). \tag{15}$$

We need to optimize (15) when two events happen: (1) During resource re-allocations, the resource allocation a is changed. (2) At the beginning of each epoch, `Model Trainer` will re-evaluate the *frozen penalty*, and compute how many additional iterations are needed to reach the model convergence. Alternatively, users can provide a plugin algorithm to decide the number of frozen layers [9], [10], [15].

B. Cluster Scheduler

TTA estimation: We have detailed modeling T^{acc} in Section III-A with any resource allocations and numbers of frozen layers. Considering that we have a set of N jobs $J = \{j_1, j_2, \ldots, j_N\}$ and M GPUs in the cluster. With the *layer tuner*, we can predict job j_k's TTA under the resource allocation a as follows:

$$T_k^{\text{acc}}(a) = T_k^{\text{age}} + \frac{P_k}{\max_{m,\ell} \mathtt{E}(a, s, m, \ell)}, \tag{16}$$

where T_k^{age} is the elapsed time since job j_k is submitted, P_k refers to the remaining number of processed samples to complete job j_k.

Optimization objective: The scheduling objective of IceFrog is to improve the job efficiency. We directly use the number of requested GPUs to represent different resource allocations for simplicity and flexibility. To quantify the benefit brought by layer elasticity, we define `IceShare` as follows:

$$\mathtt{IceShare}_k(a) = \frac{\min_{a' \in A_k} T_k^{\text{acc}}(a')}{T_k^{\text{acc}}(a)}, \tag{17}$$

where A_k is a subset of $\mathbb{N}$ indicating the range of allocated GPUs for job j_k. Note that we only consider GPU sharing for jobs that accept single GPU training. In addition, we do not distinguish the resource topology to optimize the scheduling objective. `IceShare` measures the slowdown caused by the allocated GPUs a compared to the maximum allowed GPUs. Higher `IceShare` means that this job enjoys more benefits from allocated resources. To enforce that each job shares a similar `IceShare`, we aim to optimize the scheduling objective as follows:

$$\arg\max_{\mathbf{X}} \min_{j_k \in J} \left[w_k \cdot \sum_{a \in A_k} x_{k,a} \cdot \mathtt{IceShare}_k(a) \right]. \tag{18}$$

Let $\mathbf{X}$ represent a binary matrix with $|J|$ rows and $(M+2)$ columns, representing the resource allocation for each job. The additional two columns denote scenarios where no GPUs are allocated and where resources are allocated with GPU sharing. Each binary element $x_{k,a}$ in $\mathbf{X}$ indicates whether the resource allocation of job j_k is a. w_k is a weight to quantify the importance of job j_k, and we set w_k as the same value for all workloads in our evaluation. We enforce the following constraints when optimizing (18): (1) the sum of allocated resources cannot exceed the cluster capacity; (2) each job should be assigned one resource allocation (zero resource allocation means no allocated resources). IceFrog uses the Integer Linear Programming (ILP) solver to yield an optimal solution to (18). Then, we adopt the same placement policy in [45] to satisfy the resource request.

Scheduling scalability: `Cluster Scheduler` optimizes (18) using the ILP solver, incurring the search time up to 11.5 seconds in a 48-GPU cluster. The search cost will increase with higher job loads and cluster capacity. To improve the scalability, our system provides three dedicated designs: (1) We cache the solution in the previous round to speed up the optimization in this round; (2) We only consider the number of allocated resources instead of resource topology to reduce the space for resource allocation. Additionally, we only allocate the entire GPU nodes for jobs that request a large number of GPUs to further reduce the search overhead. (3) We partition the cluster and jobs into several disjoint parts and execute the ILP solver for different partitions in a parallel manner. This enables IceFrog to scale up the cluster capacity by 20 $\times$ and achieve comparable performance to the ideal solution.

V. Evaluation

We perform both physical (Section V-B and V-D) and simulation (Section V-E) experiments to validate the superiority of IceFrog.

A. Model Zoo and Workload

We present a full set of DL tasks in Table VI. Each DL task contains the dataset, model name, range of allocated GPUs, range of batch sizes, size, target validation metric, the fraction of jobs, speedup gain by *layer tuner*, and tuned FreezeOut respectively. In particular, tuned FreezeOut refers to that we use *effective progress* to tune the hyper-parameters of FreezeOut [11] based on the GPU request and batch size scale. We sample the batch size from a 2-exponential distribution within the specified range. For the target validation metric, we set an appropriate value that can be achieved by the DL tasks by training all layers and freezing certain layers. For size, we use a similar technique in [4] to catalog each DL task as **S**(mall). **M**(edium), **L**(arge) and **XL**(arge) based on its GPU time. The fraction of jobs indicates the fraction of each DL task in our evaluation trace. We also report the speedup gain brought by our *layer tuner* and tuned FreezeOut over different sizes and allocated GPUs.

Workload construction: We randomly sample 120 jobs from hour 3 to hour 6 in Philly trace [34], and denote this job load as $1\times$. To construct $W\times$ jobload, we sample $120 \times W$ jobs. Each sample in the Philly trace only provides the submission time, the number of GPUs, and the duration. The former two attributes can be directly used to construct our workload. We assign DL tasks for each workload based on the product of job duration and GPU requests. The assigning process is to choose one DL task in Table VI to match the size of such a job.

B. Physical Experiments and Evaluation

Implementation: For `Model Trainer`, we implement all the modules upon PyTorch 2.4 to adapt to layer-elastic training. For `Cluster Scheduler`, we set the scheduling interval as 300 seconds to balance the scheduling performance and overhead (as discussed in Section V-E). In the end-to-end experiment, we set a fixed maximum number of frozen layers for each DL task, specifically at 90% of the total number of layers.

Cluster testbed: We set up our physical experiments in a cluster consisting of 12 GPU nodes. Each node is equipped with 4 A800-80 GB GPUs,[3] $1\times$200 Gbps HDR InfiniBand, 64 CPU cores, and 256 GB memory, connected via PCIe 3.0. We

[3]Indeed, each node consists of 8 GPUs, and we selected IDs 0 to 3 for our evaluation to increase the number of GPU nodes. This approach allows more jobs to undergo multi-node communication.

TABLE VI
THE DL TASK SPECIFICATION IN WORKLOAD CONSTRUCTION

Dataset	Model	GPU Range	Batch Size Range	Size	Validation	Frac	TTA Speedup by	
							Layer Tuner	Tuned FreezeOut
CIFAR10 [31]	ResNet18 [28]	1-8	32-512	S	94% top1 acc.	24%	1.2-2.1×	1.0-1.7×
	VGG16 [29]	1-8	32-512	S	92% top1 acc.	24%	1.1-1.4×	1.0-1.3×
	MobileNetV2 [27]	1-8	32-512	S	93% top1 acc.	6%	1.1-3.7×	1.0-1.9×
	GoogleNet [30]	1-32	32-4096	M	94% top1 acc.	4%	1.2-1.8×	1.0-1.5×
	ResNet50 [28]	1-32	32-4096	M	94% top1 acc.	4%	1.2-1.8×	1.1-1.5×
ImageNet [33]	ResNet-18 [28]	4-32	200-6400	L	70% top1 acc.	2%	1.2-1.5×	1.1-1.2×
	MobileNetV2 [27]	4-32	200-6400	L	69% top1 acc.	2%	1.1-1.5×	1.1-1.3×
	ResNet-50 [28]	4-32	200-6400	XL	75% top1 acc.	1%	1.1-1.6×	1.0-1.3×
VOC [32]	YOLO [26]	4-32	8-256	L	73% mAP	2%	1.2-1.6×	1.1-1.6×
SQuAD [43]	LLaMA-3B(SDP) [42]	4-32	128-512	L	0.85 F1-score	1%	1.4-1.8×	1.1-1.3×
	LLaMA-3B(HP) [42]	4-32	128-512	L	0.85 F1-score	1%	1.4-1.7×	1.1-1.2×
SST2 [44]	LLaMA-7B(SDP) [42]	4-32	128-512	XL	96% top1 acc.	1%	1.2-1.9×	1.1-1.3×
	LLaMA-7B(HP) [42]	4-32	128-512	XL	96% top1 acc.	1%	1.3-1.7×	1.1-1.2×
Naruto-Caption [46]	DDPM [47]	1-32	16-256	M	20 clip score	4%	1.3-1.9×	1.1-1.4×
WikiText [48]	BERT [25]	1-8	32-1024	S	79 val perplexity	24%	1.1-1.9×	1.1-1.5×

TABLE VII
END-TO-END RESULTS OF PHYSICAL EXPERIMENTS

Policy	Average JCT (Hours)			
	All	Small	Medium	Large
Optimus	0.93	0.74	0.71	2.39
+ Tuned FreezeOut	0.79	0.56	0.63	2.45
Optimus	0.88	0.6	0.63	2.92
+ Tuned FreezeOut	0.81	0.52	0.63	2.88
+ Tuned FreezeOut + GPU Sharing	0.78	0.47	0.67	2.89
Pollux	0.72	0.5	0.4	2.52
+ Tuned FreezeOut	0.67	0.41	0.45	2.61
ICEFROG	**0.48**	**0.26**	**0.22**	**2.19**

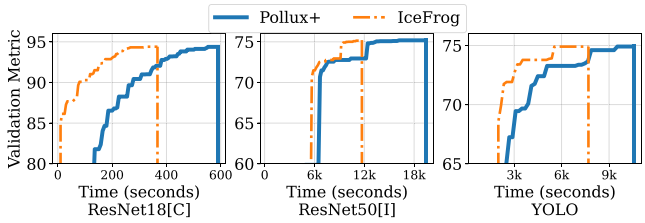


Fig. 8. Time-to-accuracy results in physical experiments for different DL tasks.

deploy Kubernetes 1.18.2 along with CephFS 14.2.8 to store checkpoints. We use the aforementioned workload construction to synthesize a trace containing 120 jobs submitted within the first 4 hours. Considering the expensive cost of physical evaluation, only three traces are adopted.

Baselines: We primarily compare ICEFROG with three efficient schedulers: Lucid [2], Optimus [3] and Pollux [4]. Lucid is a GPU sharing-enabled scheduler without considering elastic training. Lucid, Optimus, and ICEFROG employ a fixed global batch size, while Pollux dynamically configures the batch size for each workload. We adopt FreezeOut [11] to enhance our baselines. The incorporation of FreezeOut reinforces that ICEFROG is a more efficient scheduler than directly integrating layer elasticity into existing schedulers.

Moreover, we enhance Optimus with GPU sharing as follows: we utilize our GPU sharing prediction method to select potential jobs to pack and allocate resources for the remaining jobs based on available resources. Because Pollux employs batch size scaling to increase GPU utilization, it does not offer opportunities to pack jobs together.

End-to-end evaluation results: Table VII presents the average JCT of all, small, medium, and (extremely) large jobs for different systems. Section V-A describes how to categorize jobs based on their sizes. Overall, ICEFROG can bring 1.94×, 1.84×, and 1.51× improvement over Lucid, Optimus, and Pollux respectively. Besides, we observe that tuned FreezeOut can boost the performance of workloads with different sizes. The JCT speedup for Lucid, Optimus, and Pollux brought by FreezeOut is 1.17×, 1.08×, and 1.07×. Layer elasticity can expedite individual workloads by up to 1.9×, but these baselines cannot adapt hyper-parameters of layer-elastic workloads dynamically to the resource allocations and batch sizes. Hence, the overall JCT speedup brought by layer elasticity is limited. This indicates that without a dedicated scheduler design, existing schedulers fail to fully utilize the potential of layer-elastic optimization techniques to enhance cluster efficiency.

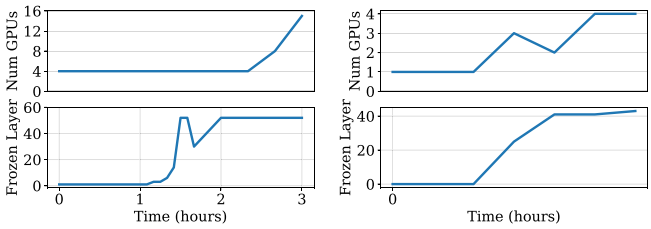
Additionally, GPU sharing does not yield a significant speedup for Optimus. We reinforce Optimus with sharing via packing jobs first and then allocating resources elastically for remaining jobs. However, the jobs that are packed together experience only a modest 1.03× speedup.

In our evaluation, ICEFROG takes average 3.6 (maximal 8.4) seconds per scheduling interval to optimize `IceShare`. Fig. 8 presents the time-to-accuracy results of the same workload managed by both Pollux+Tuned FreezeOut and ICEFROG. ICEFROG improves the TTA over different DL tasks compared to Pollux+Tuned FreezeOut by 1.2 - 1.8×.

A closer look: We provide a closer look at the dynamical variations of the number of frozen layers and resource allocations for layer-elastic workloads. We present the number of GPUs (**first row**) and the number of frozen layers (**second row**) of YOLO and MobileNetV2 throughout the training course in Fig. 9(a) and (b) respectively. Both jobs are from our physical experimental trace. We observe the DL model gradually increases the number of frozen layers during training. In the later stage of layer-elastic training, the training configurations tend to be stable.

C. Simulation Configurations

For the evaluation in Section V-E, we build a simulator to analyze the key designs of ICEFROG including large-scale scenarios.



(a) YOLO on VOC. (b) MobileNetV2 on CIFAR10.

Fig. 9. The configurations of layer-elastic workloads vary with time in the physical evaluation of IceFrog.

TABLE VIII
Comparison Between Physical and Simulator W.R.T. The Speedup of IceFrog Against Different Policies

Policy	physical	sim-30	sim-60	sim-90
IceFrog vs. Lucid	1.94×	1.99×	1.91×	1.73×
IceFrog vs. Optimus	1.84 ×	1.50×	1.89×	2.20×
IceFrog vs. Pollux	1.51 ×	1.34×	1.57×	1.68×

Simulator construction: For training without GPU sharing, we measure GPU memory, GPU utilization, T_{comp}, and T_{sync} for different GPU allocations with a maximal allocation of 48 GPUs. We evaluate the range of batch sizes and frozen layers without exceeding the GPU memory limit. We adopt linear interpolation to estimate the GPU memory, GPU utilization, and throughput for unseen configurations. For GPU sharing, we measure the actual job throughput for job pairs between small tasks to reduce the overhead of simulator construction.

Simulating gradient statistical information under different numbers of frozen layers and bath sizes requires extensive experiments. Following Pollux's simulator construction, we freeze $25\%, 50\%, 75\%$ of layers starting from $25\%, 50\%, 75\%$ of the training progress, i.e., each batch size has a total of 10 variants of gradient statistics. We also evaluate the variance and squared norm of the gradient of each layer in each epoch across different batch sizes in Table VI with a base-2 exponential increasing order. Similar to throughput estimation, we utilize linear interpolation between the nearest batch size and the number of frozen layers to simulate the gradient statistics of a job given a certain batch size, frozen layers, and training epoch.

Simulator fidelity: The overhead of saving and resuming DLT jobs is an important factor of the simulation fidelity. To minimize the performance gap between the simulator and physical experiment, we adjust the re-allocation overhead to 30, 60, and 90 seconds, and present the JCT speedup (y-axis) of IceFrog compared to Lucid, Optimus, and Pollux in Table VIII. Specifically, `physical` indicates the physical experiment in Section V-B, and `sim-S` indicates setting the resource re-allocation overhead as `S` seconds for the simulator. When we vary the re-allocation overhead, the speedup performance gap between the physical and simulation experiments is minimal when we change the re-allocation overhead to 60 seconds. Therefore, we select re-allocation overhead 60 seconds in our simulation.

D. Impact of Effective Progress

We conduct a detailed physical empirical analysis of our proposed *effective progress*.

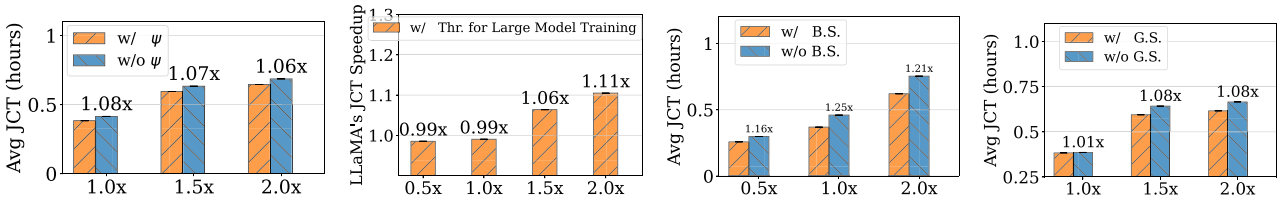
Impact of frozen penalty: The *frozen penalty* approximates how many additional training iterations are needed to reach model convergence. We consider replacing *frozen penalty* with tuned FreezeOut and *layer-aware throughput*, and compare the scheduling performance of IceFrog and IceFrog + FreezeOut in Fig. 10(a). Obviously, *frozen penalty* plays a key role in the scheduling design. It accelerates the JCT around 1.1 × over job loads compared to tuned FreezeOut. *Frozen penalty* performs better in characterizing the model convergence and determining frozen layers.

Impact of layer-aware throughput for large model training: We have developed a layer-aware throughput model for parallelization techniques adopted by large model training. To investigate its effectiveness, we replace the throughput model for SDP and HP with the throughput model for DDP. We focus on the JCT speedup for LLaMA across varying job loads in Fig. 10(b). The benefits are more pronounced under high job loads, exceeding 1.1×. The high resource contention requires accurate performance modeling to make efficient scheduling decisions. Thus, large model training enjoys more benefits from a dedicated throughput model for large model training in high job loads.

Extension to batch size scaling: Our proposed *effective progress* naturally aligns with the batch size scaling technique adopted in [4]. We incorporate batch size scaling into IceFrog and report the JCT speedup achieved by batch size scaling across varying job loads in Fig. 10(c). Overall, IceFrog can leverage batch size scaling to improve cluster-wide latency by a factor of 1.16 to 1.21. This suggests that layer freezing serves as an orthogonal acceleration technique, complementary to batch size scaling.

Impact of GPU sharing: GPU sharing can reduce queuing delay, however, its effectiveness hinges upon the job load. Fig. 10(d) shows the JCT between GPU sharing enabled and disabled scheduler respectively. The speedup gain of GPU sharing is more desirable with a high job load. GPU sharing allows packing jobs into a single GPU to free more GPUs for other jobs, thus attaining cluster-wide latency speedup. GPU sharing can improve cluster-wide efficiency and particularly excels in handling resource contention.

Impact of slowdown factor: We examine the impact of the slowdown factor of (8). For jobs classified as insensitive to GPU sharing, we assign a constant value to the slowdown factor. Particularly, we choose a 2 × job load to conduct empirical analysis, as GPU sharing under this load demonstrates significant JCT speedup benefits. We report the average JCT as the slowdown factor ranges from 0.6 to 1.0 in Table IX. Our findings indicate that setting the slowdown factor to 0.9 yields optimal JCT performance. Overall, the performance variance remains minimal. Although the constant value does not precisely capture runtime slowdown, it provides valuable guidance to the scheduler by indicating which jobs can be packed together on a single GPU without significantly degrading performance. As a



(a) Frozen Penalty. (b) Thr. for Large-scale Parall. (c) Batch Size Scaling. (d) JCT.

Fig. 10. [Physical] The impact of *effective progress* : (a) The impact of frozen penalty across varying job loads; (b) The layer-aware throughput analysis for large-scale parallelization techniques; (c) The improvement of batch size scaling; (d) The impact of GPU sharing on the JCT performance across varying job loads.

TABLE IX
THE IMPACT OF SLOWDOWN FACTOR

$\lambda(s,m,\ell)$	0.6	0.7	0.8	0.9	1.0
Avg JCT (hours)	0.803	0.801	0.795	0.791	0.792
$\lambda(s,m,\ell)$	0.86	0.88	0.90	0.92	0.94
Avg JCT (hours)	0.794	0.792	0.791	0.792	0.792

result, ICEFROG tends to favor packing jobs onto a single GPU when resource contention is high.

E. Impact of Scheduler Design

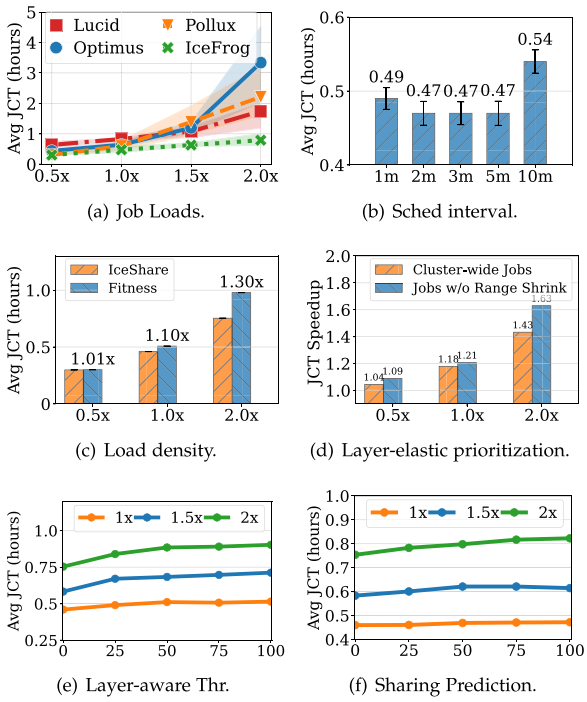
We adopt our simulator to perform a detailed analysis of our scheduling designs.

Sensitivity to the job load. We compare the agerage JCT (y-axis) of ICEFROG, Lucid+, Optimus+, and Pollux+ for different job loads (x-axis) in Fig. 11(a). The symbol '+' denotes the incorporation of tuned FreezeOut. Increasing the job load causes higher resource contention and a larger average JCT. The speedup brought by ICEFROG is more significant when the job load is high. Overall, ICEFROG outperforms Lucid+, Optimus+, and Pollux+ by $1.71-2.20\times$, $1.34-4.21\times$, $1.10-2.80\times$ speedup across different job loads.

Scheduling interval: We vary the scheduling interval from 1 minute to 10 minutes. Fig. 11(b) shows the impact of the scheduling interval (x-axis) on the average JCT (y-axis) using $1\times$ load. A smaller interval results in increased context switch overhead, while a larger interval sacrifices scheduling flexibility. The optimal performance is achieved at a 5-minute interval. Considering the potential overhead from the ILP solver, we select a 5-minute interval for ICEFROG with a minor scheduling performance drop.

Impact of optimization objective: Pollux [4] proposes `Fitness` to enforce the instantaneous fairness of cluster-wide jobs, which implicitly improves the efficiency of elastic scheduling. We design `IceShare` to directly enhance the cluster efficiency. Fig. 11(c) presents the JCT of `IceShare` and `Fitness` over different job loads. The superiority of `IceShare` becomes more pronounced as job loads increase. Under high job loads, `IceShare` tends to pack jobs and accommodate more jobs to complete earlier.

Layer-elastic prioritization: We investigate the benefits of optimizing `IceShare` to prioritize layer-elastic with larger frozen

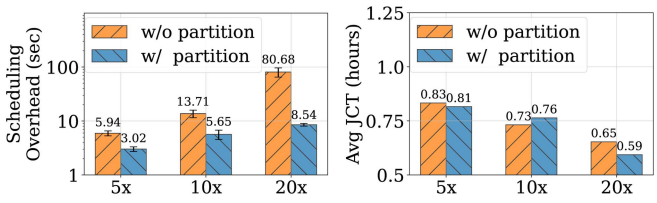


(a) Job Loads. (b) Sched interval. (c) Load density. (d) Layer-elastic prioritization. (e) Layer-aware Thr. (f) Sharing Prediction.

Fig. 11. [Simulation] The impact of key scheduling designs: (a) Simulation results of different scheduling policies across varying job loads; (b) The impact of scheduling interval on the scheduling performance; (c) The impact of the scheduling objective on the JCT performance; (d) The impact of layer-elastic prioritization; (e) The error analysis of layer-aware throughput across varying job loads; (f) The error analysis of GPU sharing prediction.

layers, thus motivating more users to submit layer-elastic workloads. Specifically, we explore the effects of shrinking the range of the number of frozen layers for half workloads and compare the JCT speedup between optimizing `IceShare` and `Fitness`. Our study focuses on how the JCT speedup (y-axis) varies in the job load (x-axis) for all workloads and workloads without range shrink. Fig. 11(d) suggests that jobs without range shrink experience higher JCT speedup compared to cluster-wide jobs. As a result, more users tend to set a larger range for frozen layers.

Sensitivity to layer-aware throughput: Layer-aware throughput is to predict the job throughput under different resource allocations and numbers of frozen layers. We explore the sensitivity of ICEFROG to *layer-aware throughput*. Specifically, we add the Gaussian noise to the prediction results of *layer-aware throughput*, and display the average JCT in Fig. 11(e) over different



(a) Scheduling overhead. (b) Scheduling performance.

Fig. 12. [Simulation] Scalability of IceFrog: (a) The scheduling overhead across varying job loads; (b) The average JCT performance across varying job loads.

degrees of added noise. For the $2\times$ job load, the estimation error results in 17% JCT increase when the added Gaussian noise is up to 50%. Overall, the large estimation error does not significantly alter the effectiveness of resource allocation decisions.

Sensitivity to sharing prediction: To uncover the impact of GPU memory consumption and utilization estimation on scheduling performance, we add the Gaussian noise to the prediction results of the linear regression, and present the average JCT in Fig. 11(f). IceFrog performs relatively stable over varying degrees of the estimation error of GPU memory consumption and utilization across job loads. The average JCT is only increased slightly by at most 1.09 when the estimation error reaches 100%. Additionally, our evaluation shows that only two jobs are typically packed together on the same GPU device, as packing three jobs would result in GPU utilization exceeding 100%. This aligns with prior studies [2], [49]. Overall, our adopted simple rules to determine the slowdown factor enhance the robustness of IceFrog to GPU sharing estimation error. Therefore, incorrect predictions do not cause excessively poor scheduling decisions.

Large-scale simulation: The large scheduling overhead makes it inefficient to make prompt decisions about resource allocations in large-scale clusters. IceFrog solves this issue by reducing the optimization variables and partitioning the optimization variables into several parts. We scale the job load and cluster capacity by $5\times$, $10\times$, and $20\times$. Due to the simulation time cost, each experimental result is obtained from a single workload rather than the average of multiple workloads. We compare the performance of the scheduling overhead (y-axis) and JCT (y-axis, log-scaled) with and without using the partition mechanism proposed by [50] over different cluster capacities (x-axis) in Fig. 12(a) and (b). With a larger job load and cluster capacity, the scheduling overhead of `IceShare` increases to hundreds of seconds but the partition mechanism can still enforce the average scheduling overhead within 10 seconds, only accounting for 3% of scheduling interval. Moreover, the intensive search space makes partition-based optimization outperform another one when IceFrog makes scheduling decisions on a $20\times$ job load and cluster capacity.

VI. Related Works

DLT schedulers: Various scheduling systems are designed to improve the execution of DLT workloads in GPU clusters [1], [37], [38], [51], [52], [53], [54]. Optimus [3], AFS [8], and Ymir [6], [7] are resource-elastic schedulers to maximize the cluster-wide job throughput. Pollux [4] and ONES [5] are batch-elastic schedulers that adapt training configurations (e.g., batch size, learning rate) to resource allocations for higher average JCT. IceFrog is extended from Pollux [4] with a new dimension, *layer elasticity*, to further optimize the job latency. Sia [55] is a heterogeneity-aware, batch-elastic scheduler designed to outperform Pollux in heterogeneous GPU clusters. Its scheduling policy, tailored for heterogeneous resources, can be integrated into IceFrog to enhance its capability to manage resource heterogeneity effectively.

Layer elasticity. Early works [12], [13] propose to linearly freeze some layers along with the training progress without compromising model accuracy. However, they lack a reliable way to recover the potential accuracy loss when setting the inappropriate number of frozen layers. FreezeOut [11] reduces the gradient computation by progressively reducing the learning rate of these layers to zero. However, FreezeOut includes a hyperparameter to control the extent of layer freezing, which limits users from achieving optimal TTA performance. LayerOut [56] utilizes gradient statistical information to determine which layers to freeze. However, it does not restrict freezing from the first layer, resulting in negligible improvements in latency efficiency. IntelligentFreeze [35] introduces a formula to compute the normalized gradient difference but lacks a mechanism to resume training for frozen layers. To support transformer-based models, some works [9], [10] dedicate lightweight freezing policies to stop the gradient computation of compute-expensive transformer layers. Egeria [15] and its variant [36] can guarantee statistical efficiency and unfreeze some layers to recover the model performance. However, they require the online model quantization to determine the number of frozen layers. The model quantization algorithms for many DLT tasks [26], [42] are complex, and the execution overhead of quantized models on CPUs is significant, which hampers the efficiency of layer elasticity. Overall, these techniques only focus on speeding up individual workloads and lack explicit modeling for layer-elastic training.

VII. Conclusion and Future Work

This paper presents IceFrog, a scheduling system that exploits layer elasticity to improve the efficiency of DLT workloads in GPU clusters. We propose *effective progress* to balance the throughput and model accuracy of layer-elastic jobs. We devise `IceShare` to directly maximize cluster-wide *effective progress*. Our extensive experiments show that IceFrog has better latency efficiency than existing schedulers. The large-scale simulation demonstrates its scalability.

We consider the following directions as future work. (1) This paper primarily focuses on evaluating homogeneous GPU clusters. We can extend IceFrog to realize cluster scheduling with diverse GPU types and networking links. (2) While this paper mainly evaluates sharded data parallelism and pipeline parallelism, future work can incorporate additional parallelism strategies including expert parallelism, sequence parallelism, and tensor parallelism. Both directions require an accurate

throughput predictor to enable effective scheduling decisions. To achieve this, we can construct an offline performance model and integrate online profiling information to deliver precise throughput predictions.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 595–610.

[2] Q. Hu, M. Zhang, P. Sun, Y. Wen, and T. Zhang, "Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 457–472.

[3] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–14.

[4] A. Qiao et al., "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2021, Art. no. 1.

[5] Z. Bian, S. Li, W. Wang, and Y. You, "Online evolutionary batch size orchestration for scheduling deep learning workloads in GPU clusters," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–13.

[6] W. Gao, W. Zhuang, M. Li, P. Sun, Y. Wen, and T. Zhang, "Ymir: A scheduler for foundation model fine-tuning workloads in datacenters," in *Proc. 38th ACM Int. Conf. Supercomput.*, 2024, pp. 259–271.

[7] W. Gao, P. Sun, Y. Wen, and T. Zhang, "Titan: A scheduler for foundation model fine-tuning workloads," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 348–354.

[8] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park, "Elastic resource sharing for distributed deep learning," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 721–739.

[9] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "Pipetransformer: Automated elastic pipelining for distributed training of large-scale models," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 4150–4159.

[10] Y. Liu, S. Agarwal, and S. Venkataraman, "Autofreeze: Automatically freezing model blocks to accelerate fine-tuning," 2021, *arXiv:2102.01386*.

[11] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Freezeout: Accelerate training by progressively freezing layers," 2017, *arXiv: 1706.04983*.

[12] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein, "SVCCA: Singular vector canonical correlation analysis for deep learning dynamics and interpretability," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6078–6087.

[13] A. Morcos, M. Raghu, and S. Bengio, "Insights on representational similarity in neural networks with canonical correlation," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 5732–5741.

[14] L. Yang, S. Lin, F. Zhang, J. Zhang, and D. Fan, "Efficient self-supervised continual learning with progressive task-correlated layer freezing," 2023, *arXiv:2303.07477*.

[15] Y. Wang, D. Sun, K. Chen, F. Lai, and M. Chowdhury, "Egeria: Efficient DNN training with knowledge-guided layer freezing," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 851–866.

[16] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference,," in, *Low-Power Computer Vision*. London, U.K.: Chapman and Hall/CRC, 2022.

[17] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 88–100, Jan. 2022.

[18] Q. Weng et al., "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 945–960.

[19] Y. Wu et al., "Elastic deep learning in multi-tenant GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 144–158, Jan. 2022.

[20] X. Miao et al., "Galvatron: Efficient transformer training over multiple GPUs using automatic parallelism," *Proc. VLDB Endow.*, vol. 16, no. 3, pp. 470–479, Nov. 2022.

[21] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–16.

[22] Y. Zhao et al., "PyTorch FSDP: Experiences on scaling fully sharded data parallel," 2023, *arXiv:2304.11277*.

[23] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[24] D. Narayanan et al., "Pipedream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, 2019, pp. 4171–4186.

[26] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[27] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[30] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[31] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, Dept. Comput. Sci. Univ. Toronto, Univ. Toronto, 2012.

[32] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, pp. 303–338, 2010.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[34] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *Proc. USENIX Annu. Techn. Conf.*, 2019, pp. 947–960.

[35] X. Xiao, T. Bamunu Mudiyanselage, C. Ji, J. Hu, and Y. Pan, "Fast deep learning training through intelligently freezing layers," in *Proc. 2019 Int. Conf. Internet Things-IEEE Green Comput. Commun.-IEEE Cyber Phys. Social Comput.-IEEE Smart Data*, 2019, pp. 1225–1232.

[36] G. Yuan et al., "Layer freezing & data sieving: Missing pieces of a generic framework for sparse training," 2022, *arXiv:2209.11204*.

[37] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang, "Chronus: A novel deadline-aware scheduler for deep learning training jobs," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 609–623.

[38] W. Gao, Z. Ye, P. Sun, T. Zhang, and Y. Wen, "UniSched: A unified scheduler for deep learning training jobs with different user demands," *IEEE Trans. Comput.*, vol. 73, no. 6, pp. 1500–1515, Jun. 2024.

[39] FairScale authors, "Fairscale: A general purpose modular pytorch library for high performance and large scale training," 2021. [Online]. Available: https://github.com/facebookresearch/fairscale

[40] "NCCL," 2022. [Online]. Available: https://developer.nvidia.com/nccl

[41] D. Li, H. Wang, E. Xing, and H. Zhang, "AMP: Automatically finding model parallel strategies with heterogeneity awareness," in *Proc. Adv. Neural Inf. Process. Syst.*, 2022, pp. 6630–6639.

[42] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for C++ server workloads," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 541–556.

[43] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," 2018, *arXiv: 1806.03822*.

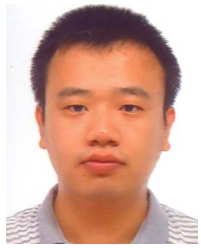
[44] R. Socher et al., "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proc. 2013 Conf. Empirical Methods Natural Lang. Process.*, Seattle, Washington, USA, 2013, pp. 1631–1642. [Online]. Available: https://www.aclweb.org/anthology/D13-1170

[45] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella, "Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 703–723.

[46] E. Cervenka, "Naruto blip captions," 2022. [Online]. Available: https://huggingface.co/datasets/lambdalabs/naruto-blip-captions/
[47] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," 2020. [Online]. Available: https://arxiv.org/abs/2006.11239
[48] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016, *arXiv:1609.07843*.
[49] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 481–498.
[50] D. Narayanan et al., "Solving large-scale granular resource allocation problems efficiently with pop," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 521–537.
[51] J. Gu et al., "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 485–500.
[52] H. Zhao et al., "Hived: Sharing a GPU cluster for deep learning with guarantees," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 515–532.
[53] W. Gao et al., "Autosched: An adaptive self-configured framework for scheduling deep learning training workloads," in *Proc. 38th ACM Int. Conf. Supercomput.*, 2024, pp. 473–484.
[54] S. Pandey, A. Yazdanbakhsh, and H. Liu, "TAO: Re-thinking DL-based microarchitecture simulation," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 8, no. 2, pp. 1–25, May 2024. [Online]. Available: https://doi.org/10.1145/3656012
[55] S. J. Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, and G. R. Ganger, "SIA: Heterogeneity-aware, goodput-optimized ML-cluster scheduling," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 642–657.
[56] K. Goutam, S. Balasubramanian, D. Gera, and R. R. Sarma, "Layerout: Freezing layers in deep neural networks," *SN Comput. Sci.*, vol. 1, no. 5, Sep. 2020, Art. no. 295, doi: 10.1007/s42979-020-00312-x.

Wei Gao received the BS degree from Beihang University, Beijing China in 2019 and the PhD degree from Nanyang Technological University, Singapore in 2025. His research interests include distributed machine learning systems, cluster resource management, and workload scheduling.

Zhuoyuan Ouyang received the MS degree from the School of Physical and Mathematical Sciences from Nanyang Technological University, Singapore, in 2023. He is currently working as a research associate with Nanyang Technological University, Singapore.



Peng Sun received the PhD degree in computer science from Nanyang Technological University, Singapore. He is currently a senior research scientist in SenseTime Group Limited. Previously he worked as a research engineer in Nanyang Technological University, Baidu Institue of Deep Learning and Huawei 2012 Labs. His research interests include cloud computing, computer networking, data center, Big Data and large-scale cluster computing systems for machine learning.

Tianwei Zhang (Member, IEEE) received the bachelor's degree from Peking University in 2011, and the PhD degree from Princeton University in 2017. He is an associate professor with the School of Computer Science and Engineering, Nanyang Technological University. His research focuses on computer system security. He is particularly interested in security threats and defenses in machine learning systems, autonomous systems, computer architecture and distributed systems.

Yonggang Wen (Fellow, IEEE) received the PhD degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2008. He is a professor of computer science and engineering with Nanyang Technological University, Singapore, where he has served as an associate dean (Research) with the College of Engineering since 2018. He serves on Editorial Boards for multiple transactions and journals, including IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, IEEE WIRELESS COMMUNICATION MAGAZINE, IEEE COMMUNICATIONS SURVEY AND TUTORIALS, and IEEE TRANSACTIONS ON MULTIMEDIA.