JABAS: Joint Adaptive Batching and Automatic Scaling for DNN Training on Heterogeneous GPUs

Gyeongchan Yun
UNIST
rugyoon@unist.ac.kr

Junesoo Kang
UNIST
jsonbirth@unist.ac.kr

Hyunjoon Jeong
UNIST
with1015@unist.ac.kr

Sanghyeon Eom
UNIST
djatkdgus789@unist.ac.kr

Minsung Jang
Samsung SDS
mdpe36@gmail.com

Young-ri Choi
UNIST
ychoi@unist.ac.kr

Abstract

Adaptive batching is a promising technique to reduce the communication and synchronization overhead for training Deep Neural Network (DNN) models. In this paper, we study how to speed up the training of a DNN model using adaptive batching, without degrading the convergence performance in a heterogeneous GPU cluster. We propose a novel DNN training system, called *JABAS* (*J*oint *A*daptive *B*atching and *A*utomatic *S*caling). In JABAS, a DNN training job is executed on a DNN training framework called IIDP, which provides the same theoretical convergence rate of distributed SGD in a heterogeneous GPU cluster. To maximize the performance of the job with adaptive batching, JABAS employs adaptive batching and automatic resource scaling jointly. JABAS changes a global batch size every p iterations in a fine-grained manner within an epoch, while auto-scaling to the best GPU allocation for the next epoch in a coarse-grained manner. Using three heterogeneous GPU clusters, we evaluate JABAS for seven DNN models including large language models. Our experimental results demonstrate that JABAS provides 33.3% shorter training time and 54.2% lower training cost than the state-of-the-art adaptive training techniques, on average, without any accuracy loss.

CCS Concepts: • **Computing methodologies** → **Machine learning**.

Keywords: Distributed Systems, Systems for Machine Learning

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696078

ACM Reference Format:
Gyeongchan Yun, Junesoo Kang, Hyunjoon Jeong, Sanghyeon Eom, Minsung Jang, and Young-ri Choi . 2025. JABAS: Joint Adaptive Batching and Automatic Scaling for DNN Training on Heterogeneous GPUs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3696078

1 Introduction

Recent advance in machine learning algorithms and accelerator technologies enables Deep Neural Networks (DNNs) to be used in various domains including image classification [21, 38], object detection [65, 67], machine translation [76, 81], language modeling [8, 14], and recommendation [52]. When speeding up the training of DNN models by leveraging parallelism, adaptive batching is a promising technique to reduce the communication and synchronization overhead. For distributed training, it is beneficial to increase a global batch size, having higher throughput, but a large global batch size may cause accuracy loss. Adaptive batching attempts to resolve this issue such that a DNN training job starts with a small global batch size, but increases the batch size adaptively during training based on a gradient-based metric such as Gradient Noise Scale (GNS) [62], gradient similarity [63], and gradient norm [3].

However, it is still challenging to use adaptive batching effectively for distributed training. First, the existing state-of-the-art adaptive batching system which changes a global batch size based on GNS and throughput [62] may suffer from noticeable accuracy loss as prior studies pointed out [24, 84]. Second, to optimize the performance of a DNN training job, adaptive batching needs to be integrated with automatic resource scaling sophisticatedly. Existing studies increase the amount of resources for training a job without increasing a global batch size during training [20, 25, 33] or increase a global batch size while using a fixed amount of resources [63], resulting in sub-optimal performance. With adaptive batching, as the job progresses, a global batch size tends to increase over time as the negative effect of a large global batch size becomes less [49, 74]. Thus, when scaling resources automatically for the job, it is essential to consider the characteristics

of adaptive batching. Third, adaptive batching needs to be applied efficiently for heterogeneous GPU environments. While prior adaptive batching systems [3, 62, 63] only consider homogeneous GPU clusters, heterogeneous GPU clusters become inevitable [9, 29, 30, 51, 78] as new GPU architectures are released at a rapid pace. For example, it was reported that three and six types of heterogeneous GPUs are used in Microsoft [9] and Alibaba PAI production clusters [78], respectively, for DNN training. For a heterogeneous GPU cluster, the existing systems with the same mini-batch size per GPU (i.e., local batch size) do not fully utilize the GPUs due to stragglers. To improve the efficiency of heterogeneous GPUs, different local batch sizes are used for training [10, 82], but this violates the important assumption of "Mini-batch samples are independent and identically distributed (i.i.d.)". We observe that using different local batch sizes for adaptive batching results in a significant convergence degradation (Section 2.3). Thus, effective adaptive batching techniques that increase GPU efficiency while preserving convergence are necessary.

In this work, we study how to speed up the training of a DNN model using adaptive batching, without degrading the convergence performance in a heterogeneous GPU cluster. We propose a novel DNN training system, called *JABAS* (*J*oint *A*daptive *B*atching and *A*utomatic *S*caling). In JABAS, a DNN training job is executed on a DNN training framework, *I*ndependent and *I*dentical *D*ata *P*arallelism (IIDP), where a heterogeneous GPU can process a different number of training samples depending on its computation capability and memory capacity while preserving the statistical efficiency as in a homogeneous GPU cluster. To maximize the performance of the job with adaptive batching, JABAS employs adaptive batching and automatic resource scaling jointly. JABAS changes a global batch size every p iterations in a fine-grained manner within an epoch, while JABAS auto-scales to the best GPU allocation for the next epoch in a coarse-grained manner.

The main contributions of this work are as follows: First, we develop a novel DNN training framework, IIDP. In IIDP, each worker uses the same local batch size, but multiple workers called *virtual stream workers* (VSWs) can be executed concurrently on a single GPU by leveraging the GPU multi-stream. IIDP also allows these VSWs to execute multiple mini-batches via Gradient Accumulation (GA). Furthermore, IIDP employs an efficient synchronization mechanism among VSWs, that is, the local aggregation and weight update techniques. Second, we present a configuration solver based on dynamic programming to configure a job running on IIDP. It predicts the iteration time of the job and finds the best configuration in terms of a local batch size, the numbers of VSWs, and GA steps for each GPU from a large complex search space. Third, we present a fine-grained adaptive batching technique that uses a fixed allocation of heterogeneous GPUs within an epoch, but adapts a global batch size every p iterations based on the gradient similarity [63] and reconfigures VSWs of the job for an adapted global batch size. Fourth, we introduce a coarse-grained automatic resource scaling technique, where JABAS optimally scales heterogeneous GPU resources based on the prediction of the global batch size trajectory using ensemble learning. At last, we implement JABAS[1] based on PyTorch, a commonly used Deep Learning (DL) framework. Using three heterogeneous GPU clusters, we evaluate the training time and cost of JABAS for seven different DNN models including large language models, BERT-large, GPT3-XL, and LLaMA2-7B. Our experimental results demonstrate that JABAS provides 33.3% shorter training time and 54.2% lower training cost than the state-of-the-art adaptive training techniques, on average, without any accuracy loss.

2 Background and Motivation

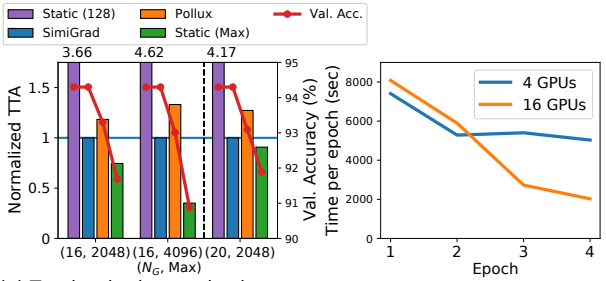
2.1 Distributed Training with Adaptive Batching

Data Parallelism (DP) To speed up DNN training with a large dataset, the most common parallel technique, Data Parallelism (DP) has been leveraged. Each GPU (referred to as a worker) has a replica of a DNN model and computes gradients locally from a mini-batch in a partitioned dataset, which are synchronized among the workers iteratively. In terms of a mini-batch size in DP, the size of a mini-batch processed by each worker is referred to as *local batch size*, while the total mini-batch size processed by all workers is referred to as *global batch size*.

Adaptive batching As synchronization overhead becomes the bottleneck and scalability of throughput is limited, large-batch training is applied to reduce the number of synchronization steps. However, a large global batch size may incur a generalization gap which degrades the DNN model accuracy [23, 71]. To overcome this problem, an *adaptive batching* technique that starts with a small global batch size and increases the batch size during training has been introduced [3, 62, 63]. With adaptive batching, a learning rate scaling rule is required to adjust a learning rate by a factor of global batch size change [32, 37].

Pollux [62] determines an adaptive global batch size based on the *goodput* metric which considers both training throughput and statistical efficiency measured by GNS [49]. GNS can measure a quantity of how much training progress can be made by increasing a global batch size compared to the current size. GNS values are monitored by workers over an epoch. If an estimated value of goodput for a hypothetically increased global batch size is larger than a threshold, Pollux concludes that the increased global batch size is good to use for training. However, once a global batch size increases, Pollux will not decrease it, even if the current global batch size hurts the convergence. It has been reported that adap-

[1] https://github.com/unist-ssl/JABAS



(a) Total train time and val. accuracy (b) Perf. with different config.

Figure 1. Performance of static and adaptive training and performance with different numbers of GPUs

tive batching based on the goodput suffers from accuracy loss [24, 84].

SimiGrad [63] is proposed to reduce the computation overhead of GNS. It leverages a simple gradient similarity metric such that it divides the workers of a DNN training job into two groups, computes the cosine similarity of the aggregated gradients of each group, and compares the value with a given threshold. As the overhead of computing a gradient similarity metric is small, a global batch size can be adjusted at fixed intervals, that is, every p iterations. If the measured gradient similarity is lower (higher) than the target threshold, SimiGrad can increase (decrease) a global batch size responsively in a fine-grained manner. Note that Pollux and SimiGrad use Gradient Accumulation (GA) [19] if the targeted global batch size cannot be configured by the number of GPUs and the amount of GPU memory, in total, given for the job.

Figure 1(a) shows the comparison of training methods with static and adaptive batching for ResNet-18 on CIFAR-10. We experiment with two configuration parameters: (i) the total number of GPUs, N_G (ii) the maximum global batch size, Max, for Pollux and SimiGrad. Pollux changes a global batch size every epoch, while SimiGrad changes it every 100 iterations, and their initial global batch size is set to 128. In the figure, x-axis is a configuration of (N_G, Max), while y-axis on the left side is the time to target accuracy (TTA) normalized to that of SimiGrad and that on the right side is a validation accuracy, and Static(x) indicates static training with a global batch size of x. For the experiments with 16 and 20 GPUs, we use four and five nodes, respectively, each of which has 4 TITAN Xp GPUs. From the results, we observe a similar performance trend for all three different configurations. The main takeaways are as follows: (i) Static(Max) shows the fastest training time, but the accuracy loss occurs compared to the target accuracy (94.3%) due to the large global batch size, while Static(128) has the slowest training time but provides the desirable convergence quality. (ii) SimiGrad achieves the second fastest training time while achieving the target accuracy as Static(128). (iii) Pollux is slower than SimiGrad while facing accuracy loss. (The performance of Pollux will be discussed in detail in Section 7.7.) Therefore, in this work, we chose to use SimiGrad as a basic method for adaptive batching.

2.2 Issues of Adaptive Batching with Fixed Resources

As prior adaptive batching systems simply use all of the GPU resources given for a job[2], the performance of the job will vary depending on a given GPU allocation. Figure 1(b) shows the training time per epoch with SimiGrad [63] for GNMT using two different GPU allocations of 4 and 16 TITAN Xp GPUs. Both experiments begin with an initial global batch size of 128, increasing to a maximum of 4096. For the first two epochs, the performance with 4 GPUs is 10.13% higher than that with 16 GPUs because of a small local batch size and high communication overhead with 16 GPUs. With 16 GPUs, a fraction of the training time for the first two epochs is 31.7%. As a global batch size increases, from the third epoch, the performance with 16 GPUs becomes 51% higher than that with 4 GPUs, resulting in a total training time that is 70% shorter than that with 4 GPUs. The overall cost of 16 GPUs for training is 2.34 times higher than that of 4 GPUs. From the results, we can observe that using a large number of GPUs at the beginning of training where a small global batch size is used may not be beneficial because of the communication overhead, and it may not be cost-effective, possibly causing low resource utilization. On the other hand, using a small number of GPUs for the later part of training increases the training time, even though it provides low cost. Therefore, it is critical to scale GPU resources automatically according to the increase of a global batch size.

2.3 Issues of Heterogeneous GPUs

Prior adaptive batching systems [3, 62, 63] have adopted DP as a parallel training method. A naive way to train a DNN model with DP in a heterogeneous GPU cluster composed of GPUs with different computation capabilities and memory amounts is to choose the same local batch size for the GPU with the smallest memory in the cluster. In this case, the performance is limited by the GPU with the lowest computation capability, suffering from low throughput. To mitigate this issue, the previous studies [10, 82] use an imbalanced local batch size depending on a computation capability, along with *weighted gradient aggregation* which adopts an imbalanced (i.e., different) local batch size as a weight when aggregating local gradients. We refer to DP with the weighted gradient aggregation as *WDP*.

As WDP is a widely adopted technique [33, 41, 59] in heterogeneous GPU environments, an adaptive batching technique may simply adopt WDP. However, we observe that WDP causes a significant convergence degradation for static and adaptive training. Figure 2 shows the total training time

[2]Pollux [62] also presents a simple automatic resource scaling system with adaptive batching. This will be compared with our work in Section 7.

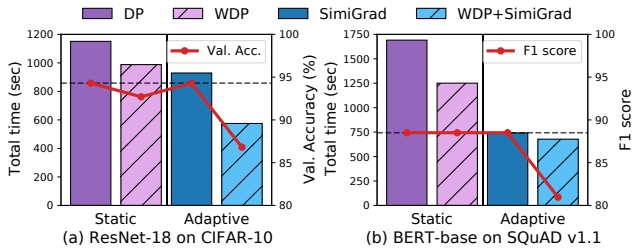


Figure 2. Adaptive batching on heterogeneous GPUs

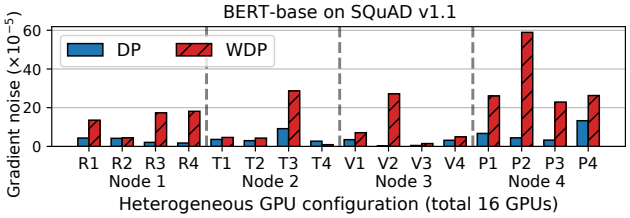


Figure 3. Gradient noise of DP and WDP. GPU computational capability is RTX 3090 (R) > TITAN RTX (T) > V100 (V) > P100 (P).

and validation result of both static and adaptive training for two DNN models, ResNet-18 with CIFAR-10 and BERT-base with SQuAD v1.1 on a heterogeneous GPU cluster composed of 16 GPUs with four different types, called cluster A, which will be explained in Section 7.1. In the figure, the two static training techniques, the naive DP, denoted as DP, and WDP are used, while the two adaptive training techniques, SimiGrad [63] (with the same local batch size as in DP), and SimiGrad with WDP, denoted as WDP+SimiGrad, are used. When WDP is used for static and adaptive training, the training time is shortened for both the DNN models. However, even though static training of BERT-base with WDP achieves the same validation result as DP, the validation results with WDP in all the other cases degrade by up to 7.5%.

WDP suffers from convergence degradation as WDP entails the overlooked yet critical issue of breaking the theoretical assumption that "Mini-batch samples are i.i.d." by using different local batch sizes. Since gradients are stochastic estimations of true gradients, gradient noise (variance) depends on a local batch size [61]. In other words, WDP cannot consider the differences in gradient noise from imbalanced (small and large) local batch sizes. This leads to a difference in convergence semantics [17, 35, 61, 77], which is a root cause of degraded model quality. Moreover, DNN models with Batch Normalization (BN) [26] can be more adversely affected as BN adjusts the activation output with the mean and variance of local mini-batch samples, being influenced by how many samples are collected (i.e., local batch size) [19, 23]. Figure 3 shows the gradient noise of DP and WDP measured for BERT-base after a certain number of iterations on cluster A. From the results, we can see that the gradient noise of WDP is higher than that of DP. In particular, the gradient noise with WDP measured in Node 4, which has the slowest GPU type resulting in the smallest local batch size, is significantly higher than those with WDP in the other nodes as well as those with DP, since a smaller local batch size causes a higher gradient noise [61]. With adaptive batching, WDP becomes a more serious problem because the aforementioned gradient-based metrics are derived based on gradient variance information under the i.i.d. assumption [49, 63]. Therefore, the gradient-based metric with WDP does not properly reflect the current statistical efficiency, observing a tendency that WDP+SimiGrad increases a global batch size incorrectly at the early part of training for the experiment in Figure 2.

Instead of WDP, one of the possible ways to effectively use heterogeneous GPUs for DNN training is to decouple workers from the GPUs such that a single GPU is used to execute multiple workers, each of which processes training samples of the same local batch size. To achieve this, we leverage the GPU multi-stream within the same process. A GPU stream is a queue of GPU operations (i.e., kernel, memory copy) executed in the FIFO order, while GPU kernels in different streams have no dependency. Note that spatial sharing techniques such as NVIDIA MPS and MIG [53, 54] help multiple processes share Streaming Multiprocessors (SMs). However, MPS may cause performance degradation by GPU memory over-subscription [15, 45], and also a popularly used optimized communication library, NCCL [55], cannot be used with MPS as it allows only one process per GPU to communicate with other GPUs [56]. For MIG, it is only supported on the latest GPU architectures such as Ampere and Hopper. Therefore, these techniques are not considered in this work.

3 System Overview

In JABAS, each node is configured with multiple homogeneous GPUs, but the nodes can have different types of GPUs and the number of GPUs per node can vary. Our proposed system JABAS aims to jointly leverage adaptive batching and automatic resource scaling for a DNN training job on heterogeneous GPUs, maximizing the throughput without affecting the convergence negatively.

Figure 4 shows the system architecture and the workflow of JABAS. There are two novel aspects in the architecture of JABAS. Firstly, heterogeneous GPUs can be used in the same manner as homogeneous GPUs for training via the *independent and Identical Data Parallelism* (IIDP) framework that enables a single GPU to execute multiple workers. When running a job on IIDP while improving the hardware efficiency of heterogeneous GPUs, the same local batch size can be used for each worker without violating the important assumption of i.i.d. Secondly, JABAS employs adaptive batching and automatic resource scaling jointly. It adapts a global batch size

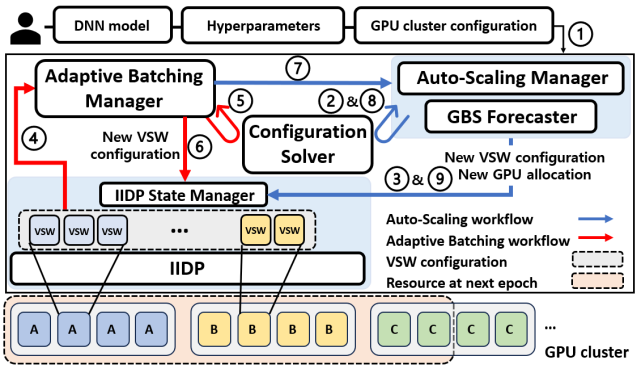


Figure 4. System overview

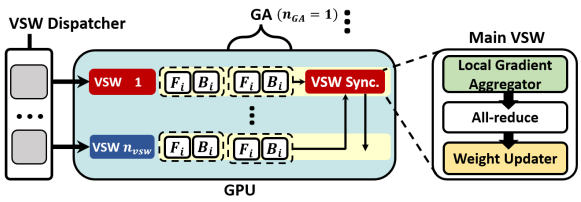


Figure 5. Overview of IIDP

every p iterations, within the same epoch, in a fine-grained manner, minimizing the negative impact of adaptive global batch sizes. On the other hand, as a global batch size may fluctuate over time, JABAS performs automatic resource scaling for each epoch in a coarse-grained manner, maximizing the training efficiency. To optimize performance, the trajectory prediction of global batch sizes based on ensemble learning is used to scale GPU resources for the next epoch.

We first describe two basic components of JABAS and then discuss the workflow of JABAS.

IIDP IIDP leverages the GPU multi-stream features to execute the multiple workers on the same GPU, invoking GPU kernels issued by each of the workers on a different stream. This worker is referred to as *virtual stream worker* (VSW). In addition, each VSW can process multiple mini-batches sequentially using gradient accumulation (GA). Therefore, within a single GPU, IIDP leverages the combination of parallel processing via the multi-stream and sequential processing via GA, maximizing the training throughput. To execute a job on IIDP using a set of heterogeneous GPUs, IIDP needs to configure the numbers of VSWs, and GA steps on each GPU for a given global batch size. This configuration is provided by the *configuration solver*.

Configuration solver In a heterogeneous GPU cluster, depending on the GPU type, the range of possible local batch sizes used for VSWs varies, and for a given local batch size, the maximum number of VSWs for each GPU, and consequently, the computation time of the GPU also vary. Among numerous possible configurations on IIDP, the configuration solver finds the best configuration, that is, a local batch size, B_L used by VSWs, and the numbers of VSWs, n_{vsw}, and GA steps, n_{GA}, for each GPU type, which balances the computation load of each GPU mitigating a straggler for a given GPU allocation and global batch size. To do so, it uses a *performance prediction model* to accurately estimate the iteration time for each configuration and a dynamic programming algorithm. This is used by other components in JABAS so that the job can run on IIDP with the best performance.

Workflow of JABAS To train a DNN model on heterogeneous GPUs, an initial global batch size, the minimum local batch size, and a set of heterogeneous GPUs to train with, G_{all}, are given by a user (①). JABAS has two phases for executing a DNN training job.

Initial phase For the initial global batch size, JABAS first finds the best GPU allocation using the *auto-scaling manager* (Section 6.3). For each possible GPU allocation, the auto-scaling manager finds the best configuration using the configuration solver and estimates the epoch time (②). It then selects a GPU allocation with the minimum epoch time. Note that for the best GPU allocation, a subset of G_{all} can be used. The selected GPU allocation with its best configuration is sent to the IIDP state manager (③). After configuring each GPU, the execution of the job is initiated on IIDP.

Runtime phase During an epoch, for every p iterations, each VSW participates in computing the gradient similarity metric [63] and forwards it to the *adaptive batching manager* (④) (Section 6.1). Based on the metric, the adaptive batching manager determines a global batch size which will be used for the next p iterations. In turn, the best configuration for this adapted global batch size is computed using the configuration solver (⑤) and is sent to the IIDP state manager (⑥), continuing the training. At the end of the epoch, the adaptive batching manager sends the trajectory of global batch sizes during the epoch to the *Global Batch Size (GBS) forecaster* (⑦) (Section 6.2), which predicts the trajectory of global batch sizes for the next epoch using ensemble learning of the Gaussian Process Regression (GPR) and Exponential Smoothing model (ESM). The auto-scaling manager then uses the trajectory prediction to auto-scale the GPU resources, that is, to find the best GPU allocation for the next epoch using the configuration solver (⑧). This phase continues (⑨) until the job is done.

4 IIDP

IIDP aims to preserve the theoretical assumption regardless of GPU heterogeneity while maximizing throughput. With a static global batch size N and the number of iterations T, training on IIDP can guarantee the same convergence rate of distributed SGD: $O(1/\sqrt{NT})$ [40]. In developing IIDP, the following considerations should be incorporated. First, even though a larger number of workers are used for training as multiple workers are executed on a single GPU, the communication overhead among them should be minimized. Second,

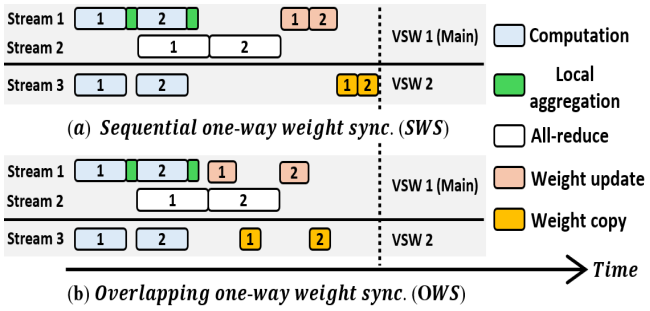


Figure 6. Two methods of one-way weight synchronization

executing multiple optimizers on the single GPU may hurt throughput significantly (which is discussed in Section 7.4). Such optimizer overhead should be reduced.

Figure 5 shows the system architecture of IIDP. For each GPU, a multi-threaded *GPU process* is created to run n_{vsw} *VSW dispatchers*. Each VSW dispatcher will execute a VSW by concurrently launching GPU kernels that are needed to process a mini-batch or mini-batches via GA if $n_{GA} > 0$ on different GPU streams. In addition, within a VSW, multi-stream can be used to support IIDP features and GPU kernel optimization libraries such as cuDNN [11]. IIDP has a *main VSW* per GPU, which manages the synchronization among the VSWs. The *local gradient aggregator* in the main VSW aggregates gradients locally, which is followed by global synchronization via All-reduce communication. Once the synchronization is done among the main VSWs, the *weight updater* in each main VSW updates the weights and copies them to the other VSWs running on the same GPU.

Local and global gradient aggregation At each iteration, every VSW (including a main VSW) independently computes the gradients. Once the computation is done in each VSW, the main VSW triggers the local gradient aggregator to aggregate all gradients computed by the VSWs in the same GPU. This is designed to make use of NCCL [55] for All-reduce operations [31, 34, 68] across GPUs. The main VSW then communicates with other main VSWs via All-reduce communication for global synchronization such that the collected gradients are averaged by the sum of $n_{vsw} * (n_{GA} + 1)$ for every GPU, not the total number of GPUs, in the cluster. As a result, IIDP achieves the same gradient aggregation semantics and communication overhead of the naive DP.

In this process of the local and global aggregation, IIDP employs a *gradient buffer*, which is filled by multiple gradients, as a unit of operations similar to existing DL frameworks and techniques [43, 68, 69]. Each VSW computes a gradient in a backward pass and copies it to its assigned gradient buffer whose value will be aggregated to the main VSW's gradient buffer by the local aggregator. Then, the main VSW proceeds global aggregation for this gradient buffer using a stream different from the gradient computation stream, overlapping the computation with the communication.

One-way weight synchronization within a GPU As a main VSW only has the aggregated gradients after the global aggregation, the main VSW updates the DNN model weights (*weight update*) by an optimization algorithm (e.g., SGD, and Adam), called *optimizer*, and copies the updated version to all the other VSWs in the same GPU (*weight copy*). We devise two methods of processing weight update and copy: (i) Sequential one-way Weight Synchronization (SWS) and (ii) Overlapping one-way Weight Synchronization (OWS).

Figure 6(a) and (b) show the execution of SWS and OWS, where a number in a box indicates an index of a gradient buffer. For SWS, the weight update and copy procedures (represented by red and yellow boxes, respectively, in the figure) are executed sequentially after All-reduce communications (represented by white boxes) are done. For OWS, the weight update and copy procedures can be overlapped with the All-reduce operation. To achieve this, we develop a *shard optimizer* which can update a part of the DNN model weights. When the All-reduce operation is done for a gradient buffer, the weight updater of a main VSW executes this shard optimizer to do the partial weight update and then proceeds with the partial weight copy for the gradient buffer. At the same time, since the All-reduce operation and the weight update operation run on different streams, the All-reduce operation for the next gradient buffer can continue, thereby speeding up the training.

One-way weight synchronization method selection When the All-reduce operation for a large gradient buffer, is overlapped with the weight update and copy operation in OWS, performance interference can occur. At runtime, our selection algorithm decides which method will be used for a DNN model such that if a ratio of the average size of the gradient buffers to the size specified by the user is larger than a threshold, R_{buffer}, SWS is used and otherwise, OWS is used. In our experiments, we set R_{buffer} to 50%.

5 Configuration for a DNN Training Job

5.1 Performance Prediction Model

To find the best configuration of VSWs for a DNN training job, it is necessary to accurately estimate the iteration time according to the numbers of VSWs, n_{vsw} and GA steps, n_{GA}, with a certain local batch size, B_L, on each heterogeneous GPU g. In our prediction model, we consider that gradient buffers are used for communication, and also take the optimizer overhead into account.

While the minimum local batch size $minB_L$ is given by a user, JABAS profiler finds the maximum local batch size, $maxB_L^g$, that a VSW can process on each g by increasing a local batch size in the unit of $minB_L$, and also finds the maximum number of VSWs, n_{max}, which can be executed concurrently on g, with each local batch size. Note that we only consider multiples of $minB_L$ as possible local batch sizes to reduce the profiling overhead and the complexity

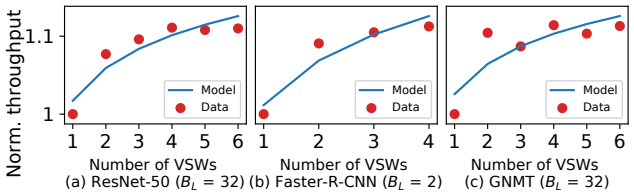


Figure 7. Prediction with logarithmic regression model

of finding the best configuration. For each local batch size B_L from $minB_L$ to $maxB_L^g$, JABAS profiles the computation time of GPU g for an iteration, including the ratio of forward and backward passes ζ, when n_{vsw} is set to 1 and n_{max}. With the profiled times of the above two cases, we first build a computational throughput model to predict the computation time of VSWs for given n_{vsw} and B_L, on GPU g in Eq. (1).

$$thp_{vsw}(B_L, n_{vsw}, g) = \alpha_{thp} * ln(n_{vsw}) + \beta_{thp} \quad (1)$$

We choose the logarithmic regression model [12] as we observe that the throughput trend of VSWs is similar to the roofline model [80] of an accelerator. Figure 7 shows the predicted and actual throughput for three models on a TITAN RTX GPU over various values of n_{vsw}. In the figure, our model provides the average prediction error of 1.46%. Then, the computation time of n_{vsw} VSWs that process a mini-batch of B_L on GPU g is as follows:

$$t_{vsw}(B_L, n_{vsw}, g) = (B_L * n_{vsw})/thp_{vsw}(B_L, n_{vsw}, g) \quad (2)$$

For global gradient synchronization time, JABAS profiler gets the size of all gradient buffers for a DNN model. For the size of transferred data d, the total number of GPUs N_G, and network bandwidth K, the theoretical All-reduce time is $(d * 4(N_G - 1)/N_G)/K$ [34]. In practice, as the peak network bandwidth cannot achieve the theoretical one, we build two linear regression models (in Eq. (3)) for practical All-reduce time predictions within a node and between nodes, respectively, via collecting real All-reduce times of 10 tensors of arbitrary size.

$$t_{allreduce}(d) = \alpha_{comm} * (d * 4(N_G - 1)/N_G)/K + \beta_{comm} \quad (3)$$

For the size of gradient buffer $b_i (1 \le i \le l)$ where l is the total number of gradient buffers, the total communication time for a backward pass is estimated as follows:

$$t_{comm} = \sum_{i=1}^{l} t_{allreduce}(b_i) \quad (4)$$

To estimate the weight synchronization time, JABAS profiles optimizer time t_{optim} and weight copy time t_{copy}. We observe an estimated time as follows:

$$t_{update}(n_{vsw}, g) = \tau * (t_{optim} + t_{copy} * (n_{vsw} - 1)) \quad (5)$$

When OWS is used in IIDP, the weight synchronization and All-reduce operations can overlap, except for the gradients in the last gradient buffer b_l. Therefore, $\tau \approx b_l/W$ where the total parameter size of the model is W. On the other hand, $\tau = 1$ for SWS.

Algorithm 1 Configuration Solver

```
Input: B_G, G
Output: (B*_G, B*_L, C*)
B*_G, B*_L, Thp*, C* ← 0, 0, 0, {}
maxB_L ← GetMaxLocalBatchSize(minB_L, G)
H ← OneOfTwoGroups(G)                    ▷ Divide G into two identical groups
for B_L in [minB_L, maxB_L] do           ▷ For each possible local batch size
    A ← {}
    M ← round(B_G/B_L/2)                 ▷ Num. of decoupled workers in a group
    If M < N_H then break                ▷ No solution
    for g in H do                        ▷ Find all possible configurations of g for B_L
        maxn_vsw ← min(M, GetMaxNumberOfVSWs(g, B_L)))
        for n_vsw in [1, maxn_vsw] do
            for n_GA in [0, ⌊M/n_vsw⌋] do
                t ← EstimateIterationTime(B_L, n_vsw, n_GA, g)
                A ← A ∪ {(g, n_vsw, n_GA, t)}
            end for
        end for
    end for
    C ← {a ∈ A  s.t  a.n_vsw * (a.n_GA + 1) = 1}
    A ← PruneSlowConfigurations(A\C)
    for a in SortedByIterationTime(A) do
        C ← argmax(F(C), F(C ∪ a))
    end for
    if F(C) > Thp* and C.n_w = M then
        B*_G, B*_L, Thp*, C* ← B_L * M * 2, B_L, F(C), C
    end if
end for
```

To sum up, an iteration time of the corresponding arguments $(B_L, n_{vsw}, n_{GA}, g)$ is estimated by Eq. (6) where $t_{vsw}^{fwd} = t_{vsw} * \zeta$ and $t_{vsw}^{bwd} = t_{vsw} * (1 - \zeta)$.

$$t = t_{vsw} * n_{GA} + t_{vsw}^{fwd} + max(t_{vsw}^{bwd}, t_{comm}) + t_{update} \quad (6)$$

5.2 Configuration Solver

The goal of the configuration solver is to find a configuration of a job on IIDP, which maximizes the throughput for a given global batch size, B_G, and a set of, possibly heterogeneous, GPUs, G. As IIDP decouples workers from the GPUs using VSWs and GA, the total number of *decoupled workers*, n_w is computed as the sum of $n_{vsw} * (1 + n_{GA})$ for each GPU, and thus, B_G is computed as $n_w * B_L$. In this case, the goal is equivalent to minimizing an iteration time spent by the slowest decoupled worker in the configuration for decoupled workers of n_w and it has the optimal sub-structure property (which will be discussed as an example at the end of this section). Therefore, we employ a dynamic programming algorithm. If the estimation of an iteration time is accurate, the configuration solver guarantees the optimal solution.

Algorithm 1 shows the configuration solver. For a given global batch size B_G, minimum local batch size $minB_L$, and a set of GPUs G, the algorithm finds the best configuration based on the prediction of the performance over various values of B_L in the range between $minB_L$ and $maxB_L$. To compute $maxB_L$, the configuration solver first gathers $maxB_L^g$ for each GPU g in G from the profiled results and selects

the minimum value among them (Line 4). To adopt SimiGrad [63] for adaptive batching, decoupled workers need to be divided into two groups so that the gradient similarity aggregated within each group is compared with each other. For G, we assume that each node consists of an even number of GPUs, and thus, the total number of allocated GPUs is also even. In this case, we consider that a GPU allocation unit consists of two GPUs located in the same node, and the two GPUs are an identical pair. Therefore, G can be divided into two identical groups such that one GPU from each pair belongs to each group (Line 5).

For each local batch size B_L, the configuration solver first generates a search space A (Line 7-18). M is the total number of decoupled workers in each group, which is a rounded value of $B_G/B_L/2$ (Line 8). (Note that as the value is rounded to make M an integer, B_G^* which is the computed output for a global batch size can be different a bit from B_G which is provided as input.) If the number of GPUs in each group H (i.e., N_H) is larger than M, then the given B_L cannot be used as some of the GPUs in H will not be used for training, having no possible configuration with B_L (Line 9). For each GPU g in H, it finds all possible configurations of g (Line 10-18). Each configuration is different in terms of n_{vsw} and n_{GA}, but the number of decoupled workers in g is from 1 to M. For each configuration on g, the configuration solver estimates the iteration time using the performance prediction model (Eq. (6) in Section 5.1) and adds the configuration to A (Line 14-15).

After the search space A is computed, for each unique number of decoupled workers on every g, the configuration solver prunes slower configurations with longer iteration times and keeps only one configuration with the minimum iteration time (Line 20). It then sorts the remaining configurations by their iteration times and uses them for dynamic programming to find the best configuration with B_L (Line 21-23). Let L be the number of possible B_L. The complexity of dynamic programming is reduced to $O(LM^2N_H)$, while without pruning, the complexity is $O(LM^3N_H)$.

We define the optimal sub-structure as F and the objective of dynamic programming in Eq. (7) where $C.n_w$ and $C.t$ indicate the number of total decoupled workers and maximum iteration time with a configuration C, respectively. The throughput with C, $F(C)$, is computed as a global batch size $(= B_L * C.n_w * 2)$ divided by $C.t$.

$$\underset{C^*}{\text{argmax}}\, F(C) = (B_L * C.n_w * 2)/C.t \quad \text{s.t.} \quad C.n_w = M \qquad (7)$$

Figure 8 demonstrates an example of Algorithm 1, where a yellow box indicates a decoupled worker. Consider a case of $B_G = 12$, $H = \{GPU_A, GPU_B\}$, and $B_L = 1$. Then, $M = 6$ (=12/1/2). The top of the figure shows a process to generate a search space A for $B_L = 1$ (Line 7-18), where for each GPU of H, all possible configurations where $M \leq 6$ are added to A. Then, the configurations are sorted by the iteration times as in the figure. For GPU_A, two configurations, a_1 and a_2, have

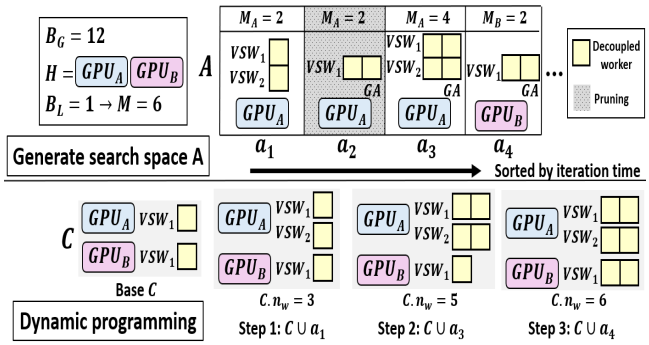


Figure 8. An example of Algorithm 1

the same number of decoupled workers (i.e., $M_A = 2$). Thus, a_2 is pruned as its iteration time is longer than that of a_1 (Line 20). The bottom of the figure shows a dynamic programming process for $B_L = 1$ (Line 21-23) where starting with the base configuration C (Line 19), the solver enumerates each element of search space A, adding a new element to C that can increase F with the constraint of $C.n_w = 6$. Consequently, $C^* = \{a3, a4\}$ which has the shortest iteration time, resulting in maximizing throughput, with $C.n_w = 6$. In this example, the optimal sub-structure property is shown such that C^* is comprised of sub-configurations that have the minimum iteration time for n_w=3 (in Step 1) and for n_w=5 (in Step 2).

6 Joint Adaptive Batching and Automatic Scaling

6.1 Fine-grained Adaptive Batching

Using a fixed set of heterogeneous GPUs, G_{cur} during an epoch, the adaptive batching manager aims to change a global batch size based on the gradient similarity [63] and to reconfigure VSWs running on G_{cur} for an adapted global batch size, so that the training throughput is maximized. In our adaptive batching algorithm, the same user-defined parameters such as a similarity threshold γ, an adaptive interval p, and an adaptive rate α and the lower and upper bound of a global batch size (i.e., $minB_G$, and $maxB_G$) need to be provided as in SimiGrad [63]. Every p iterations, JABAS computes the gradient similarity Φ for aggregated gradients in the two groups of G_{cur}. Note that unlike SimiGrad where the number of workers in each group is the same as the number of GPUs in the group, in the case of JABAS, the number of workers in each group is computed as $n_w/2$. If the measured gradient similarity Φ is less (more) than γ, JABAS increases (decreases) current B_G by α to B'_G. In addition, B'_G must be satisfied with $[minB_G, maxB_G]$.

For B'_G, JABAS finds the best configuration on G_{cur} in terms of a local batch size B_L, along with n_{vsw} and n_{GA} for each GPU, by the configuration solver, and then also updates the learning rate by the scaling rule with B'_G. To apply the updated configuration, JABAS calls the IIDP state manager.

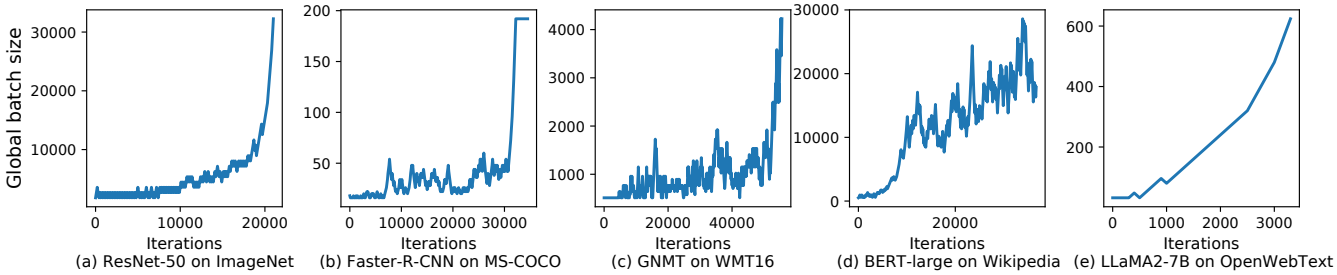


Figure 9. Global batch size trajectory by adaptive batching

The reconfiguration process can be done with negligible overhead on IIDP. As each GPU process is a multi-threaded process, when n_{vsw} increases (decreases), a thread can be added to (deleted from) the GPU process at runtime, without restarting the GPU process.

It is important to note that existing DL frameworks optimize the data loading for batched data of a static local batch size and do not consider changing the data sampling size (of the local batch size) initialized at the beginning of training. To support data sampling for dynamic B_L, the IIDP state manager notifies the data loader to change the sampling size of the data sampler and to manage to load data for the updated B_L onto the GPU.

6.2 Global Batch Size Forecast

Frequent GPU reallocations to accommodate fluctuated global batch sizes by adaptive batching during an epoch can cause high overhead as JABAS needs to checkpoint and restart the job with a new GPU reallocation. Therefore, JABAS's auto-scaling manager is designed to perform GPU reallocation in a coarse-grained manner, but it makes a resource scaling decision based on a predicted trajectory of global batch sizes for the next epoch.

Figure 9 shows how a global batch size changes during training for various DNN models. In the figures, a global batch size tends to increase exponentially while also experiencing periodic fluctuations in different patterns. The correlation between a current value of the gradient-based metric and a future global batch size is complex and difficult to know. Therefore, we consider using Gaussian Process Regression (GPR) [79] which is a nonparametric Bayesian approach to regression for the prediction. The advantages of GPR are as follows [4, 60]: (i) It can be applied to various DNN models without prior domain knowledge. (ii) It requires only a small number of samples. (iii) It can tolerate uncertainty. GPR can accurately predict a trajectory with high fluctuation, but it cannot consider the temporal dependency of global batch size changes. To reflect the temporal dependency, we then consider using the Exponential Smoothing model (ESM) [18], which is based on the exponential moving window function. We chose to use ESM as its characteristics match the changes (i.e., exponential increase with some fluctuations) of a global batch size in our case. Not only to take advantage of both prediction models but also to stabilize the prediction error rate for the models, we employ a well-known ensemble-based approach, which runs the two models concurrently and calculates the average predicted value during evaluation. Furthermore, since the time cost of ESM is very small compared to GPR, the difference in the time cost between GPR and the ensemble model during every evaluation step is negligible.

Note that previously, Shockwave [84] attempted to predict the remaining time of a DNN training job with adaptive batching for cluster scheduling by the Bayesian statistical model. However, it cannot be adopted directly to our work because it regards a possible value of the adaptive global batch size as prior knowledge, and does not consider the temporal trend.

6.3 Coarse-grained Automatic Scaling

The auto-scaling manager is called for two cases to find a GPU allocation that results in the minimum duration for the next epoch. The first case is to find an initial GPU allocation for an initial global batch size B_G^{init} of a DNN training job, while the second case is to automatically scale a GPU allocation for the next epoch as the training progresses. When a set of heterogeneous GPUs, G_{all}, can be used for training, JABAS first generates all the candidate GPU allocations, Q, which needs to be done only once at the beginning. Considering n nodes, each of which has m_i GPUs, there are $m_i/2$ groups on each node i. Preserving that GPUs in each node are in the identical type, the number of possible GPU allocations is $\prod_{i=1}^{n}(m_i/2+1)-1$. (For example, if there are two nodes, N1 composed of four GPUs with a type g_A, and N2 composed of two GPUs with a type g_B, Q is computed as (g_A, g_A), (g_B, g_B), (g_A, g_A, g_A, g_A), (g_A, g_A, g_B, g_B), and $(g_A, g_A, g_A, g_A, g_B, g_B)$.)

For each possible GPU allocation G, JABAS finds the best GPU allocation by estimating the epoch time for the next epoch based on the predicted global batch size trajectory. For the initial GPU allocation, the trajectory of global batch sizes is not available, and thus, JABAS assumes that a global batch size B_G for each p iterations is B_G^{init}. When estimating the epoch time for G and B_G at each p iterations in the trajectory, JABAS finds the best configuration by the configuration

Table 1. GPU clusters used in experiments

Type (# of GPUs)	Index	GPU (# per node)	TFLOPS (FP32)	Memory (MB)
A (16)	1	Tesla V100 (4)	14.13	16160
	2	Tesla P100 (4)	9.52	16280
	3	TITAN RTX (4)	16.31	24220
	4	RTX 3090 (4)	35.58	24260
B (32)	1, 2	GeForce RTX 2060 (4)	6.45	5933
	3, 4	Quadro P4000 (4)	5.30	8119
	5, 6	TITAN V (4)	14.90	12066
	7, 8	TITAN RTX (4)	16.31	24220
C (16)	1,2,3,4	RTX A6000 (4)	38.7	48685
D (16)	1,2,3,4	TITAN Xp (4)	12.1	12196

Table 2. DNN models and datasets used in experiments

Task	Model	Dataset	Optim.	Metric	Target
Image Classification	ResNet50 [21]	ImageNet [13]	SGD	Accuracy (%)	76
	ViT [83]				77
Object Detection	Faster-R-CNN [67]	COCO [46]	SGD	mAP @0.5:0.95	36.6
Machine Translation	GNMT [81]	WMT16 (En-De) [6]	Adam	BLEU	24.61
Language Modeling	BERT-large [14]	Wikipedia [14]	Adam	Validation loss	1.7
Language Modeling	GPT3-XL [8]	OWT* [64]	Adam	Validation loss	3*
Language Modeling	LLaMA2-7B [75]	OWT* [64]	Adam	Validation loss	3*

*OWT denotes OpenWebText dataset. For GPT3-XL and LLaMA2-7B, we set the target loss as lower than the best-reported one due to resource and time constraints.

solver, and then estimates the iteration time with the best configuration. JABAS adds up the estimated iteration times for each p iterations, computing the total epoch time. Among all possible GPU allocations, a GPU allocation with the minimum epoch time is finally selected. If the selected allocation is different from the current one, JABAS checkpoints and restarts the job for the next epoch (similar to Pollux [62]).

7 Experimental Results

7.1 Experimental setups

GPU cluster setups For our experiments, four GPU clusters are used as shown in Table 1. All the clusters are composed of 4 nodes with 16 GPUs in total except cluster B, which consists of 8 nodes with 32 GPUs. Cluster A and B are composed of heterogeneous GPU types; cluster B has higher memory variances than cluster A. Cluster C is a homogeneous GPU cluster, but to train large-scale language models in heterogeneous environment, we adjust the GPU power such that four nodes use 100%, 75%, 50%, and 25% of the GPU power, respectively, having heterogeneous computational capabilities (as similarly done in [44, 47, 50]). We also evaluate JABAS on a homogeneous GPU cluster D. Each node of clusters A and C is equipped with two Intel Xeon Gold 6226R processors (2.90 GHz) and 512 GB DRAM, and is connected with other nodes via 100 Gbps InfiniBand. In the case of clusters B and D, each node is equipped with two Intel Xeon E5-2620v4 processors (2.10 GHz) and 256 GB DRAM, and is connected via 56 Gbps InfiniBand. We implement JABAS by modifying PyTorch 1.8.1 with CUDA 11.1 (for clusters A and C) and 11.0 (for clusters B and D), cuDNN 8.2.1 and NCCL 2.7.8.

Table 3. Convergence results of WDP and WDP+SimiGrad

Model (Target metric)	ResNet-50 (76% Top-1 acc.)	ViT (77% Top-1 acc.)	F-R-CNN (36.6 mAP)	GNMT (24.61 BLEU)
WDP	72	76.64	Diverge	24.19
WDP+SimiGrad	Diverge	1.2	Diverge	19.08

Table 4. The default settings for JABAS and SimiGrad, where the similarity threshold γ, and the lower and upper bound of B_G $(minB_G, maxB_G)$ on Cluster A, B, C and D are given. Adaptive rate $\alpha = 0.1$ and adaptive interval $p = 100$ are used for experiments.

Cluster Type	Model Type	Threshold γ	$(minB_G, maxB_G)$
A	ResNet50	0.1	(256, 32678)
	ViT	0.3	(512, 32678)
	Faster-R-CNN	0.1	(16, 192)
	GNMT	0.1	(512, 4096)
B	ResNet50	0.1	(256, 32678)
	ViT	0.3	(512, 32678)
	Faster-R-CNN	0.1	(32, 192)
	GNMT	0.1	(128, 4096)
C	BERT-large	0.5	(512, 32678)
	GPT3-XL	0.9	(512, 32678)
	LLaMA2-7B	0.5	(32, 2048)
D	GNMT	0.1	(128, 4096)

Benchmarks and metrics Table 2 shows seven DNN models and datasets. Three main metrics measured in our experiments are convergence result, end-to-end training time, and cost. We devise a simple cost model based on the price of the AWS EC2 p3.8xlarge instance [5] composed of 4 V100 GPUs, which has a similar configuration to our node in cluster A. Based on this, we model the price of other types of GPUs used in our experiments on Eq. (8) where C_{A1} is the price of $A1$ instance and X is a GPU type.

$$Cost(X) = C_{A1} * (X.TFLOPS/A1.TFLOPS) \quad (8)$$

Baselines We compare JABAS with three training methods, SimiGrad [63][3], Pollux [62][4], and Pollux-AS [62] [5]. For SimiGrad and Pollux, all the GPUs in a cluster are used for training. Pollux-AS employs a scaling policy where if the current goodput with m GPUs is greater than U fraction of the ideal scaling goodput on a single GPU, it keeps scaling up one more node until the estimated goodput on scaled-up resources is at least L fraction of the ideal scaling goodput on a single GPU. We set U to 2/3 and L to 1/2, and Pollux-AS

[3]We implemented it based on PyTorch 1.8.1 because the original code of SimiGrad [1] was on top of an outdated version of DeepSpeed.
[4]https://github.com/petuum/adaptdl/tree/osdi21-artifact
[5]We implemented the auto-scaling policy based on the goodput in the original paper on top of Pollux's original code.

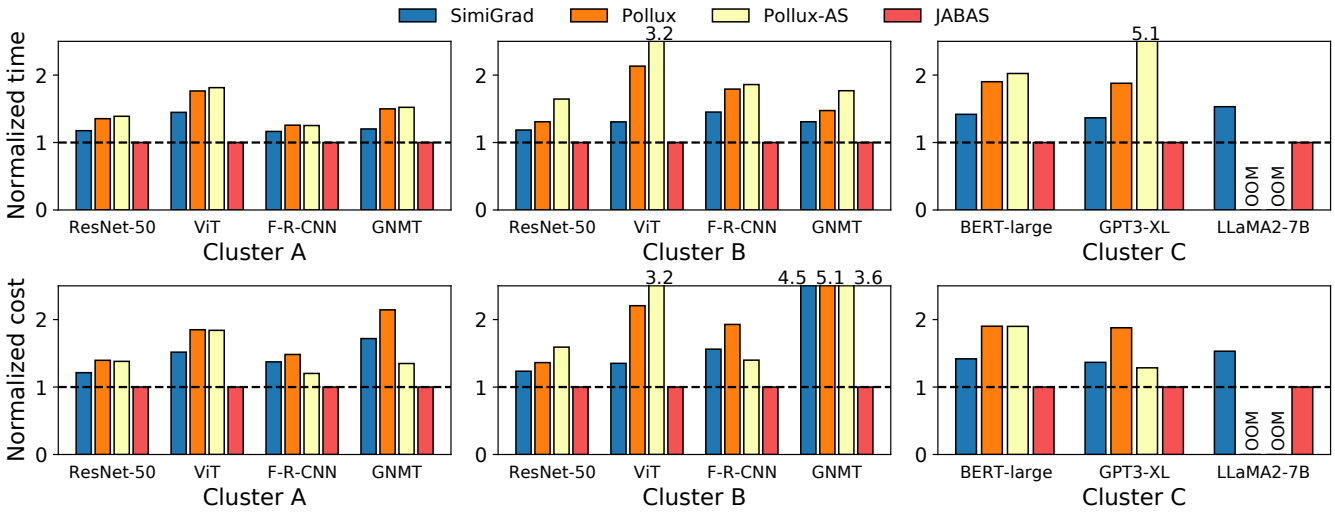


Figure 10. Training time and cost of various DNN models normalized to JABAS. F-R-CNN denotes Faster-R-CNN.

Table 5. Convergence results*

Model	ResNet-50		ViT		F-R-CNN		GNMT	
Cluster	A	B	A	B	A	B	A	B
SimiGrad	76	76	77	77	36.6	36.6	24.61	24.61
Pollux	75.71	76	76.94	77	36.6	36	24.3	22.95
Pollux-AS	75.46	74.68	77	77	36.6	35	24.1	14.55
JABAS	76	76	77	77	36.6	36.6	24.61	24.61

*BERT-large, GPT3-XL, and LLaMA2-7B are omitted as all methods achieve the target loss.

to start at one node with four GPUs as in the original paper [62], allocating the slowest GPU node initially as when fast GPUs are allocated, it tends not to scale. For the methods that do not consider heterogeneous GPUs, we use the same local batch size for all the workers as in DP, i.e., we do not apply WDP, because of serious convergence degradation with WDP as shown in Figure 2. In addition, Table 3 shows the convergence results of WDP and WDP+SimiGrad on cluster A for four DNN models. The result confirms that WDP fails to reach the target accuracy or diverges, and this issue becomes severe with WDP+SimiGrad.

We integrated SimiGrad with our dynamic data sample technique as SimiGrad becomes 2.5 times slower without it. Table 4 shows the experimental settings for JABAS and SimiGrad. The same adaptive global batch size range ($minB_G$, $maxB_G$) is also configured for Pollux and Pollux-AS.

7.2 End-to-End Performance Comparison

On clusters A and B, we compare JABAS with SimiGrad, Pollux, and Pollux-AS for ResNet-50, ViT, Faster-R-CNN, and GNMT. On cluster C, we evaluate three large language models (LLMs): BERT-large, GPT3-XL, and LLaMA2-7B. For LLaMA2-7B, we integrate ZeRO-Offload to JABAS and other baselines to offload some model and optimizer states from GPU memory to CPU memory during training [66]. Despite ZeRO-Offload, Pollux and Pollux-AS cannot train due to out-of-memory (OOM) errors by GNS monitoring.

Figure 10 shows the training time and cost of the DNN models, normalized to those with JABAS on clusters A, B, and C, while Table 5 shows their convergence results. From the results, we can observe the following. (1) JABAS outperforms other baselines in terms of training time and cost for all models in all clusters, while achieving the target accuracy, as it leverages heterogeneous GPUs efficiently via IIDP and performs joint adaptive batching and auto-scaling according to the characteristics of each model. Compared to SimiGrad, which is the second fastest method, JABAS reduces the average training time by 24.7%, 31.3%, and 43.8% on clusters A, B, and C, respectively. Also compared to a training method with the second lowest cost for each model, JABAS saves the average cost by 31.4%, 90%, and 41.1% on clusters A, B, and C, respectively. For ResNet-50 and ViT which are compute-intensive and have small communication overhead on clusters A and B, JABAS configures to use all the GPUs in each cluster except at the early stage of training. Similarly, for the three LLMs on cluster C, JABAS utilizes all the GPUs from start to end. In these cases, JABAS achieves higher performance via IIDP and dynamic VSW configuration than SimiGrad which suffers from stragglers. For GNMT which is communication-intensive, JABAS tends to minimize the use of GPUs with low computational capability in cluster A and GPUs with low memory capacity in cluster B, reducing inter-node communication, while SimiGrad and Pollux use all the GPUs. The average number of GPUs used with JABAS is 9.1 and 7 on clusters A and B, respectively, reducing training cost up to ×5.1 compared to Pollux on cluster B. (2) Compared to cluster A, the performance benefit of JABAS for time and cost tends to be higher in cluster B, which has a larger number of GPUs with higher memory variance. For SimiGrad, the straggler effect of GPUs with

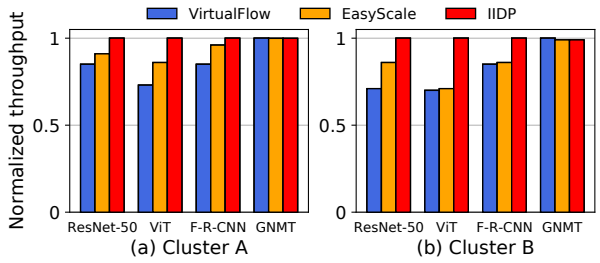


Figure 11. Training throughput normalized to IIDP

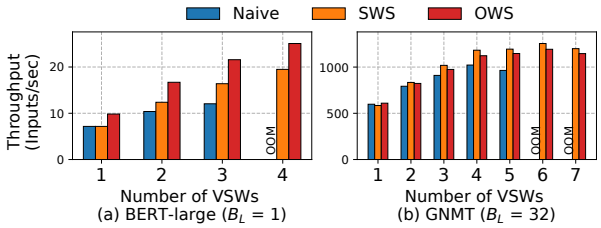


Figure 12. Effects of weight synchronization on IIDP

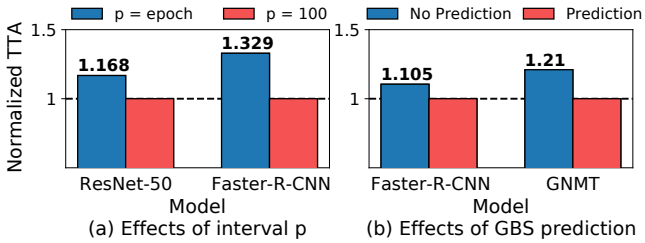


Figure 13. Ablation study

low memory capacity becomes severe in cluster B, especially for Faster-R-CNN which is memory-intensive. Pollux and Pollux-AS significantly increase the global batch size from the beginning to fully utilize a large number of GPUs in cluster B, causing a larger number of iterations per epoch for convergence and consequently higher cost.

To summarize, JABAS provides the best training time and cost, while preserving convergence, for various models over the three heterogeneous clusters, as a job running on IIDP leverages adaptive batching and auto-scaling jointly.

7.3 Evaluation of IIDP

In this section, we compare the performance of IIDP on clusters A and B with two prior techniques that decouple workers from heterogeneous GPUs, VirtualFlow [59] and EasyScale [42]. VirtualFlow allows multiple "virtual nodes" to run sequentially on a single GPU, while configuring a virtual node with a different local batch size and a different number of GA steps depending on a heterogeneous GPU type. VirtualFlow adopts WDP as it uses different local batch sizes for virtual nodes. EasyScale utilizes temporal sharing with context switching of multi-threads called EasyScaleThreads on a GPU for executing multiple workers with the same local batch size. In our experiments, we measure the optimal performance of EasyScale without context-switching overhead via GA [6]. For the same global batch size, the configurations of virtual nodes and EasyScaleThreads are computed by their configuration algorithms in the original papers.

Figure 11(a) and (b) show the throughput with the three decoupling methods, normalized to that with IIDP on clusters A and B, respectively. In the figure, IIDP is 15.9% and 10.4% faster than VirtualFlow and EasyScale, respectively, on average as it improves the parallelism within a GPU by multi-stream. For GNMT, all methods show similar performance. IIDP and EasyScale configure each worker in the same way, while the configuration of VirtualFlow is different but the iteration time is similar to the others. We observe that VirtualFlow's configuration algorithm tends to unbalance the load over the GPUs, even though different local batch sizes are used.

[6]Their experimental details are released on GitHub [2], but the executable code is not released.

7.4 Ablation Study

Effects of weight synchronization on IIDP We measure the throughput of BERT-large and GNMT with a naive method where each VSW has the optimizer, OWS, and SWS over various n_{vsw} on a single node with 4 TITAN RTX GPUs (A3 in Table 1). As shown in Figure 12, the performance of OWS and SWS for BERT-large is up to 79.1%, and 36.1% higher than the naive method, respectively, while that of OWS and SWS for GNMT is up to 19%, and 24% higher, respectively. As the optimizer overhead per iteration for BERT-large is 41% while that for GNMT is 23%, a weight synchronization method shows a stronger effect on the performance of BERT-large than GNMT. In addition, compared to the naive method, OWS and SWS save memory usage, running more VSWs on the GPU without OOM.

Effects of fine-grained adaptive batching We experiment to show the effects of granularity of adaptive batching interval p over two cases with $p = 100$ (default) and $p = epoch$ for ResNet-50 and Faster-R-CNN on cluster A. As shown in Figure 13(a), the time to target accuracy (TTA) with $p = 100$ is shorter than that with $p = epoch$ for both models. We analyze that for Faster-R-CNN, with $p = epoch$, the scalability of a global batch size decreases such that the maximum global batch size used is 48 while that with $p = 100$ is 192. The cost with $p = 100$ is 6.4% and 14.2% less than $p = epoch$ for ResNet-50 and Faster-R-CNN, respectively.

Effects of prediction of global batch size trajectory Figure 13(b) shows the performance of JABAS with and without the trajectory prediction for Faster-R-CNN and GNMT on cluster A. When the prediction is not used, the auto-scaling manager assumes that the last global batch size used in epoch i will be used for epoch $i+1$ and finds the best GPU allocation by only considering that global batch size. Leveraging the

Table 6. Error rate of iteration time on IIDP

Model	ResNet	ViT	F-R-CNN	GNMT	BERT	GPT	LLaMA
Error (%)	5.46	7.68	9	7	7.75	3.3	1.3

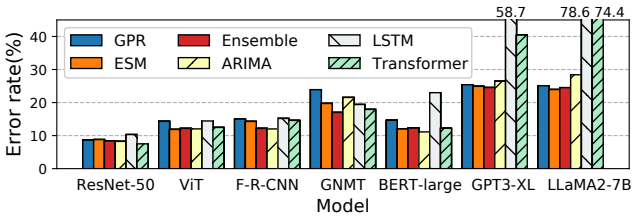


Figure 14. Error rate of GBS prediction models

trajectory prediction based on the ensemble model improves the performance up to 21%. Without prediction, for Faster-R-CNN, JABAS under-provisions GPUs, while for GNMT, it over-provisions GPUs.

7.5 Prediction Error

Iteration time on IIDP Table 6 shows iteration time prediction errors on IIDP for the seven models (5.9% on average), where ResNet, BERT, GPT, and LLaMA denote ResNet-50, BERT-large, GPT3-XL, and LLaMA2-7B, respectively. The performance prediction model in Section 5.1 can provide the estimation of an iteration time with reasonably high accuracy to the configuration solver.

Global batch size trajectory Figure 14 compares the prediction error of GPR, ESM, the GBS forecaster denoted Ensemble, and three time-series forecasting models of one statistical model, ARIMA [7] and two DNN models, LSTM [22] and Transformer [76]. ARIMA is widely used for time-series forecasting as it predicts future values based on past observations by combining the autoregressive model with a moving average [16, 70]. Ensemble shows a prediction error of 15.8% on average over all seven DNN models. In particular, it provides a lower prediction error on Faster-R-CNN and GNMT which have relatively high fluctuations in global batch sizes to the other DNN models. ARIMA and Transformer provide prediction errors similar to Ensemble except for GPT3-XL and LLaMA2-7B, but require a training process for the trajectory, which takes 16.7 and 59.4 times longer than Ensemble on average, respectively. For GPT3-XL and LLaMA2-7B, as the number of samples in the global batch size trajectory is small due to a small number of iterations even for long training time, Transformer and LSTM are unstable to predict.

7.6 Overhead Analysis of JABAS

Figure 15 shows the performance breakdown of JABAS. Overall, the overhead caused by JABAS's components accounts for less than 10% for all DNN models in all clusters. In the figure, the overhead of dynamic VSW configuration where a thread (i.e., VSW) is added or deleted on IIDP is up to 6.43%

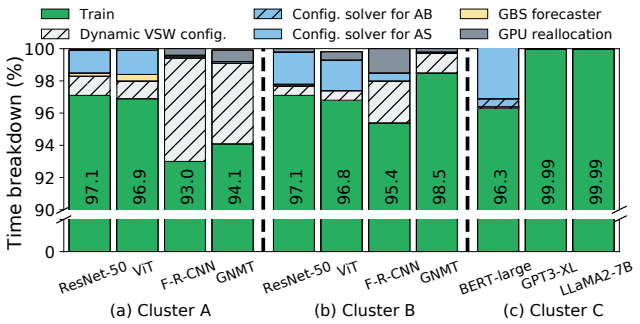


Figure 15. Performance breakdown of JABAS

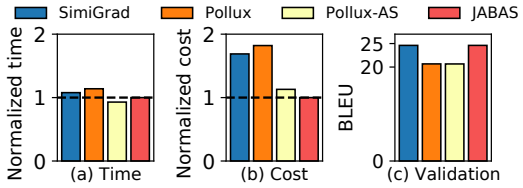


Figure 16. Training results of GNMT on homogeneous GPUs

(for Faster-R-CNN on cluster A), and 3.450%, 1.259%, on average, for all models on clusters A and B, respectively. In the case of Faster-R-CNN, we analyze that PyTorch's CUDA cache allocator overhead accounts for 97% of dynamic VSW configuration overhead. If the checkpointing technique in existing systems [20, 62] is used for dynamic VSW configuration, the training time will increase by 1.65 times on average. The average overhead of the configuration solver is only 1.06%, although the time spent by the auto-scaling (AS) manager (92.6%) is dominant compared to that by the adaptive batching (AB) manager (7.4%). As the complexity of the configuration solver is the quadratic scaling of decoupled workers M in each group (i.e., $O(LM^2N_H)$), we did an in-depth analysis for large M. In all our experiments, the maximum M is 2,592 when training BERT-large, and the overhead of executing the configuration solver for the maximum M is 27.58 seconds. For BERT-large, the total number of invocations of the configuration solver is 4,993, having the overhead of 4.49 hours. However, this overhead is only 3.64% of the whole execution time which is 123.4 hours, as a global batch size gradually increases so that large M only occurs at the end. The other overheads for the GBS forecaster and GPU reallocation, which incurs checkpointing and restarting the job, are negligible, showing 0.1% and 0.452% of the whole execution time, respectively. In the case of GPT3-XL and LLaMA2-7B, they spend almost all of the time training the models and thus, JABAS's overhead becomes insignificant.

7.7 Discussion

Performance on a homogeneous GPU cluster Figure 16 shows the training time and cost of the four training methods,

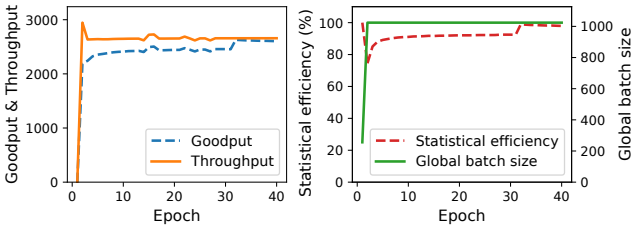


Figure 17. Training ResNet-50 with Pollux

normalized to JABAS for GNMT on the homogeneous GPU cluster D. For GNMT which is communication-intensive, JABAS reduces the cost by 19% compared to the second lowest one with Pollux-AS as JABAS uses 3.38 less GPUs on average. For training time, Pollux-AS is faster than JABAS because Pollux-AS improves training throughput by increasing a global batch size and the number of GPUs significantly at the beginning of training. However, its statistical efficiency decreases, ending up with 4 BLEU degradation.

Issues of a goodput-based adaptive batching system Figure 17 presents the goodput, throughput, statistical efficiency, and global batch size of Pollux over epochs for ResNet-50 on cluster D. In this experiment, the initial global batch size is set to 256. After the first epoch, Pollux estimates that when a global batch size increases to 1024, statistical efficiency will degrade by 75%, but throughput will increase, ending up having higher goodput. Therefore, Pollux increases a global batch size to 1024. This will negatively affect two aspects. First, increasing a global batch size at the beginning of training may incur a negative impact on convergence. Second, since statistical efficiency degrades, Pollux needs to process more iterations. Thus, in the above case, Pollux has to process 3.7 times more iterations than SimiGrad. Furthermore, Pollux needs to monitor GNS values at each iteration, which has the overhead of 14.3% of the total training time for ResNet-50, while that of SimiGrad is 0.1%.

Complexity of the configuration solver In Section 7.6, we observed the overhead of the configuration solver is up to 3.64% in our experimental setup. However, as a global batch size can be larger in a large-scale environment, the overhead of the algorithm may increase. The quadratic scaling complexity $O(LN_HM^2)$ can be reduced to $O(LN_HMlogM)$ by adopting dynamic programming optimization methods such as convex hull optimization [72] and divide-and-conquer [27]. Moreover, Monte Carlo (MC) sampling [16] can be applied to reduce the search space of dynamic programming.

8 Related Work

The GPU multi-stream is one of the promising ways to fully utilize the GPU resources for DNN training and inference. Based on the GPU multi-stream, several techniques have been proposed to execute GPU kernels efficiently on a GPU for a single DNN model [39, 57, 73, 85]. Crossbow [36] devises Synchronous Model Averaging which averages multiple models that process mini-batches of a small local batch size independently within multi-stream.

CoDDL [25] attempts to minimize average JCT with resource-sharing efficiency and short job prioritization. ElasticFlow [20] guarantees the deadline of a DNN training job by elastic resource. Scale-Train [33] reduces the dynamic resource adaptation overhead and adapts an imbalanced batch size with WDP for incoming heterogeneous GPUs. These works for resource elasticity use a fixed global batch size for convergence and configure a local batch size according to the dynamic resource changes.

KungFu [48] is a system to support user-defined adaption policy efficiently. Or et al. [58] proposed a throughput-based auto-scaling engine for DNN training jobs. However, a global batch size can be changed by resource scaling which might harm statistical efficiency without careful consideration. Prior adaptive batching methods, SimiGrad [63] and Pollux [62] are described in Section 2.1. While Pollux focuses on homogeneous GPUs, Sia [28] extends Pollux for heterogeneous GPUs, but only considers the use of homogeneous GPUs for each job and does not consider auto-scaling of a job. ShockWave [84] represents the Nash social welfare theorem and predicts the remaining time of DNN training jobs with adaptive batching to co-optimize efficiency and fairness.

9 Conclusion

In this paper, we presented a novel DNN training system for a heterogeneous GPU cluster, JABAS, which jointly uses adaptive batching and automatic resource scaling. JABAS consists of the IIDP training framework which provides the same theoretical convergence rate of distributed SGD in a heterogeneous GPU cluster, a fine-grained adaptive batching technique with dynamic configuration, which increases or decreases a global batch size based on the gradient similarity metric, and a coarse-grained automatic resource scaling technique. Our experimental results showed that JABAS provides better performance in terms of training time and cost than the state-of-the-art adaptive training techniques without any accuracy loss.

Acknowledgement

We would like to thank our shepherd Divya Mahajan and the anonymous reviewers for their invaluable comments. This research was partly supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), Samsung SDS Co., Ltd., and Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-53. Young-ri Choi is the corresponding author.

References

[1] 2021. https://github.com/SimiGrad/SimiGrad.

[2] 2023. https://github.com/sUntvoOk/EasyScale_info_for_SC23.

[3] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2021. Adaptive gradient communication via critical learning regime identification. In *Proceedings of Machine Learning and Systems (MLSys)*.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[5] AWS. 2023. AWS P3 Instance. https://aws.amazon.com/ec2/instance-types/p3.

[6] Ondrej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, et al. 2016. Findings of the 2016 conference on machine translation (WMT16). In *Proceedings of the First Conference on Machine Translation*.

[7] George EP Box and David A Pierce. 1970. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association* 65.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[9] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*.

[10] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. 2020. Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.

[12] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

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

[15] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*.

[16] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. 2024. Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[17] Fartash Faghri, David Duvenaud, David J Fleet, and Jimmy Ba. 2020. A Study of Gradient Variance in Deep Learning. *arXiv preprint arXiv:2007.04532*.

[18] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of forecasting* 4, 1.

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

[20] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9.

[23] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *proceedings of Advances in neural information processing systems (NeurIPS)*.

[24] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

[27] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[28] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[29] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*.

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

[31] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[32] Tyler B Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. 2020. AdaScale SGD: A Scale-Invariant Algorithm for Distributed Training. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.

[33] Kyeonglok Kim, Hyeonsu Lee, Seungmin Oh, and Euiseong Seo. 2022. Scale-Train: A Scalable DNN Training Framework for a Heterogeneous GPU Cloud. *IEEE Access*.

[34] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-Aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*.

[35] Bobby Kleinberg, Yuanzhi Li, and Yang Yuan. 2018. An alternative view: When does SGD escape local minima?. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[36] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. CROSSBOW: scaling deep learning with small batch sizes on multi-gpu servers. In *Proceedings of the Very Large Data Bases Endowment (VLDB)*.

[37] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[39] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[40] Simon Lacoste-Julien, Mark Schmidt, and Francis Bach. 2012. A simpler approach to obtaining an O (1/t) convergence rate for the projected stochastic subgradient method. *arXiv preprint arXiv:1212.2002*.

[41] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*.

[42] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2023. EasyScale: Elastic Training with Consistent Accuracy and Improved Utilization on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. In *Proceedings of the Very Large Data Bases Endowment (VLDB)*.

[44] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[45] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.

[46] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft COCO: Common objects in context. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[47] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[48] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[49] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*.

[50] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. 2023. SDPipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. In *Proceedings of the Very Large Data Bases Endowment (VLDB)*.

[51] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[52] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*.

[53] NVIDIA corp. . NVIDIA Multi-Instance GPU (MIG). https://docs.nvidia.com/cuda/pdf/MIG_User_Guide.pdf.

[54] NVIDIA corp. . NVIDIA Multi-Processes Service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[55] NVIDIA corp. . NVIDIA NCCL. https://developer.nvidia.com/nccl.

[56] NVIDIA corp. . NVIDIA NCCL error. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html#error-handling-and-communicator-abort.

[57] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-Of-Order BackProp: an effective scheduling technique for deep learning. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*.

[58] Andrew Or, Haoyu Zhang, and Michael Freedman. 2020. Resource elasticity in distributed deep learning. In *Proceedings of Machine Learning and Systems (MLSys)*.

[59] Andrew Or, Haoyu Zhang, and Michael None Freedman. 2022. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. In *Proceedings of Machine Learning and Systems (MLSys)*.

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

[61] Xin Qian and Diego Klabjan. 2020. The impact of the mini-batch size on the variance of gradients in stochastic gradient descent. *arXiv preprint arXiv:2004.13146*.

[62] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[63] Heyang Qin, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2021. SimiGrad: Fine-grained adaptive batching for large scale training using gradient similarity measurement. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[64] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[65] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv preprint arXiv:1804.02767* (2018).

[66] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.

[67] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[68] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

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

arXiv:1802.05799.

[70] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*.

[71] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2019. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research* 20.

[72] S Sherman. 1955. A theorem on convex sets with applications. *The Annals of Mathematical Statistics*.

[73] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. 2019. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[74] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

[75] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[77] Chong Wang, Xi Chen, Alexander J Smola, and Eric P Xing. 2013. Variance reduction for stochastic gradient optimization. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[78] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[79] Christopher Williams and Carl Rasmussen. 1995. Gaussian processes for regression. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*.

[80] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52.

[81] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[82] Qing Ye, Yuhao Zhou, Mingjia Shi, Yanan Sun, and Jiancheng Lv. 2020. DBS: Dynamic batch size for distributed deep neural network training. *arXiv preprint arXiv:2007.11831*.

[83] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. 2021. Tokens-to-Token ViT: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[84] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[85] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*.

A Artifact Appendix

A.1 Abstract

JABAS (*J*oint *A*daptive *B*atching and *A*utomatic *S*caling) is a novel DNN training system for a heterogeneous GPU cluster. Major components of JABAS are a DNN training framework called IIDP, which provides the same theoretical convergence rate of distributed SGD in a heterogeneous GPU cluster, a fine-grained adaptive batching technique with dynamic configuration, and a coarse-grained automatic resource scaling technique that leverages the prediction of global batch size changes for an epoch to auto-scale GPU resources optimally.

A.2 Description & Requirements

A.2.1 How to access. The source code of JABAS is available at https://github.com/unist-ssl/JABAS. The DOI of the artifact is https://zenodo.org/records/13827001.

A.2.2 Hardware dependencies. As JABAS is a distributed DNN training system for a heterogeneous GPU cluster, the hardware environment consisting of heterogeneous GPUs in multiple nodes is needed to show the merit of JABAS. For artifact evaluation, two nodes are required: one composed of four NVIDIA V100 GPUs and the other composed of four NVIDIA P100 GPUs.

A.2.3 Software dependencies. We implement JABAS on PyTorch 1.8.1 with the following software packages:

- Ubuntu >= 16.04
- Anaconda3 4.13.0
- Python 3.8
- NVIDIA driver >= 450.80.02
- CUDA 11.1, cuDNN 8.2.1
- Remote storage (e.g, NFS, AWS S3)

A.2.4 Benchmarks. Four DNN models (ResNet-50, ViT, Faster-R-CNN, GNMT) are provided at `examples/` in the source code repository. In addition, we provide a download script for the corresponding dataset for each DNN model. For artifact evaluation, we only provide a quick-start command to run ResNet-50 on JABAS.

A.3 Set-up

The detailed software installation steps are referred to `README.md` file in the source code repository. After software installation, for artifact evaluation, please refer to the preparation instructions in Section A.4.2, which are also detailed in `examples/resnet50/quickstart/README.md`.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1) The configuration solver based on dynamic programming in JABAS shows the configuration of IIDP. This is proven by experiment (E1).

- (C2) JABAS supports adaptive batching and auto-scaling jointly on heterogeneous GPUs without the accuracy loss. This is proven by experiment (E2) described in Section 7.2 whose results are illustrated in Figure 10 and Table 5.

A.4.2 Experiments.

We provide profile data on the below hardware setup.
Two nodes are required: one composed of four NVIDIA V100 GPUs and the other composed of four NVIDIA P100 GPUs.

Experiment (E1): [Configuration solver] [3 human-minutes + 1 compute-minutes]: It shows the configuration of IIDP with respect to 1) a local batch size 2) the number of Virtual Stream Workers (VSWs) 3) the number of Gradient Accumulation (GA) steps 4) One-way weight synchronization method.

[Preparation] Run the following command on every node:

```
$ conda activate $CONDA_ENV
$ cd $JABAS_HOME/examples/resnet50
$ export NODE0=<V100 node hostname>
$ export NODE1=<P100 node hostname>
$ ./quickstart/scripts/prepare_config.sh
```

[Execution] Run the following command:

```
$ ./quickstart/scripts/run_config_solver.sh
```

[Results] The below result is shown on the screen:

```
[INFO] Solution -
GBS: 128 | LBS: 32 | weight sync method: overlap |
config: ['node0:4GPU,VSW:1,GA:0']
```

The output of the configuration of IIDP on JABAS indicates:

1. Global Batch Size (GBS) is 128.
2. Local Batch Size (LBS) is 32.
3. One-way weight synchronization method on IIDP is overlapping.
4. For each GPU in V100 node, the number of VSWs is 1, and the number of GA steps is 0.
5. P100 node is not used.

Experiment (E2): [Train ResNet-50 on JABAS] [3 human-minutes + within 1 compute-day]: It shows the performance in terms of training time and cost while achieving the target accuracy (76%).

[Preparation] To run an experiment (E2), the ImageNet dataset pre-processed for PyTorch must be prepared.
1. Activate conda environment on every terminal.

```
$ conda activate $CONDA_ENV
```

2. Set remote storage (e.g., NFS) and dataset storage paths on every node.

```
$ export JABAS_REMOTE_STORAGE=<remote storage path>
$ export JABAS_DATA_STORAGE=<dataset storage path>
```

3. Prepare the ImageNet dataset on every node (it will take 3-5 days).

- Download the ImageNet dataset to `JABAS_DATA_STORAGE` by the following URL: http://www.image-net.org/
- Run the following command:

```
$ cd $JABAS_HOME/examples/resnet50
$ ./scripts/utils/prepare_dataset.sh
```

4. Prepare configuration for JABAS on every node. Skip this step if you have already completed experiment (E1).

```
$ export NODE0=<V100 node hostname>
$ export NODE1=<P100 node hostname>
$ ./quickstart/scripts/prepare_config.sh
```

[Execution] Run the following command:

- On V100 node (terminal 0):

```
$ ./quickstart/scripts/run_elastic_agent.sh
```

- On V100 node (terminal 1):

```
$ ./quickstart/scripts/run.sh 0
```

- On P100 node (terminal 0):

```
$ ./quickstart/scripts/run.sh 1
```

[Results] Total train time and cost are shown on the screen. The log file is saved to `JABAS_REMOTE_STORAGE`.

A.5 Notes on Reusability

For general benchmark usage on different hardware environments, the common setup is explained in `examples/README.md`. To run a different DNN model on JABAS, please refer to `README.md` file in each DNN model's directory (e.g., `examples/vit/README.md`).