Cannikin: Optimal Adaptive Distributed DNN Training over Heterogeneous Clusters

Chengyi Nie
Stony Brook University
Stony Brook, NY, USA
nie.chengyi@stonybrook.edu

Jessica Maghakian
Stony Brook University
Stony Brook, NY, USA
jessica.maghakian@stonybrook.edu

Zhenhua Liu
Stony Brook University
Stony Brook, NY, USA
zhenhua.liu@stonybrook.edu

ABSTRACT

Adjusting batch sizes and adaptively tuning other hyperparameters can significantly speed up deep neural network (DNN) training. Despite the ubiquity of heterogeneous clusters, existing adaptive DNN training techniques solely consider homogeneous environments. Optimizing distributed DNN training over heterogeneous clusters is technically challenging, and directly adapting existing techniques results in low utilization and poor performance. To solve this problem, we introduce Cannikin – a novel data-parallel distributed training system. Cannikin achieves efficient and near optimal performance by accurately modeling the optimal system performance and predicting adaptive batch size training metrics for DNNs in heterogeneous clusters. We implemented Cannikin in PyTorch and conducted experiments over 16 GPUs in Chameleon. Empirical results show that Cannikin reduces DNN training in heterogeneous clusters by up to 52% compared to the state-of-art adaptive training system and up to 85% compared to native PyTorch DistributedDataParallel.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**.

KEYWORDS

Distributed DNN training, Heterogeneous system

ACM Reference Format:
Chengyi Nie, Jessica Maghakian, and Zhenhua Liu. 2024. Cannikin: Optimal Adaptive Distributed DNN Training over Heterogeneous Clusters. In *24th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700767

1 INTRODUCTION

With the explosive increase of deep learning (DL) applications in fields such as image classification [12, 27], natural language processing [46, 53], and recommender systems [18, 59], the demand for deep neural network (DNN) training resources is doubling every six months [49]. In order to achieve efficient DNN training, practitioners rely on accelerators [24, 36, 60], hyper-parameter tuning [33], and distributed training [3, 30]. Meanwhile, the short hardware update cycle results in newly released accelerators that significantly outperform previous models within a short time [47]. Table 1 shows the evolution of NVIDIA data center GPUs released in recent years. Each new flagship model is over two times faster than the preceding flagship data center GPU. When companies and research institutes upgrade their machine learning systems, newly released GPUs are installed before older models retire, so homogeneous environments cannot always be guaranteed when running distributed training jobs. Specialized methods are required to enhance the utilization of computing resources and speed up DNN model training in heterogeneous environments.

Table 1: Evolution of NVIDIA data center GPUs

Model	Year	Archit.	CUDA Cores	Memory (GB)	FP16 (TFLOPS)
Tesla P100	2016	Pascal	3584	16	21.2
Tesla V100	2017	Volta	5120	16/32	31.4
A100	2020	Ampere	6912	40/80	77.97
H100	2022	Hopper	16896	80	204.9

Previous work on specialized distributed training for heterogeneous environments focuses on two major schemes: data parallelism [30, 48] and model parallelism [40, 50]. For data-parallelism heterogeneous distributed training systems, HetSeq [15] manually tunes the local mini batch size for each node to balance workloads, while LB-BSP [8] and DLB [55] improve performance by iteratively tuning the workloads assigned to each worker based on the computing time of each node. On the other hand, BlueConnect [10] boosts data-parallelism distributed training by optimizing communication. Existing data-parallelism systems do not jointly consider the computing and communication models for heterogeneous clusters, resulting in suboptimal performance. Model-parallelism systems [17, 40] pipeline the DNN model in heterogeneous clusters, which is near optimal for resource utilization with fine-tuning. However, model parallelism requires a specific configuration of each node in a cluster, thus limiting scalability. Furthermore, existing model-parallelism and data-parallelism methods cannot manage the sudden changes of resources that occur in clusters with dynamic resource allocation [43, 45].

Another efficient DNN training method, adaptive batch size training, tunes hyper-parameters such as batch size and learning rate during training, significantly speeding up convergence. Previous work [13, 32, 33, 45] focuses on adaptive batch size training in homogeneous environments. These approaches continuously

Middleware '24, 2024, Hong Kong, China

ACM ISBN 979-8-4007-0623-3/24/12…$15.00
https://doi.org/10.1145/3652892.3700767

monitor system metrics and optimize the batch size according to their adaption policies. However, the adaptive batch size metrics, algorithms, and adaption policies are all designed for homogeneous environments. Directly adopting existing methods in heterogeneous clusters will cause a large margin of error for adaptive batch size training metrics measurement and prediction. To the best of our knowledge, no specialized adaptive training system for heterogeneous environments has been developed so far.

There are three main challenges in designing an automatic near-optimal training system. First, each node has a different performance model in heterogeneous clusters, which causes complexity in predicting the optimal performance and the corresponding cluster configuration. The system overhead will significantly affect the training performance for larger clusters. Second, considering data parallelism distributed training in heterogeneous clusters, given a new total batch size to the cluster in the adaptive batch size training, the optimal configuration of each node will change. Given the total batch size, previous work [8, 55] iteratively tunes each node's configuration to approach the optimal performance, which is inefficient in adaptive batch size training. Third, in heterogeneous data parallelism training systems, different local batch sizes are assigned to different GPUs. This introduces challenges in accurately modeling gradient noise [33] across the heterogeneous clusters.

Based on these insights and challenges, we study the performance model of data-parallel distributed training for heterogeneous GPU clusters and propose the optimal performance *OptPerf*. For the prediction of *OptPerf* and the corresponding configuration, we design Cannikin, an efficient near-optimal data-parallel distributed training system. Taking both computing and communication models into consideration, Cannikin has accurate modeling and prediction of the performance models for DL in heterogeneous clusters. With the cluster performance model learned online, Cannikin can predict *OptPerf* and configuration with low overhead when the cluster is given a new total batch size. Cannikin optimizes the measurement of system parameters using inverse variance weighting of different observations from each node in the cluster. We also prove that in heterogeneous clusters, we can correctly model adaptive batch size training metrics, just like systems [33, 45] designed for homogeneous clusters. While developed for single DNN training, Cannikin can be readily integrated with adaptive batch size training engines [32, 45] and dynamic resource allocation schedulers [43, 45] for multiple jobs. The main contributions of this paper are:

- We are the first to consider adaptive batch sizes data parallelism DNN training in heterogeneous clusters.
- We deduce and predict the optimal performance, denoted as *OptPerf*, along with its corresponding configuration and the optimal total and local batch sizes during DNN training in heterogeneous clusters.
- We design Cannikin that can be easily integrated with the state-of-the-art adaptive batch size training systems to train the DNN models in heterogeneous clusters optimally.
- We evaluate the performance of Cannikin in two heterogeneous clusters using multiple popular DNN models. Results highlight that Cannikin reduces DNN training by up to 52% and 85% in heterogeneous clusters compared to AdaptDL [45] and PyTorch DistributedDataParallel (DDP) [30].

2 BACKGROUND

2.1 Adaptive Batch Size Training

A deep learning model usually consists of millions to trillions of parameters [5, 19, 51], requiring a long time for training. The selection of hyperparameters, such as batch size in different convergence phases, significantly impacts training efficiency. Recent work on adaptive batch size training [13, 32, 33, 45] greatly speeds up deep learning model training by dynamically tuning hyperparameters such as batch size and learning rate according to the gradient noise [33], data throughput, and other customized metrics.

To determine the most statistically efficient batch size for a training iteration, McCandlish et al. [33] propose the *gradient noise scale* (GNS), an estimation of the signal-to-noise ratio of the stochastic gradient. This metric can be used to predict the most statistically efficient batch size during training. When the gradient noise is low, a small batch size can achieve a great contribution to the convergence. When the gradient noise is large, the error of the gradient estimated by a small batch would be significant. In this situation, a larger batch size could reduce the training time with little reduction of statistical efficiency.

However, the optimal convergence progress is not guaranteed by using the most statistically efficient batch size because the most statistically efficient batch size often comes with relatively low data throughput. Pollux [45] introduces *goodput*, the product of the system throughput and statistical efficiency modeled by the GNS. Goodput optimizes training by balancing the trade-off between data throughput and statistical efficiency.

2.2 Distributed DNN Training

DNN training is computing-intensive [57]. Distributed deep learning accelerates DNN model training with multiple GPUs by either using model parallelism or data parallelism.

Model parallelism Model parallelism is a technique that distributes the DNN training across multiple nodes by splitting the DNN model itself. In heterogeneous clusters, model parallelism improves the throughput by splitting DNN models according to each node's computing and communication status, hence reducing the straggler effect. Model parallelism usually incorporates pipeline parallelism to improve the system throughput further.

However, adaptive batch-size training is challenging with model parallelism. The changing global batch size during training could affect the optimal model partition. Furthermore, adaptive batch size training requires the GNS [33] to update the batch size. In a model parallel setup, the model is split across multiple devices, and each device computes gradients only for its portion of the model. This can make it challenging to estimate the GNS in the typical way. This paper focuses on data-parallelism distributed learning. Each GPU contains the full DNN model but uses different data samples.

Data parallelism In data-parallelism distributed training, node i first trains its local mini batch by forward and backward passes for the local gradient estimation. Node i's local gradient g_i is:

$$g_i = \frac{1}{b_i} \sum_{j=0}^{b_i-1} \nabla_\theta L_{x_j}(\theta), \tag{1}$$

where θ is the vector of DNN weights, b_i is node i's local mini-batch size, x_j is a sample in local mini batch, and L is the loss function.

Upon the completion of the local gradient estimation, node i will aggregate its own gradient with all the local gradients calculated by other nodes in the cluster to get the gradient of the total batch (full batch of the DNN model):

$$g = \frac{1}{N} \sum_{i=0}^{N-1} g_i, \tag{2}$$

where N is the count of total nodes (GPUs) that train the model. Finally, each node will use the averaged gradient to update the weight parameters $\mathbf{w}$ for the next batch. In a DNN training system, the gradient averaging can be handled using backends such as NCCL [37], MPI [1], and Gloo [21], etc. Data parallelism is suitable for estimating GNS, as it facilitates the collection and aggregation of gradients across multiple mini-batches processed in parallel.

3 THE *OptPerf* OF GPU CLUSTERS

To improve the performance of a cluster, first we need to know the optimal performance the cluster can achieve. In this section, we define and deduce the optimal performance *OptPerf* of a heterogeneous GPU cluster using the metrics collected from each GPU in a general case.

3.1 The Definition of *OptPerf*

Table 2: The notation table of *OptPerf*

Notation	Definition
B	The global batch size of DNN training
b_i	The local batch size of node i
n	Total number of GPUs within the cluster
$\mathcal{N}$	The set of GPUs in the cluster
$t_i^{b_i}$	batch processing time without gradient synchronization
r_i	Ratio of local batch size to the global batch size of node i
$\mathbf{r}$	List of local batch size ratio of all GPUs in $\mathcal{N}$
T	Batch processing time
$t^i_{compute}$	The batch computing time
a_i	Parameter updating, data loading, and forward propagation time
P_i	The backpropagation time of a batch
T_{comm}	The gradient synchronization time of a batch
T_u	The last gradient bucket synchronization time
T_o	The gradient synchronization time for all the other gradient buckets
γ	Ratio of first bucket computing time to total backpropagation time
syncStart	The first bucket's ready-for-synchronization point

For data-parallelism distributed training with a given batch size B, *OptPerf* is the optimal batch processing time a heterogeneous GPU cluster can achieve by ideally tuning each node's local mini-batch size. Consider heterogeneous GPU Cluster A with a set of nodes $\mathcal{N}$, $|\mathcal{N}| = n$, note that the parameters are defined in Table 2. Due to heterogeneity, we assume there exist nodes $i \neq j$ that $t_i^b \neq t_j^b$ with the same b. The local mini batch sizes satisfy $\sum_{i\in\mathcal{N}} b_i = B$. In this paper, we only consider synchronized data-parallelism distributed training, meaning all nodes synchronize their gradient after each batch. For this method, fast nodes in the cluster always wait for the stragglers to finish local gradient estimation, hence the batch processing time $T = max\{t_0^{b_0}, t_1^{b_1}, \ldots, t_{n-1}^{b_{n-1}}\}$. We can infer that there exists an optimal local mini batch size ratio $\mathbf{r}_{\text{opt}}$ that minimizes the batch processing time T. We define the minimized batch processing time to be *OptPerf*. To determine *OptPerf* and $\mathbf{r}_{\text{opt}}$ for heterogeneous cluster A with total batch size B, we model the performance of GPUs in the cluster as a function of T, B, and $\mathbf{r}$.

3.2 Performance Modeling

When considering the performance model of a heterogeneous cluster, rather than simultaneously modeling the performance of heterogeneous nodes, we can instead model the performance of each GPU separately and then combine all the GPU performance models to determine the cluster's performance. In data parallel distributed training, the batch processing time comprises the computing time for local gradient estimation and the communication time for gradient synchronization across nodes.

3.2.1 Computing Time. The batch computing time can be separated into data loading, forward propagation, backward propagation and parameter updating. For any node i, $t^i_{compute}$ is a linear function of local batch size b_i [31, 45]. Furthermore, the parameter updating time remains constant regardless of variations in the local batch size. In contrast, data loading time, forward propagation time, and backpropagation time exhibit a linear relationship with b_i. So for all nodes in Cluster A, the computing time can be expressed as:

$$\begin{aligned} t^i_{compute} &= a_i + P_i, \ \forall i \in \mathcal{N}, \\ a_i &= q_i b_i + s_i, \ \forall i \in \mathcal{N}, \\ P_i &= k_i b_i + m_i, \ \forall i \in \mathcal{N}, \end{aligned} \tag{3}$$

q_i, s_i, k_i, and m_i are coefficients related to GPU types and DL jobs. Note that within a heterogeneous GPU cluster, different GPU models exhibit varying pairs of q_i and s_i, as well as k_i and m_i, even when performing the same DL job. If cluster A has n different types of GPUs, there are n different pairs of linear functions corresponding to each type of GPUs.

3.2.2 Gradient Synchronization Time. We focus on the ring All-reduce mechanism [41] adopted by Pytorch DistributedDataParallel [30]. Ring All-reduce is a synchronized communication method that starts the gradient synchronization when all nodes are ready to synchronize and ends the synchronization when all nodes finish the gradient synchronization. T_{comm} is dependent on model size (size of gradient parameters) and network status. In the scenario that the network and allocated resources are stable in a heterogeneous cluster, even though each node's network performance varies, T_{comm} is a learnable constant when we train the same job with different batch sizes.

3.2.3 Computing and Communication Overlap. Modern distributed training frameworks [30, 48] support the overlap between gradient computing and synchronization by separating the locally-computed gradients into buckets [30]. Rather than synchronizing after all nodes finish computing the full gradient at the end of backpropagation, each gradient bucket starts synchronization when all nodes finish computing the gradient of the same bucket. In batch processing, only the last bucket cannot overlap its synchronization with its gradient computation. So Cluster A's per batch gradient synchronization time T_{comm} is the sum of T_u, the last gradient bucket synchronization time, and T_o, the gradient synchronization time for all the other gradient buckets: $T_{comm} = T_o + T_u$.

For all nodes in Cluster A, the gradient size (model size) is determined when training starts. Different types of GPUs have the same gradient computing procedure [2]. Even though the local batch size of each node is different, the gradient bucket will be ready for synchronization at a fixed proportion of each node's backpropagation time P_i. We can infer that the starting point of the gradient synchronization of node i is:

$$syncStart_i = a_i + \gamma P_i, \tag{4}$$

where the overlap ratio γ is the ratio of the first bucket computing time to the total backpropagation time. The first bucket computing time is the period from the start point of the backward pass to the first bucket ready for synchronization point. The first bucket computing cannot be overlapped with the gradient synchronization.

Although varying b_i changes P_i, T_{comm} is a fixed, learnable constant. So the computing and communication overlap pattern differs when the local mini batch size varies. The first gradient bucket's ready-for-synchronization point is determined by the computing time along with γ, which is a constant that can be accurately measured through all the nodes in the cluster. The following buckets' synchronization can be blocked by the previous buckets' synchronization. To eliminate the effect of measurement error and system overhead, we only measure *syncStart* and assume all buckets' computing time and communication time are evenly distributed in the rest of the gradient computing time and communication time. There are two possible overlap patterns of computing and communication.

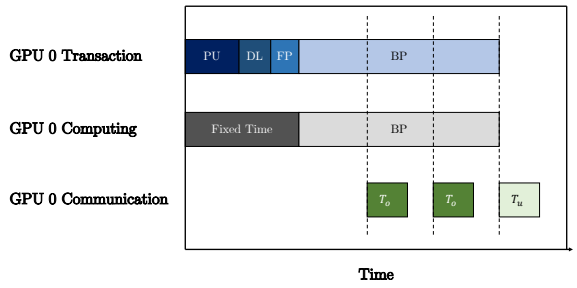

Figure 1: A node running in the computing-bottleneck situation. PU, DL, FP, and BP are the parameter update, data loading, forward propagation, and backpropagation.

When computing is the bottleneck. Figure 1 shows a computing-bottleneck node. The dashed lines show the buckets' ready-for-synchronization points. When $(1-\gamma)P_i \geq T_o$, node i's bottleneck is the gradient computation. In this case, the gradient synchronization of each bucket finishes before the next gradient bucket is ready for synchronization, so T_o fully overlaps with the gradient computing. The total processing time of one batch for node i in cluster A is:

$$T = t^i_{compute} + T_u. \tag{5}$$

When communication is the bottleneck. Figure 2 shows the communication-bottleneck pattern. If $(1-\gamma)P_i < T_o$, node i's bottleneck is the gradient communication. Although the unfinished bucket synchronization won't block gradient computing (due to the All-reduce mechanism), the synchronization of previous unfinished

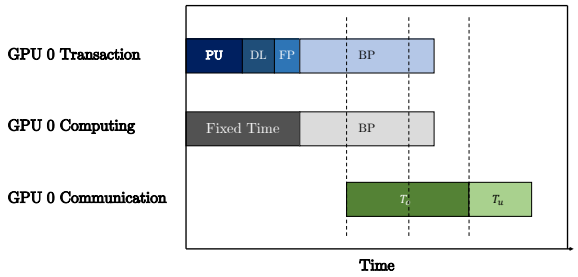

Figure 2: A communication-bottleneck node.

buckets will block the synchronization of the following bucket. In this situation, the total processing time of one batch for node i is:

$$T = syncStart_i + T_{comm}. \tag{6}$$

3.3 Expression of OptPerf

To predict *OptPerf* for heterogeneous clusters, with learned performance models of all GPUs, first we deduce cluster A's batch processing time T. In cluster A,

$$T = \max\left\{\max_{i\in\mathcal{N}}\{t^i_{compute} + T_u\}, \max_{i\in\mathcal{N}}\{syncStart_i + T_{comm}\}\right\}. \tag{7}$$

Since each node's bottleneck is unknown, minimizing T is a mixed integer linear programming problem. Rather than solve an NP-hard problem, we instead provide the criteria for different situations to achieve *OptPerf*.

When optimizing the performance of Cluster A, the batch processing time T_i of stragglers during training can be reduced by adjusting the local mini batch sizes for all nodes. Intuitively, the cluster's fast nodes should be assigned larger local mini batches, while the stragglers should have smaller batches. With the computing and communication overlap patterns of each node in Section 3.2.3, we first look into two special scenarios.

All nodes are computing-bottleneck. Since $(1-\gamma)P_i \geq T_o$, $\forall i \in \mathcal{N}$, all the nodes' performance models are Equation (5). *OptPerf* is achieved when all nodes in cluster A have the same computing time $t_{compute}$. The proof is in Appendix A.1.

All nodes are communication-bottleneck. If $(1-\gamma)P_i < T_o$, $\forall i \in \mathcal{N}$, all nodes' performance models are Equation (6). *OptPerf* will be achieved when all nodes start the first gradient bucket synchronization at the same time. The proof is in Appendix A.2.

The general case. In the general case, some nodes' bottlenecks are computing while others' are communication (see Figure 3). The computing-bottleneck nodes' performance models follow Equation (5), and the communication-bottleneck nodes' performance models follow Equation (6). *OptPerf* is achieved when all computing-bottleneck nodes have the same computing time $t_{compute}$ and all communication-bottleneck nodes start the first bucket synchronization simultaneously. Moreover, the computing and communication bottleneck nodes simultaneously get ready for the last bucket synchronization. The proof is located in Appendix A.3. With *OptPerf*, we can determine the optimal performance of a heterogeneous

cluster with different batch sizes using the parameters measured and learned during training.

4 SYSTEM DESIGN OF CANNIKIN

In this section, we give details on the workflow of Cannikin and describe how Cannikin configures the cluster before the start of each epoch, optimizes the measurement of learnable parameters, guarantees the gradient quality, and how Cannikin integrates with existing adaptive batch size training systems.

4.1 Workflow of Cannikin

Cannikin optimizes the goodput [45] for each training epoch, which is the product of the system throughput and its corresponding convergence efficiency. We search for *OptPerf* to optimize the system throughput for heterogeneous GPUs in Section 4.2 and Section 4.3 and deduce the GNS for heterogeneous GPUs to predict the convergence efficiency in Section 4.4.

Figure 4 shows the overview of the workflow of Cannikin. In a heterogeneous environment, after a user submits a training Job J to the dynamic resource job scheduler, the job scheduler allocates a number of (possibly heterogeneous) GPUs to form Cluster A to initialize Job J. Before the start of each epoch, the adaptive batch size engine enumerates the total batch size candidates from the batch size range [45]. The optimizer uses performance models learned by the analyzer to predict *OptPerf* with its corresponding total batch size and $\mathbf{r}_{\text{opt}}$ for the next training epoch, then loads each node's local mini batch based on $\mathbf{r}_{\text{opt}}$ and starts the next training epoch. During the training epoch, each node continually collects performance metrics and learns the performance models locally. After each training epoch, the analyzer gathers the updated performance models of all nodes. Figure 5 shows the batch size of each node during the training of CIFAR-10. During the training, the local batch size of each GPU will increase during training due to the increase in global batch size. However, $\mathbf{r}_{\text{opt}}$ varies with different global batch sizes because the bottleneck changes from communication to computing with the increase of local batch size.

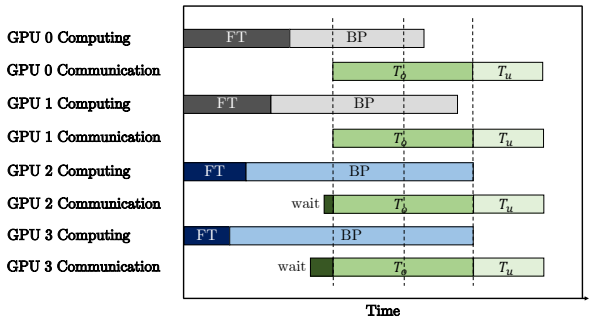

Figure 3: An example of the general case, where GPU 0 and GPU 1 are communication-bottleneck, GPU 2 and GPU 3 are computing-bottleneck. FT is the abbreviation of fixed time.

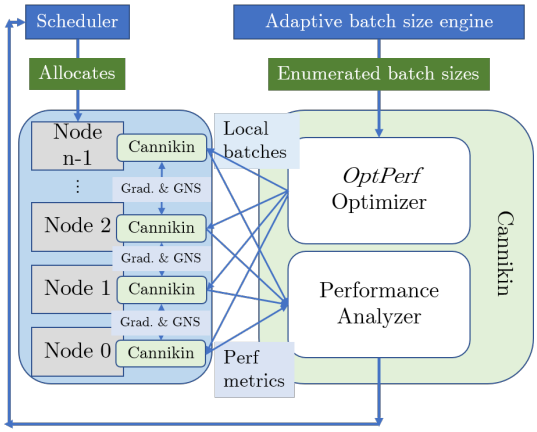

Figure 4: The overall workflow of Cannikin.

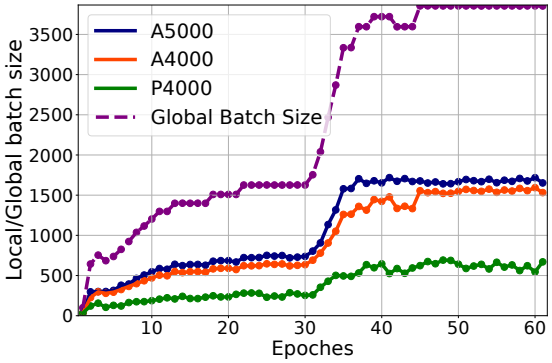

Figure 5: Global batch size and local batch size of each node during CIFAR-10 training.

4.2 *OptPerf* Optimizer

Cannikin continuously learns the computing and communication models for all nodes during the training process. Despite having performance models for all nodes, the overlap state of each node in the cluster remains unknown as it is contingent on the total batch size. For example, a larger total batch size can lead to more nodes experiencing computational bottlenecks. To address this challenge, we have developed a novel search algorithm to unveil the overlap state for all nodes across the cluster.

Determine the overlap state. Given an enumerated total batch size, Cannikin uses Algorithm 1 to determine the overlap state, then predict *OptPerf* with $\mathbf{r}_{\text{opt}}$ for each node.

If all nodes are computing or communication-bottleneck, we can use Check 1 and Check 2 to verify. However, when nodes are mixed-bottleneck, the overhead of the enumeration method to determine each node's overlap pattern is relatively high. Algorithm 1 addresses this issue in the following steps: If node i is a computing (communication) bottleneck node in check 1 and check 2, then node i is also a computing (communication) bottleneck node in the mixed-bottleneck situation. For all other outliers that have different

Algorithm 1: Overlap state and *OptPerf* configuration

Input: Total batch size $B = \sum_{i=0}^{n-1} b_i$.
Given: $\gamma, T_o, T_u, \{a_i, k_i, m_i\}, i = 0, 1, \ldots, n-1$.
▷ Overlap ratio, two parts of communication time, computing coefficients.
/*Check 1: All nodes are computing-bottleneck*/
Solve $t_{compute} = t^0_{compute} = t^1_{compute} = \cdots = t^{n-1}_{compute}$.
if $\forall (1-\gamma)P_i \geq T_o$ **then**
  /*If all nodes are computing-bottleneck. */
  $OptPerf = t_{compute} + T_u$;
  return $OptPerf, b_i, i = 0, 1, \ldots, n-1$.
  break
/*Check 2: If nodes are communication-bottleneck.*/
Solve $syncStart = syncStart_0 = \cdots = syncStart_{n-1}$.
if $\forall (1-\gamma)P_i < T_o$ **then**
  /*All nodes are communication-bottleneck.*/
  $OptPerf = syncStart + T_{comm}$;
  return $OptPerf, b_i, i = 0, 1, \ldots, n-1$.
  break
/*The nodes are mixed-bottleneck.*/
while $(\exists\ syncStart_i > syncStart') \vee (\exists\ t_{compute} > t'_{compute})$ **do**
  /*Search the overlap state.*/
  $beg = outlier_{min}$;
  $end = outlier_{max}$;
  $C = (beg+end)/2$;
  /*Node 0 to Node $C-1$ are computing-bottleneck, Node C to Node $n-1$ are comm-bottleneck.*/
  Solve $T_{comb} = t'_{compute} = syncStart' + T_o$.
$OptPerf = T_{comb} + T_u$.
return $OptPerf, b_i, i = 0, 1, \ldots, n-1$.

overlap patterns in check 1 and check 2, we use a binary-search-like algorithm to determine the computing and communication bottleneck nodes. We rank all the intermediate nodes in increasing order based on the fixed processing time and then set a hypothetical bottleneck boundary node that separates the computing-bottleneck and communication-bottleneck nodes. For any overlap state, the mixed-bottleneck *OptPerf* solver from Section 3.3 will indicate $(\forall\ syncStart \leq syncStart') \wedge (\forall\ t_{compute} \leq t'_{compute})$ if the overlap pattern is correct. Thus we can iteratively set the middle element to be the boundary node until we find the correct overlap pattern.

Since the time complexity of checks 1 and 2 is at most $O\left((n+1)^3\right)$ while solving linear equations [6] and the time complexity of the mixed-bottleneck search algorithm is at most $O(\log n)$, Algorithm 1 is $O\left((n+1)^3 \log n\right)$. In Section 4.5, we improve the time complexity of Algorithm 1 to $O\left((n+1)^3\right)$.

Approaching *OptPerf* when no available performance models. As the computation time scales linearly with the local batch size of each GPU, deriving the computing time models a_i, P_i to b_i necessitates the execution of at least two distinct local batch sizes per GPU. Consequently, during the initial two epochs, there will be no available performance model to predict *OptPerf*. In this scenario, we employ the inverse proportion of the sample computation time for each node to determine their respective local batch sizes for the next epoch. Assume the per sample computing time of node i at the previous epoch is $t^i_{sample} = \frac{t^i_{compute}}{b^i_{current}}$, where $b^i_{current}$ is the local batch size of node i in the previous epoch. The local batch size of node i for the next epoch can be expressed as:

$$b^i_{next} = \frac{\sum_{i\in\mathcal{N}} t^i_{sample}}{t^i_{sample}} \left(\sum_{i\in\mathcal{N}} \frac{\sum_{i\in\mathcal{N}} t^i_{sample}}{t^i_{sample}} \right)^{-1} B, \tag{8}$$

where B is the total batch size for the upcoming epoch. With this method, each node can experiment with various local mini-batch sizes necessary for performance model learning, and Cannikin iteratively approaches *OptPerf* when no available performance models. Note that the primary purpose of this method is to adjust each node's local batch size for performance model learning, rather than relying on the less efficient iterative method to find *OptPerf*. Once the performance models are established, they are employed to predict *OptPerf* before each epoch.

4.3 Optimized Gradient Aggregation

In homogeneous environments, the cluster can aggregate the local gradient of each node via averaging. This procedure guarantees each training sample has an identical weight in the global gradient after synchronization. However, local gradient averaging cannot be utilized for adaptive local batch training in heterogeneous clusters due to the variety of local batch sizes. Simply averaging each node's local gradient results in over-representation of training samples from smaller local batches in the global gradient. To address this problem, LB-BSP [8] introduced proportional-weighted gradient aggregation. By weighting each local gradient proportionally to the local batch size, samples assigned to different nodes have identical weights in the global gradient. Cannikin computes the global gradient g using:

$$g = \sum_{i\in\mathcal{N}} r_i g_i, \tag{9}$$

where g_i is the local gradient in Equation (1) and r_i is the local mini batch ratio of node i. For *i.i.d.* data, g is equivalent to the averaged gradients in homogeneous environments.

4.4 Gradient Noise Scale in Heterogeneous Environment

Adaptive batch size training uses the gradient noise scale (GNS) [33] to model the convergence efficiency (statistical efficiency). The GNS, $\mathcal{B}_{noise}$, measures how large the gradient is compared to its variance: $\mathcal{B}_{noise} = \mathrm{tr}(\Sigma)/|G|^2$, where Σ is the covariance matrix and G is the noiseless true gradient. Since $\mathrm{tr}(\Sigma)$ and $|G|^2$ are not available in practice, standard methods instead rely on good estimators of these two quantities. Previous work has only considered how to estimate $\mathrm{tr}(\Sigma)$ and $|G|^2$ (and thus $\mathcal{B}_{noise}$) in homogeneous clusters.

To compute the GNS, we first construct local estimates $\mathcal{G}_i$ and $\mathcal{S}_i$ of $|G|^2$ and $\mathrm{tr}(\Sigma)$ for each node i:

$$\mathcal{G}_i = \frac{1}{B-b_i}(B|g|^2 - b_i|g_i|^2), \quad \mathcal{S}_i = \frac{b_i B}{B-b_i}(|g_i|^2 - |g|^2) \tag{10}$$

where the estimates incorporate local and global gradient information. For any batch of size b, the expected gradient norm $\mathbb{E}[|g_{est}|^2]$ satisfies the equality $\mathbb{E}[|g_{est}|^2] = |G|^2 + \frac{1}{b}\mathrm{tr}(\Sigma)$ [33]. Using this

equation, we can prove that $\mathcal{G}_i$ and $\mathcal{S}_i$ are unbiased estimators of $|G|^2$ and $\text{tr}(\Sigma)$, respectively. Aggregating the local estimates $\mathcal{G}_i$ and $\mathcal{S}_i$ across all nodes can provide high quality, unbiased estimates of $|G|^2$ and $\text{tr}(\Sigma)$ that have improved, lower variance. The variance of these estimators is crucial since the standard ratio estimator used for the GNS is inherently biased [33].

In homogeneous clusters, it is optimal to separately aggregate $\mathcal{G}_i$ and $\mathcal{S}_i$ via averaging. However, in Lemma B.1, we prove that the variance of both $\mathcal{G}_i$ and $\mathcal{S}_i$ depend on the local mini batch size. Furthermore, the local estimates of $\text{tr}(\Sigma)$ and $|G|^2$ for different nodes are correlated via dependence on $|g|^2$. As a result, aggregating $\mathcal{G}_i$ and $\mathcal{S}_i$ is more challenging for heterogeneous clusters. The following theorem states the optimal weighted combination of the local estimators, with the proof located in Appendix B.

THEOREM 4.1. *$\mathcal{G} = \sum_{i\in N} w_i^{\mathcal{G}} \mathcal{G}_i$ and $\mathcal{S} = \sum_{i\in N} w_i^{\mathcal{S}} \mathcal{S}_i$ are minimum variance, unbiased linear estimators of $|G|^2$ and $tr(\Sigma)$ when:*

$$\mathbf{w}^{\mathcal{G}} = \frac{\mathbf{1}^T A_{\mathcal{G}}^{-1}}{\mathbf{1}^T A_{\mathcal{G}}^{-1}\mathbf{1}}, \qquad \mathbf{w}^{\mathcal{S}} = \frac{\mathbf{1}^T A_{\mathcal{S}}^{-1}}{\mathbf{1}^T A_{\mathcal{S}}^{-1}\mathbf{1}}, \tag{11}$$

where $\mathbf{1}$ is an n-dimensional column vector of ones and both $A_{\mathcal{G}}$ and $A_{\mathcal{S}}$ are $n \times n$ matrices with respective entries $a_{\mathcal{G}}(i,j)$ and $a_{\mathcal{S}}(i,j)$:

$$a_{\mathcal{G}}(i,i) = \frac{B + 2b_i}{B^2 - Bb_i}, \quad a_{\mathcal{G}}(i,j) = \frac{B^2 - b_i^2 - b_j^2}{B(B-b_i)(B-b_j)} \text{ for } i \neq j$$

$$a_{\mathcal{S}}(i,i) = \frac{Bb_i}{B - b_i}, \quad a_{\mathcal{S}}(i,j) = \frac{b_i b_j (B - b_i - b_j)}{(B-b_i)(B-b_j)} \text{ for } i \neq j$$

Cannikin delivers a novel method to estimate the GNS $\mathcal{B}_{noise}$ in heterogeneous clusters. First, each node estimates the sum of the variances of the individual gradient components and the global norm of the gradient using (10). We optimally aggregate the local estimates $\mathcal{G}_i$ and $\mathcal{S}_i$ using (11), and then take the ratio of the resulting terms to get the global GNS $\mathcal{B}_{noise} = \mathcal{S}/\mathcal{G}$. Despite the added challenge of heterogeneity, Figure 6 shows Cannikin's convergence is comparable to the homogeneous baseline AdaptDL with the same training epochs, which means the larger batch size chosen by Cannikin won't harm the convergence efficiency.

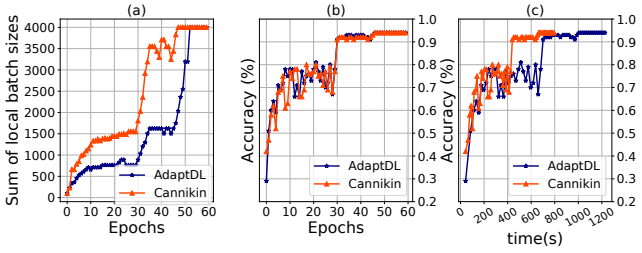

Figure 6: (a) the batch size of each epoch for CIFAR10 training. (b) the accuracy of each epoch. (c) the accuracy of time.

4.5 Implementation

Cannikin is implemented as a PyTorch library based on AdaptDL [45] that can be imported into DNN training scripts. Cannikin introduces the `HeteroDataLoader` class, which unevenly loads local mini batches to each node based on the *OptPerf* prediction. Cannikin is open-sourced at GitHub https://github.com/chengyinie/hetero_adaptdl. The implementation of Cannikin addresses the following concerns to improve efficiency.

Parameter learning. During each epoch, Cannikin collects the backpropagation time (P_i) and the total data loading time, optimization steps, and forward propagation(a_i) for each local batch size. With the collected data from two epochs using different local mini-batch sizes, each node can construct the computing time model to the local mini batch size by solving linear equations. In the subsequent epochs, more local batch sizes allow for the refinement of the computation time model, making it increasingly accurate.

When it comes to learning the communication time (T_{comm}) and γ, it's important to note that T_{comm} and γ remain constant across different local and batch sizes. Cannikin collects the overlap ratio (γ) and communication times (T_o and T_u) for each node in the cluster. We proceed to optimize the learning of γ and T_{comm} as follows.

Total batch size selection. Although the search algorithm efficiently finds the overlap pattern, the overhead can be significant if we determine the overlap pattern for each total batch size candidate determined by the adaptive batch size engine [45] before every epoch. Cannikin instead calculates $OptPerf_{init}$ for all batch size candidates after the initial epoch. In the upcoming epochs, since *OptPerf* is unrelated to the training progress, Cannikin uses $OptPerf_{init}$ and the updated GNS to choose the total batch size. Then Cannikin determines *OptPerf* along with $\mathbf{r}_{\text{opt}}$ according to the updated performance metrics. If the overlap pattern has changed from the initial pattern, Cannikin will start over to determine the pattern for each candidate again to choose the total batch size. Otherwise Cannikin will update $OptPerf_{init}$ for the corresponding total batch size candidate. With this strategy, in most epochs Cannikin only needs to determine *OptPerf* for one total batch size.

Overlap state searching. In the initialization epoch, Cannikin goes through all the total batch size candidates and calculates *OptPerf* for each candidate. When the total batch size increases, more cluster nodes will be computing-bottleneck nodes. Hence in the total batch size enumeration from small to large in sequence, the search starting point of an enumerated candidate is the overlap pattern of the previous one. In following epochs, the search starting point of an enumerated candidate is its overlap state in $OptPerf_{init}$.

5 EVALUATION

We evaluate the effectiveness of Cannikin in optimal distributed DNN training using convergence time, batch processing time, prediction accuracy, and system overhead. Key results are as follows:

- Cannikin reduced the overall convergence time by up to 85% and 52% in heterogeneous clusters compared with PyTorch and the adaptive batch size training system AdaptDL.
- Compared to the state-of-art data parallel distributed training strategies for heterogeneous clusters, Cannikin reduces the batch processing time by up to 18%.
- Cannikin predicted *OptPerf* for heterogeneous clusters within 7% error with low overhead less than 4%.

5.1 Experimental Setup

Testbed. We conduct our experiments in two different clusters: cluster *A* and cluster *B*. Cluster *A* is a heterogeneous 3-node cluster with different types of NVIDIA GPUs specified in Table 3. Cluster *B* is a heterogeneous 10-server cluster consisting of 16 GPUs, as detailed in the Table 4. Note that in Cluster *B*, each GPU is a node for data-parallelism distributed DL training.

Table 3: Hardware specification of cluster *A* in evaluation

Node type	Node count	GPU model	GPU Count	Main Memory	CPU
a5000	1	RTX A5000	1	32GB	i9-10980XE
a4000	1	RTX A4000	1	32GB	Xeon W-2255
p4000	1	Quadro P4000	1	32GB	Xeon W-2102

Table 4: Hardware specification of cluster *B* in evaluation

Node type	Node count	GPU model	GPU Count	Main Memory	CPU
a100	1	A100	4	512GB	Xeon Plati. 8380*2
v100	1	V100	4	128GB	Xeon Gold 6230*2
rtx	8	RTX6000	1	192GB	Xeon Gold 6126*2

Workloads. Evaluated workloads are listed in Table 5. The range of batch sizes is determined by each GPU's memory; the initial batch size is relatively small [45] and configured by users. We adopt the canonical setting for each training task for the optimizer, learning rate scaler, and target metrics choices, with the philosophy of testing different models and optimizers on applications of various sizes.

Baselines. We evaluate Cannikin by comparing it with the state-of-art adaptive batch size training system, data-parallelism heterogeneous distributed DL training system, and PyTorch DDP:

- AdaptDL [45]: The state-of-the-art adapted distributed DNN training system for homogeneous clusters.
- LB-BSP [8]: LB-BSP is a data-parallelism distributed training system for heterogeneous GPU clusters, which recurrently tune each node's local mini batch size for efficient model training. We set step size $\Delta = 5$ in our experiments, which is identical to the original paper.
- HetPipe[40]: HetPipe is a distributed training system for heterogeneous GPU clusters, which integrates the pipelined model parallelism and data parallelism.
- PyTorch DistributedDataParallel [30]: Pytorch DDP is one of the most efficient distributed training libraries for homogeneous clusters.

5.2 Performance with Heterogeneous GPUs

5.2.1 Overall convergence performance. We compared Cannikin with the baselines in cluster *B* for the overall convergence performance evaluation. Figure 7 shows the convergence processes of example tasks Cifar10 and Imagenet training in cluster *B*. Due to the weighted gradient aggregation, each gradient descent step of Cannikin is equivalent to the homogeneous gradient descent given the same total batch size. This equivalence guarantees that the convergence is not compromised. With this precondition, Cannikin achieves a convergence speed up from throughput improvement and the improved prediction of optimal total batch size in heterogeneous clusters, thus increasing the adaptive training system's goodput. Our results show Cannikin reduces the convergence time by 52% and 29% for CIFAR-10 and ImageNet compared with AdaptDL.

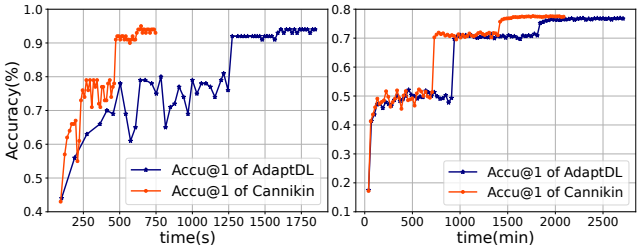

Figure 7: Convergence process of ResNet-18 on CIFAR-10 (left) and ResNet-50 on ImageNet (right).

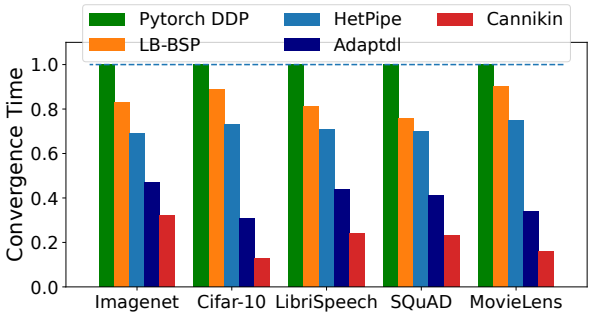

Figure 8: Normalized convergence time of all training tasks.

The normalized overall convergence time for each evaluated workload is depicted in Figure 8. For PyTorch DDP, which trains deep learning models with fixed total batch size and distributes local batch sizes evenly across heterogeneous clusters, the speedup achieved by Cannikin is primarily attributed to its optimized prediction of total/local batch sizes during training. Unlike PyTorch DDP, which uses fixed batch sizes, AdaptDL evenly distributes local batch sizes across the cluster and predicts the optimal total batch size in homogeneous environments. In the context of AdaptDL, the speedup observed with Cannikin results from the optimized selection of local batch sizes to maximize the utilization of heterogeneous GPUs and the improved prediction of total batch sizes in heterogeneous environments. LB-BSP iteratively tunes local batch sizes for each GPU within heterogeneous clusters which improves the utilization of the heterogeneous GPUs. However, LB-BSP only supports the DL training with a fixed total batch size, and LB-BSP doesn't consider the communication and communication overlap. The speedup with Cannikin compared to LB-BSP primarily arises from the faster determination of the optimal local batch sizes considering communication and computing overlap and optimized total batch size selection during training. The results indicate Cannikin significantly enhances the overall convergence time to achieve the

Table 5: The models, datasets information of Cannikin's evaluation.

Task	Dataset	Model	Size	Optimizer	LR scaler	B_0	Target
Image Classification	ImageNet [12]	ResNet-50 [19]	25.6M	SGD	Adascale	100	75% Top1 acc.
Image Classification	CIFAR-10 [27]	ResNet-18 [19]	11M	SGD	Adascale	64	94% Top1 acc.
Speech Recognition	LibriSpeech [39]	DeepSpeech2 [4]	52M	SGD	Adascale	12	WER = 40.0%
Question Answering	SQuAD [46]	BERT [14]	110M	AdamW	Square-Root	9	F1 = 88%
Recommendation	MovieLens [18]	NeuMF [20]	5.2M	Adam	Square-Root	64	Hit rate = 69%

target accuracy, with improvements of up to 85%, 52%, and 82% compared to PyTorch DDP, AdaptDL, and LB-BSP respectively.

5.2.2 Batch processing time for heterogeneous clusters. We evaluate Cannikin using two methodologies for batch processing time. The first is the fixed total batch size training, i.e., classical DNN training. The second is the adaptive batch size situation when the total batch size varies in each training epoch. Since AdaptDL's batch processing time in heterogeneous clusters is equivalent to Pytorch DDP, we don't consider AdaptDL in this section.

With fixed batch size. We fix the total batch size of the cluster and each node's optimal local mini batch size ratio $\mathbf{r}_{\text{opt}}$. Figure 9 shows an example of Cannikin and LB-BSP training ResNet-50 with ImageNet in cluster *A*. Given the total batch size of 128, Cannikin and LB-BSP initialize training by evenly assigning local batch size for each node. Cannikin approach *OptPerf* as early as the third epoch, because Cannikin requires two epochs to learn the performance models discussed in Section 4.2. However, LB-BSP requires more than ten epochs to reach its best performance.

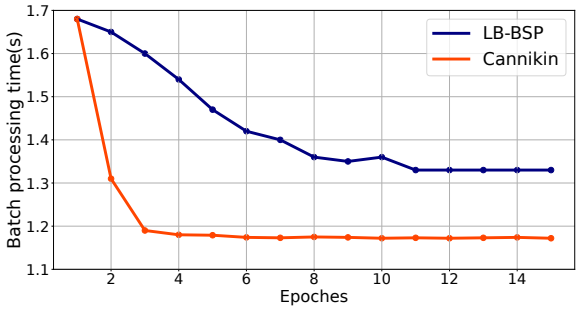

Figure 9: Cluster A's batch size processing time when training ImageNet from evenly assigned local mini batch size initialization given fixed total batch size 128.

Assume Cannikin and each compared method have reached their best batch processing time for the given batch size. Figure 10 shows *OptPerf* of cluster *B* compared with the baselines. We can observe that *OptPerf* is at most 18% faster compared with LB-BSP, and up to 53% faster than the Pytorch DDP. Note that when the batch size is large enough, all nodes become computing bottleneck. The *OptPerf* will be achieved when all nodes have the same computing time $T_{compute}$. The performance of LB-BSP will approach *OptPerf* because, asymptotically, the two have the same target that all nodes have the same $T_{compute}$.

With adaptive batch size. Since only data-parallelism distributed systems are sensitive to batch size changes, we evaluate Cannikin with LB-BSP for the adaptive batch size situation. Assuming Cannikin and LB-BSP have already achieved their best performance for the previous batch size, now given a new batch size that is 10% of the total batch size range larger than the previous one, the batch processing time of LS-BSP will become sub-optimal because the $\mathbf{r}_{\text{opt}}$ has changed. In the meantime, Cannikin can still accurately predict the *OptPerf* for the newly assigned batch size, just like the fixed batch size situation. Figure 10 shows the batch processing time of LS-BSP in cluster *B* for adaptive batch size training.

5.3 *OptPerf* Prediction

In cluster *A*, we evaluate the prediction of *OptPerf* with and without inverse variance weighting in measurement compared to the manually tuned *OptPerf*. Results show that without inverse variance weighting, the maximum error of *OptPerf* prediction can reach up to 21%. With the inverse variance weighting method introduced in Section 4.5, Cannikin's prediction of *OptPerf* in small and medium models like NeuMF, ResNet-18, and ResNet-50 have a maximum 3% error. For larger models like BERT and DeepSpeech2, larger model sizes lead to more gradient buckets to synchronize, which increases the probability of contingency in gradient synchronization, so the maximum error in the prediction of *OptPerf* is 7% in the batch size range. However, Cannikin trains with varying batch sizes, so the maximum 7% error of the *OptPerf* prediction is only used in a fraction of the entire training process.

5.4 Overhead and Scalability of Cannikin

Table 6 shows the overhead of Cannikin for each task we deployed in the large-scale test cluster *B*. The overhead encompasses the time required to evaluate each candidate's total batch size alongside its corresponding *OptPerf*, as well as the configuration time for each node's local batch size and local training data index. For all the medium and large applications, the configuring time for *OptPerf* of Cannikin before each epoch is much less than 1% of the total epoch training time across all candidate batch sizes in the range specified by [45], which is insignificant for the entire training process. For small applications like CIFAR-10 and MovieLens, the overhead of Cannikin will reach up to 9% and 12% when the system runs with batch sizes around the upper limit of the batch size range. However, during training, the system will use the batch sizes near the upper limit only when the model almost converges. The period of time used for batch sizes near the upper limit for training is a minority part of training time. Considering the entire training progress, the overheads of CIFAR-10 and MovieLens are 2.7% and 3.9%.

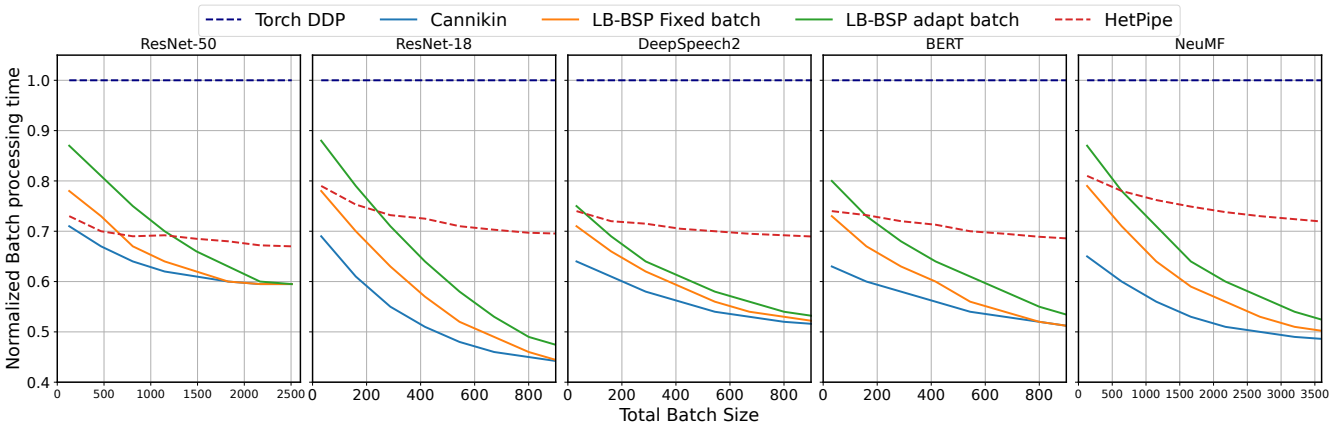

Figure 10: Each evaluation task's normalized batch processing time to the total batch size.

Table 6: The overhead analysis of Cannikin.

Dataset	Model	Max Overhead	Overall Overhead
ImageNet	ResNet-50	$\ll 1\%$	$\ll 1\%$
LibriSpeech	DeepSpeech2	$\ll 1\%$	$\ll 1\%$
SQuAD	BERT	$\ll 1\%$	$\ll 1\%$
CIFAR-10	ResNet-18	9%	2.7%
MovieLens	NeuMF	12%	3.9%

6 DISCUSSION

Impact of varying heterogeneity degree. The performance improvement compared with the baseline depends on the heterogeneity of the cluster. Generally speaking, a cluster with more heterogeneity will benefit more from Cannikin. In homogeneous clusters, Cannikin's performance is identical to AdaptDL. In Cluster *B*, the fastest GPU, A100, is about 3.42 times faster compared with RTX6000, which is the slowest GPU. The degree of heterogeneity we evaluated in this paper generally exists in today's computing platform. As shown in Table 1, after two years, the H100 GPU is faster than A100 by more than 4 times.

To provide a theoretical understanding of the potential performance improvement in a heterogeneous cluster, we can frame the problem as a load-balancing optimization. Consider a scenario where worker A is N times faster than worker B. If both workers are assigned the same amount of work, there is a clear inefficiency. However, the overall throughput improves if we allocate N times more work to worker A. The theoretical upper bound on this improvement is given by the ratio $\frac{2}{N+1}$. This result demonstrates that optimal task distribution in heterogeneous clusters can significantly enhance performance, especially as the gap between hardware capabilities widens.

Potentials with Sharing-caused heterogeneity. The heterogeneity can arise not only from hardware differences but also from resource sharing. Recent studies [52, 54] introduced GPU-sharing mechanisms that enable the sharing of a single GPU's resources among multiple instances. In this context, even when the same GPU type is present within a cluster, the resources available at each node can still exhibit heterogeneity during distributed training.

We create Cluster *C*, a 16-node homogeneous cluster in Chameleon Cloud [25]. Each node is equipped with one NVIDIA RTX6000 GPU. We use the container's constraint to construct the heterogeneous environment. We adopt docker containers [34] for cluster *C* to configure the heterogeneous environment. In each node, we start two docker containers. The first container runs Cannikin distributed training workloads, and the second docker container runs a local dummy GPU workload to share the same GPU's computing power and memory with Cannikin. To tune each node's computing power, we manually adjust the local dummy GPU workload's batch size to change the computing power and memory of Cannikin workloads.

The results indicate that Cannikin's performance in Cluster *C* aligns with that of Cluster *A* and *B*. This brief experiment demonstrates the potential of Cannikin in addressing heterogeneity induced by resource sharing.

Adapt to schedulers for heterogeneous clusters. Existing dynamic resource allocating schedulers [43, 45, 58] only support the scheduling of homogeneous clusters. Sia [22] is a scheduler with heterogeneity awareness. However, the resources allocated for each job are still homogeneous for each job. Cannikin supports job schedulers that allocate a heterogeneous cluster for each job, which can significantly increase resource utilization and flexibility.

7 OTHER RELATED WORK

Performance modeling of DNN training. The importance of performance modeling in deep neural network training has been shown in recent research work. Paleo [44] proposed the DNN training model by studying the operator topology. Clockwork [16] model the GPU runtime with tracing. The All-reduce communication between nodes is studied and modeled [42]. For the cloud-based DNN training [31], the accurate performance modeling and prediction significantly increase the training efficiency. From the scheduler perspective, an accurate DNN performance modeling [28, 35, 43, 45] increases the resource utilization and improves the fairness of multiple jobs execution.

Accelerating ML on heterogeneous environments. Most previous work about ML acceleration for heterogeneous clusters is on the scheduler level for multiple jobs, like Hare [9], EasyScale [29] dynamically assigned workers to scale distributed training for heterogeneous GPUs, Gandiva [7], GPUlet [11]. However, looking into the single job level for the schedulers, the training strategy is still homogeneous. For job-level optimization on heterogeneous clusters, SnuHPL [26] improved the training in a heterogeneous HPC system by optimizing the data distribution for a given cluster configuration. BytePS [23] accelerated DNN training by leveraging CPU resources. However, BytePS focuses on the heterogeneity between CPU/GPU and improves the communication mechanism of all-reduce and the parameter server. Cannikin is a job-level-optimized system designed for heterogeneous GPU clusters. It automatically explores and determines the optimal local batch sizes assigned to each GPU and total batch sizes. HeteroG [56] proposed operation-level hybrid parallelism to deploy the DNN training model to heterogeneous GPUs, Hetpipe [40] integrated pipelined model parallelism with data parallelism in heterogeneous clusters; however, they only considered fixed batch size training, whereas adaptive batch size training significantly improves DNN training performance.

8 CONCLUSION

In this paper, we introduced OptPerf, the optimal batch processing time for data-parallel DNN training in heterogeneous clusters, to reduce the impact of stragglers. We design Cannikin, a scalable, near-optimal distributed training system that leverages OptPerf to handle GPU heterogeneity. Cannikin is the first adaptive distributed training system for heterogeneous clusters with near-optimal performance and high scalability by overcoming challenges such as optimal scenario determination and metrics measurement caused by heterogeneity. Cannikin outperforms the state-of-the-art systems for diverse workloads in real heterogeneous clusters.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and the guidance of our shepherd, Dr. Raúl Gracia. This work is supported by the US National Science Foundation under grant numbers CNS-2106027, CNS-2214980, CNS-2146909, and CCF-2046444.

A PROOF OF OPTIMALITY CONDITIONS

A.1 Compute-Bottleneck Scenario

Proof. When computing is the bottleneck for all nodes, the total processing time of one batch for the cluster is $\max_{i\in\mathcal{N}}\{t^i_{compute} + T_u\}$. Since T_u is the same across all nodes, we can just consider $\max_{i\in\mathcal{N}}\{a_i + k_i b_i + m_i\}$. We need:

$$\begin{aligned} \min \quad & \mu \\ \text{s.t.} \quad & a_i + k_i b_i + m_i - \mu \le 0, \quad \forall i \in \mathcal{N} \\ & B - \sum_{i\in\mathcal{N}} b_i = 0. \end{aligned}$$

This optimization problem has corresponding Lagrangian:

$$L(\mu, \mathbf{b}, \lambda, \nu) = \mu + \sum_{i\in\mathcal{N}} \lambda_i (a_i + k_i b_i + m_i - \mu) + \nu \left(B - \sum_{i\in\mathcal{N}} b_i \right)$$

Using the complimentary slackness conditions, we can solve and get $\lambda_i = (1/k_i)(\sum_{i\in\mathcal{N}} 1/k_i)^{-1}$. The Karush-Kuhn-Tucker (KKT) conditions state that the optimal solution μ^* to this problem must satisfy $\lambda_i(a_i + k_i b_i + m_i - \mu^*) = 0$. Since λ_i is strictly positive, $a_i + k_i b_i + m_i = \mu^*, \forall i \in \mathcal{N}$ so $t^i_{compute} = t^j_{compute}, \forall i, j \in \mathcal{N}$. □

A.2 Communication-Bottleneck Scenario

Proof. When communication is the bottleneck for all nodes, the total processing time of one batch is $\max_{i\in\mathcal{N}}\{syncStart_i + T_{comm}\}$. Since T_{comm} is the same across all nodes, we can just consider $\max_{i\in\mathcal{N}}\{a_i + \gamma(k_i b_i + m_i)\}$. Using the same technique as in the previous proof, we now get that for the optimal solution μ^*, $a_i + \gamma(k_i b_i + m_i) = \mu^*, \forall i \in \mathcal{N}$. □

A.3 General Optimal Scenario

Proof. Let $\mathcal{N}_1$ be the set of computation-bottleneck nodes in $\mathcal{N}$ and $\mathcal{N}_2$ be the set of communication-bottleneck nodes. We need:

$$\begin{aligned} \min \quad & \mu \\ \text{s.t.} \quad & a_i + k_i b_i + m_i - \mu \le 0, \quad \forall i \in \mathcal{N}_1 \\ & a_i + \gamma(k_i b_i + m_i) + T_o - \mu \le 0, \quad \forall i \in \mathcal{N}_2 \\ & B - \sum_{i\in\mathcal{N}} b_i = 0. \end{aligned}$$

If we construct the Lagrangian for this problem, we see that by solving the complimentary slackness equations, the coefficients λ_i are strictly positive for all i. Thus the optimal solution μ^* satisfies $a_i + k_i b_i + m_i = \mu^*$ for $i \in \mathcal{N}_1$ and satisfies $a_i + \gamma(k_i b_i + m_i) + T_o = \mu^*$ for $i \in \mathcal{N}_2$, giving the desired result. □

B THE GNS IN HETEROGENEOUS CLUSTERS

Proof of Theorem 4.1.

Proof. Since $\mathcal{G}_i$ is an unbiased estimator of $|G|^2$, $\mathcal{G}$ is an unbiased estimator when $\sum_{i\in\mathcal{N}} w_i^{\mathcal{G}} = 1$. Furthermore $\mathcal{G}$ is the minimum variance, unbiased linear estimator when $\mathbf{w}$ minimizes the quadratic form of $\Sigma(\mathcal{G}_i)$, where $\Sigma(\mathcal{G}_i)$ is the correlation matrix of the estimators $\mathcal{G}_i$. Using Lagrange multipliers, we get:

$$\mathbf{w}^{\mathcal{G}} = \frac{\mathbf{1}^T \Sigma(\mathcal{G}_i)^{-1}}{\mathbf{1}^T \Sigma(\mathcal{G}_i)^{-1} \mathbf{1}}$$

Similarly, we get that $\mathcal{S}$ is the unbiased linear estimator of $\mathrm{tr}(\Sigma)$:

$$\mathbf{w}^{\mathcal{S}} = \frac{\mathbf{1}^T \Sigma(\mathcal{S}_i)^{-1}}{\mathbf{1}^T \Sigma(\mathcal{S}_i)^{-1} \mathbf{1}}$$

where $\Sigma(\mathcal{S}_i)$ is the covariance matrix of the estimators $\mathcal{S}_i$.

To compute $\mathbf{w}^{\mathcal{G}}$, we require $\Sigma(\mathcal{G}_i)$. By definition, the matrix's diagonal elements are $\mathrm{Var}(\mathcal{G}_i)$ and the off-diagonal elements are $\mathrm{Cov}(\mathcal{G}_i, \mathcal{G}_j)$ for $i \neq j$. Lemma B.1 gives us $\mathrm{Var}(\mathcal{G}_i)$ and Lemma B.2 gives us $\mathrm{Cov}(\mathcal{G}_i, \mathcal{G}_j)$. Since all terms of $\Sigma(\mathcal{G}_i)$ have a common factor of $4|G|^2\mathrm{tr}(\Sigma)$, this factor will cancel for $\mathbf{w}^{\mathcal{G}}$, so we can equivalently solve for $\mathbf{w}^{\mathcal{G}}$ using the matrix $A_{\mathcal{G}}$ instead of $\Sigma(\mathcal{G}_i)$, where:

$$a_{\mathcal{G}}(i,i) = \frac{B + 2b_i}{B^2 - Bb_i}, \quad a_{\mathcal{G}}(i,j) = \frac{B^2 - b_i^2 - b_j^2}{B(B - b_i)(B - b_j)} \text{ for } i \neq j$$

We can use a similar argument for $\mathcal{S}$ with Lemmas B.4 and B.3, where rather than using $\Sigma(\mathcal{S}_i)$ we can use the matrix $A_{\mathcal{G}}$ with

entries:

$$a_{\mathcal{S}}(i,i) = \frac{Bb_i}{B-b_i}, \quad a_{\mathcal{S}}(i,j) = \frac{b_i b_j (B-b_i-b_j)}{(B-b_i)(B-b_j)} \text{ for } i \neq j$$

□

Lemma B.1. *The estimators $\mathcal{G}_i$ and $\mathcal{S}_i$ have variances:*

$$Var(\mathcal{G}_i) = 4|G|^2 tr(\Sigma)\left(\frac{B+2b_i}{B^2-Bb_i}\right)$$

$$Var(\mathcal{S}_i) = 4|G|^2 tr(\Sigma)\left(\frac{Bb_i}{B-b_i}\right)$$

where $tr(\Sigma)$ is the sum of the variances of the individual gradient components and $|G|^2$ is the global norm of the gradient.

Proof. First we compute the variance of $\mathcal{G}_i$:

$$\begin{aligned}
\mathrm{Var}(\mathcal{G}_i) &= \mathrm{Var}\left(\frac{B}{B-b_i}|g|^2 - \frac{b_i}{B-b_i}|g_i|^2\right) = \\
&\overset{(1)}{=} \left(\frac{B}{B-b_i}\right)^2 \mathrm{Var}(|g|^2) + \left(\frac{b_i}{B-b_i}\right)^2 \mathrm{Var}(|g_i|^2) - \\
&\quad -2\left(\frac{B}{B-b_i}\right)\left(\frac{b_i}{B-b_i}\right)\mathrm{Cov}(|g|^2,|g_i|^2) = \\
&\overset{(2)}{=} \frac{B+b_i}{(B-b_i)^2}\left(4|G|^2\mathrm{tr}(\Sigma)\right) - \frac{2Bb_i}{(B-b_i)^2}\cdot\frac{4b_i|G|^2\mathrm{tr}(\Sigma)}{B^2} = \\
&= 4|G|^2\mathrm{tr}(\Sigma)\left(\frac{B+2b_i}{B^2-Bb_i}\right)
\end{aligned}$$

where (1) follows from the variance of sums of random variables, (2) follows from Lemma B.4 and Lemma B.5. We can similarly compute the variance of $\mathcal{S}_i$:

$$\begin{aligned}
\mathrm{Var}(\mathcal{S}_i) &= \mathrm{Var}\left(\frac{b_i B}{B-b_i}|g_i|^2 - \frac{b_i B}{B-b_i}|g|^2\right) = \\
&\overset{(1)}{=} \left(\frac{b_i B}{B-b_i}\right)^2 \left(\mathrm{Var}(|g_i|^2) + \mathrm{Var}(|g|^2) - 2\mathrm{Cov}(|g|^2,|g_i|^2)\right) = \\
&\overset{(2)}{=} 4|G|^2\mathrm{tr}(\Sigma)\left(\frac{b_i B}{B-b_i}\right)^2\left(\frac{1}{b_i}+\frac{1}{B}-\frac{2b_i}{B}\right) = \\
&= 4|G|^2\mathrm{tr}(\Sigma)\left(\frac{Bb_i}{B-b_i}\right)
\end{aligned}$$

where again (1) follows from the variance of sums of random variables, (2) follows from Lemma B.4 and Lemma B.5. □

Lemma B.2. *The estimators $\mathcal{G}_i$ and $\mathcal{G}_j$ have covariance:*

$$Cov(\mathcal{G}_i,\mathcal{G}_j) = 4|G|^2 tr(\Sigma)\frac{B^2-b_i^2-b_j^2}{B(B-b_i)(B-b_i)}$$

Proof. Using the definition of $\mathcal{G}_i$ and $\mathcal{G}_j$:

$$\begin{aligned}
\mathrm{Cov}(\mathcal{G}_i,\mathcal{G}_j) &= \mathrm{Cov}\left(\frac{B|g|^2-b_i|g_i|^2}{B-b_i}, \frac{B|g|^2-b_j|g_j|^2}{B-b_j}\right) = \\
&\overset{(1)}{=} \frac{B^2\mathrm{Var}(|g|^2)}{(B-b_i)(B-b_j)} - \frac{Bb_i\mathrm{Cov}(|g|^2,|g_i|^2)}{(B-b_i)(B-b_j)} - \frac{Bb_j\mathrm{Cov}(|g|^2,|g_j|^2)}{(B-b_i)(B-b_j)} = \\
&\overset{(2)}{=} \frac{4|G|^2\mathrm{tr}(\Sigma)}{(B-b_i)(B-b_j)}\left(\frac{1}{B}-\frac{b_i^2}{B}-\frac{b_j^2}{B}\right)
\end{aligned}$$

where (1) follows from the covariance of linear combinations of random variables and the independence of g_i and g_j, and (2) follows from Lemma B.4 and Lemma B.5. □

Lemma B.3. *The estimators $\mathcal{S}_i$ and $\mathcal{S}_j$ have covariance:*

$$Cov(\mathcal{S}_i,\mathcal{S}_j) = 4|G|^2 tr(\Sigma)\frac{b_i b_j(B-b_i-b_j)}{(B-b_i)(B-b_j)}$$

Proof. Using the definition of $\mathcal{S}_i$ and $\mathcal{S}_j$:

$$\begin{aligned}
\mathrm{Cov}(\mathcal{S}_i,\mathcal{S}_j) &= \mathrm{Cov}\left(\frac{Bb_i}{B-b_i}(|g_i|^2-|g|^2), \frac{Bb_j}{B-b_j}(|g_j|^2-|g|^2)\right) = \\
&\overset{(1)}{=} \frac{B^2 b_i b_j}{(B-b_i)(B-b_j)}\left(\mathrm{Var}(|g|^2) - \mathrm{Cov}(|g|^2,|g_i|^2) - \mathrm{Cov}(|g|^2,|g_i|^2)\right) = \\
&\overset{(2)}{=} \frac{4|G|^2\mathrm{tr}(\Sigma)B^2 b_i b_j}{(B-b_i)(B-b_j)}\left(\frac{1}{B}-\frac{b_i}{B^2}-\frac{b_j}{B^2}\right)
\end{aligned}$$

where (1) follows from the covariance of linear combinations of random variables and the independence of g_i and g_j, and (2) follows from Lemma B.4 and Lemma B.5. □

Lemma B.4. *For any estimated gradient g_{est} with corresponding batch size b, the variance of the gradient norm satisfies:*

$$Var(|g_{est}|^2) \approx \frac{4|G|^2 tr(\Sigma)}{b}$$

Proof. Error propagation using Taylor's rule [38] (also known as the delta method) gives us the approximation:

$$\mathrm{Var}(|g_{est}|^2) \approx 4\mathbb{E}[|g_{est}|]^2\mathrm{Var}(|g_{est}|) = 4|G|^2\cdot\frac{1}{b}\mathrm{tr}(\Sigma)$$

□

Lemma B.5. *For local gradient g_i at node i with batch size b_i and global gradient g with batch size B and computed using Eq (9), the covariance of the two gradient norms is:*

$$Cov(|g|^2,|g_i|^2) = \frac{4b_i|G|^2 tr(\Sigma)}{B^2}$$

Proof. The global gradient norm $|g|^2$ can be written in terms of g_i and non-g_i components:

$$|g|^2 = \frac{b_i^2}{B^2}|g_i|^2 + \frac{1}{B^2}\sum_{j\notin b_i}(\nabla_\theta L_{x_j}(\theta))^2$$

Rewriting the covariance using this expression:

$$\begin{aligned}
\mathrm{cov}(|g|^2,|g_i|^2) &= \mathrm{cov}\left(\frac{b_i^2}{B^2}|g_i|^2 + \frac{1}{B^2}\sum_{j\notin b_i}(\nabla_\theta L_{x_j}(\theta))^2, |g_i|^2\right) = \\
&\overset{(1)}{=} \frac{b_i^2}{B^2}\mathrm{cov}(|g_i|^2,|g_i|^2) + \frac{1}{B}\mathrm{cov}\left(\sum_{j\notin b_i}(\nabla_\theta L_{x_j}(\theta))^2, |g_i|^2\right) = \\
&\overset{(2)}{=} \frac{b_i^2}{B^2}\mathrm{Var}(|g_i|^2) \overset{(3)}{=} \frac{b_i^2}{B^2}\cdot\frac{4|G|^2\mathrm{tr}(\Sigma)}{b_i} = \frac{4b_i|G|^2\mathrm{tr}(\Sigma)}{B^2}
\end{aligned}$$

where (1) follows from the covariance of linear combinations of random variables, (2) follows from the variance-covariance relationship and the independence of g_i and g_j for $i \neq j$ and (3) follows from Lemma B.4. □

REFERENCES

[1] 1993. MPI: A message passing interface. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 878–883. https://doi.org/10.1145/169627.169855

[2] 2011. Chapter 4 - The CUDA Execution Model. In *CUDA Application Design and Development*, Rob Farber (Ed.). Morgan Kaufmann, Boston, 85–108. https://doi.org/10.1016/B978-0-12-388426-8.00004-5

[3] Martín Abadi, Paul Barham, and etc Chen, Jianmin. 2016. TensorFlow: A system for large-scale machine learning. https://doi.org/10.48550/ARXIV.1605.08695

[4] Dario Amodei, Rishita Anubhai, and etc Battenberg, Eric. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. https://doi.org/10.48550/ARXIV.1512.02595

[5] Paul Barham, Aakanksha Chowdhery, and etc Dean, Jeff. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 430–449. https://proceedings.mlsys.org/paper/2022/file/98dce83da57b0395e163467c9dae521b-Paper.pdf

[6] A. Bojańczyk. 1984. Complexity of Solving Linear Systems in Different Models of Computation. *SIAM J. Numer. Anal.* 21, 3 (1984), 591–603. http://www.jstor.org/stable/2157070

[7] 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* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/3342195.3387555

[8] Chen Chen, Qizhen Weng, and etc Wang, Wei. 2020. Semi-Dynamic Load Balancing: Efficient Distributed Learning in Non-Dedicated Environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 431–446. https://doi.org/10.1145/3419111.3421299

[9] Fahao Chen, Peng Li, Celimuge Wu, and Song Guo. 2022. Hare: Exploiting Inter-Job and Intra-Job Parallelism of Distributed Machine Learning on Heterogeneous GPUs. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/3502181.3531462

[10] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 241–251. https://proceedings.mlsys.org/paper/2019/file/9b8619251a19057cff70779273e95aa6-Paper.pdf

[11] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. https://doi.org/10.48550/ARXIV.1712.02029

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

[15] Yifan Ding, Nicholas Botzer, and Tim Weninger. 2021. HetSeq: Distributed GPU Training on Heterogeneous Infrastructure. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 17 (May 2021), 15432–15438. https://doi.org/10.1609/aaai.v35i17.17813

[16] Arpan Gujarati, Reza Karimi, and etc Alzayat, Safya. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 25, 20 pages.

[17] Aaron Harlap, Deepak Narayanan, and etc Phanishayee, Amar. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. https://doi.org/10.48550/ARXIV.1806.03377

[18] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (dec 2015), 19 pages. https://doi.org/10.1145/2827872

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. https://doi.org/10.48550/ARXIV.1512.03385

[20] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. https://doi.org/10.48550/ARXIV.1708.05031

[21] Facebook incubator. [n. d.]. Facebookincubator/gloo: Collective Communications Library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo.

[22] Suhas Jayaram Subramanya, Daiyaan Arfeen, and etc Lin. 2023. Sia: Heterogeneity-Aware, Goodput-Optimized ML-Cluster Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 642–657. https://doi.org/10.1145/3600006.3613175

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

[24] Norman P. Jouppi, Cliff Young, and etc Patil, Nishant. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[25] Kate Keahey, Jason Anderson, and etc Zhuo Zhen. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[26] Jinpyo Kim, Hyungdal Kwon, and etc Kang, Jintaek. 2022. SnuHPL: High Performance LINPACK for Heterogeneous GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/3524059.3532370

[27] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images.

[28] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/3342195.3387547

[29] Mingzhen Li, Wencong Xiao, and etc Sun, Biao. 2022. EasyScale: Accuracy-consistent Elastic Training for Deep Learning. https://doi.org/10.48550/ARXIV.2208.14228

[30] Shen Li, Yanli Zhao, and etc Varma, Rohan. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[31] Liang Luo, Peter West, and etc Patel, Pratyush. 2022. SRIFTY: Swift and Thrifty Distributed Neural Network Training on the Cloud. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 833–847. https://proceedings.mlsys.org/paper/2022/file/f457c545a9ded88f18ecee47145a72c0-Paper.pdf

[32] Luo Mai, Guo Li, and etc Wagenländer. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 937–954.

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

[34] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[35] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[36] Eriko Nurvitadhi, Jeffrey Cook, and etc Mishra, Asit. 2018. In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 106–1064. https://doi.org/10.1109/FPL.2018.00027

[37] Nvidia. [n. d.]. Nvidia/NCCL: Optimized Primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl.

[38] Gary W Oehlert. 1992. A note on the delta method. *The American Statistician* 46, 1 (1992), 27–29.

[39] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[40] Jay H. Park, Gyeongchan Yun, and etc Chang M. Yi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 307–321. https://www.usenix.org/conference/atc20/presentation/park

[41] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations. *J. Parallel Distrib. Comput.* 69, 2 (feb 2009), 117–124. https://doi.org/10.1016/j.jpdc.2008.09.002

[42] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations. *J. Parallel Distrib. Comput.* 69, 2 (feb 2009), 117–124. https://doi.org/10.1016/j.jpdc.2008.09.002

[43] Yanghua Peng, Yixin Bao, and etc Chen, Yangrui. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3190508.3190517

[44] Qi, Evan R. Sparks, and Ameet S. Talwalkar. 2016. Paleo: A Performance Model for Deep Neural Networks. In *International Conference on Learning Representations*.

[45] Aurick Qiao, Sang Keun Choe, and etc Subramanya, Suhas Jayaram. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[46] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2383–2392. https://doi.org/10.18653/v1/D16-1264

[47] R.R. Schaller. 1997. Moore's law: past, present and future. *IEEE Spectrum* 34, 6 (1997), 52–59. https://doi.org/10.1109/6.591665

[48] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[49] Jaime Sevilla, Lennart Heim, and etc Ho, Anson. 2022. Compute Trends Across Three Eras of Machine Learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN55064.2022.9891914

[50] Mohammad Shoeybi, Mostofa Patwary, and etc Puri, Raul. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. https://doi.org/10.48550/ARXIV.1909.08053

[51] Ashish Vaswani, Noam Shazeer, and etc Parmar, Niki. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[52] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[53] Yonghui Wu, Mike Schuster, and etc Chen, Zhifeng. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. https://doi.org/10.48550/ARXIV.1609.08144

[54] Wencong Xiao, Shiru Ren, and etc Yong Li. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[55] Qing Ye, Yuhao Zhou, Mingjia Shi, Yanan Sun, and Jiancheng Lv. 2022. DLB: A Dynamic Load Balance Strategy for Distributed Training of Deep Neural Networks. *IEEE Transactions on Emerging Topics in Computational Intelligence* (2022), 1–11. https://doi.org/10.1109/TETCI.2022.3220224

[56] Xiaodong Yi, Shiwei Zhang, and etc Luo, Ziyue. 2020. Optimizing distributed training deployment in heterogeneous GPU clusters. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 93–107. https://doi.org/10.1145/3386367.3432728

[57] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3225058.3225069

[58] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. 2017. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 390–404. https://doi.org/10.1145/3127479.3127490

[59] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Comput. Surv.* 52, 1, Article 5 (feb 2019), 38 pages. https://doi.org/10.1145/3285029

[60] Xiaofan Zhang, Junsong Wang, and etc Zhu, Chao. 2018. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240801