Espresso: Cost-Efficient Large Model Training by Exploiting GPU Heterogeneity in the Cloud

Qiannan Zhou†, Fei Xu†*, Lingxuan Weng†, Ruixing Li†, Xudong Wu†, Li Chen‡, Zhi Zhou§, Fangming Liu¶
†Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University.
‡University of Louisiana at Lafayette. §Sun Yat-sen University. ¶Peng Cheng Laboratory.

***Abstract*—As Transformer-based models deepen and datasets expand, training large models demands numerous accelerators, particularly GPUs, bringing high cloud expenses. However, conventional *homogeneous* resource provisioning is inefficient due to limited cloud resources and low GPU utilization. This challenge necessitates heterogeneous GPU provisioning for training in clouds. Current research on large model training often focuses on load balancing of stages, neglecting the varying computing and memory demands across stages. Additionally, the allocation of heterogeneous GPUs for training has surprisingly received little attention. This paper introduces *Espresso*, a cost-efficient GPU provisioning framework that unifies the *heterogeneous GPU allocation* (GPU allocator) and *adequate stage placement* (stage placer) for large model training in the cloud. Specifically, the GPU allocator proposes a *cost tree-based* provisioning strategy to prioritize searching allocation plans with lower costs and reduce unnecessary branches by multi-dimensional pruning methods. The resource-aware stage placer further devises a *compute-memory ratio* to optimize communication and computation efficiency during training. We have open-sourced a prototype of *Espresso* and conducted prototype experiments on four representative large models in public clouds. Extensive experiment results demonstrate that *Espresso* guarantees the performance for large model training while saving costs by up to 49.8% compared to state-of-the-art solutions, yet with acceptable runtime overhead.**

***Index Terms*—Large model training, resource provisioning, stage placement, heterogeneous GPU environments**

I. Introduction

With breakthroughs in techniques such as natural language processing [1] and computer vision [2], many enterprises and research institutions (*e.g.,* OpenAI, NVIDIA) are joining the research and application of large models, thereby driving up the demand for computational resources. Public cloud services, such as Amazon AWS, Google Cloud, and Microsoft Azure, serve as the mainstream platforms supporting large model training due to their scalability, cost efficiency, and ease of use [3]. With the continuous advancement and generational update of GPU technology, multiple GPU types are commonly running in public cloud environments. The diversity of GPUs, including both high-end types (*e.g.,* NVIDIA H100, A100) and low-end types (*e.g.,* NVIDIA T4 and P100), provides a broader range of choices for resource provisioning, significantly enhancing the flexibility of large model training strategies.

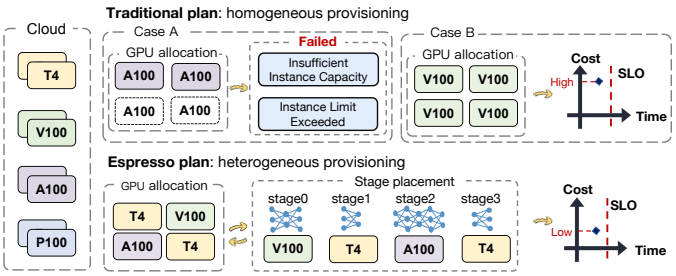


Fig. 1: Overview of *Espresso*. Compared to the traditional *homogeneous* resource provisioning plan, *Espresso* outputs a *cost-efficient heterogeneous* resource provisioning plan by the adequate GPU allocation and judicious stage placement for guaranteeing the SLO of large model training in the cloud.

Unfortunately, distributed training for large models commonly requires *homogeneous* GPUs and *gang-scheduling*, which indicates that the training job cannot be initiated unless all the required GPU resources are available simultaneously [4]. This is because distributed training involves partitioning the input data and/or the model layers, and thus needs to determine the GPU allocation and stage placement before model training. In such a scenario, cloud users often encounter the GPU unavailability issue due to *instance limit exceeded* or *insufficient instance capacity* [5], bringing a long queueing delay for users' training jobs. On the other hand, large model training with homogeneous GPUs can lead to resource wastage for cloud providers. As an example, the existing scheduler needs to allocate 4 NVIDIA A100 GPUs for a training job, if its computational power requirement (in FLOPs) of the large model is estimated as 3.2 GPUs, which in turn increases the training budget for users. Though such low GPU utilization issues can be mitigated by GPU sharing techniques, they inevitably introduce noticeable performance interference among jobs [6]. As a result, it has become increasingly compelling to effectively leverage heterogeneous GPUs for facilitating large model training in the cloud.

However, it is non-trivial to provision a cluster of heterogeneous GPUs for effectively training large models in the cloud, which faces several key challenges summarized below.

• ***Heterogeneous* GPU allocation.** Previous works on scheduling training jobs in heterogeneous clusters mainly focus on improving resource utilization [7]–[9] and ensuring fairness [10]–[12], from the *cloud providers'* perspective. Nevertheless,

*Corresponding author: Fei Xu (fxu@cs.ecnu.edu.cn). This work was supported in part by the NSFC under Grants 62372184 and 62202266, the Science and Technology Commission of Shanghai Municipality under Grant 22DZ2229004, the NSF under Grant OIA-2327452, the Louisiana BoRSF under Grant LEQSF(2024-27)-RD-B-03, the Major Key Project of PCL under Grants PCL2024A06 and PCL2022A05, and the Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

these studies neglect the cost efficiency of training jobs, which has proved to be the primary concern of cloud users [13]. Moreover, it encounters heavy computational overhead to identify a cost-efficient GPU allocation plan from a more extensive search space of heterogeneous GPU clusters compared to that of homogeneous GPU clusters. According to Sec. II-B, selecting an optimal GPU allocation plan from a vast search space can save the monetary cost by up to $1.41\times$ while guaranteeing training performance compared to the best homogeneous plan.

• ***Complex* stage placement.** Cloud users require partitioning model layers into stages and placing them on GPUs, referred to as the *stage placement* [3], when applying pipeline parallelism in heterogeneous clusters. Previous research mainly considers GPU computing capability [14] or memory capacity [15], separately, which misses the co-optimization opportunities of the heterogeneous GPU sequence and the varying demands of each stage for computing and memory resources. A more recent work Metis [16] balances layers across stages based on GPU computation capabilities to reduce computation time, but it does not adequately consider the impact of communication time. As illustrated in Sec. II-C, an optimal stage placement plan that jointly accounts for GPU computing capability and memory capacity can reduce training time by up to 31.9%.

To address the challenges above, we design *Espresso* (as shown in Fig. 1), a cost-efficient GPU provisioning framework for minimizing training costs while ensuring the Service Level Objectives (SLOs) of large model training in heterogeneous environments. We make the following contributions.

▷ We devise a heterogeneity-aware GPU allocator (Sec. III-B) that guarantees the large model training SLO on heterogeneous GPUs. To quickly identify a cost-efficient provisioning plan, we propose a *cost tree-based* provisioning strategy to prioritize searching allocation plans with lower costs and reduce unnecessary branches by multi-dimensional pruning methods.

▷ We design a resource-aware stage placer (Sec. III-C) that defines a *compute-memory ratio (CMR)* to jointly consider the impact of GPU memory capacity and computing capability on training performance. We place the front stages on GPUs with lower *CMR* and the rear stages on GPUs with higher *CMR*, so as to effectively balance the pipeline and overlap communication with computation, significantly reducing training time.

▷ We implement a prototype of *Espresso* (https://github.com/icloud-ecnu/Espresso) in JuChi Cloud[1] and conduct extensive prototype experiments with four representative Transformer-based models and four different GPU types. The experimental results show that *Espresso* can maintain DNN training performance while saving up to 49.8% of monetary costs, compared to the existing solutions.

II. Background and Motivation

In this section, we illustrate how the heterogeneous GPUs and stage placement can impact the performance and cost of large model training in a cluster of heterogeneous GPUs.

[1]JuChi Cloud (https://matpool.com) is a public GPU cloud in China.

TABLE I: Four representative heterogeneous GPUs deployed in our experiments in JuChi Cloud.

GPU Type	Mem. (GB)	Max. Quota	TFLOPS	Price (¥/h)
A6000	48	8	154.8	6
A30	24	16	165	4
RTX3090	24	16	142	2.5
A4000	16	16	76	2

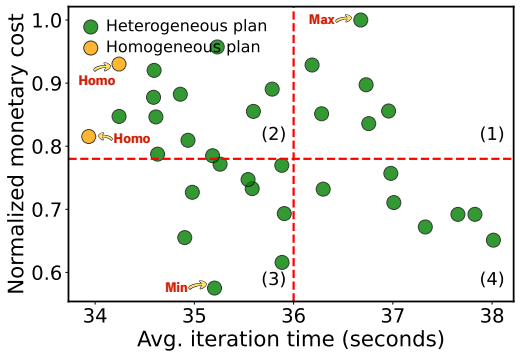


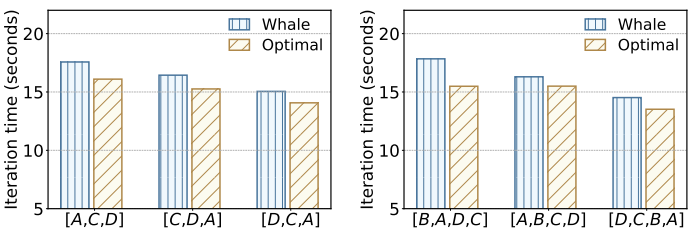
Fig. 2: Training iteration time and normalized monetary cost of 35 effective GPU allocation plans (denoted as [GPUtype : #num]) for LLaMA-3B with the training SLO set as 40 seconds. Min refers to the cheapest heterogeneous allocation plan, *i.e.,* [A6000: 1, RTX3090: 5], and Max refers to the most expensive heterogeneous allocation plan, *i.e.,* [A6000: 1, A30: 7]. The two homogeneous and feasible allocation plans are [A30: 7], [A30: 8].

A. Distributed Training of Large Models

Training large models involves processing billions of parameters on massive datasets typically with 3D parallelism, which combines three training parallelism strategies, *i.e.,* data parallelism (DP) [17], tensor parallelism (TP) [18], and pipeline parallelism (PP) [19]. Specifically, DP divides the data into multiple *mini-batches* and distributes them across numerous computing units, with each unit maintaining a full model replica. To particularly deal with large models, TP slices the model layer into several tensor chunks, while PP further partitions the model into several *stages* and executes them in sequence on different computing units. In more detail, each stage consists of several contiguous model layers and divides the mini-batch into multiple micro-batches. Accordingly, the output of one stage serves as the input for the next stage (*i.e.,* stage dependency). In particular, each stage synchronizes gradients *only* after the *forward pass* and *backward pass* of all its micro-batches in one iteration [20]. This allows the computation of the previous stage and the gradient synchronization communication of the current stage to be executed simultaneously. Accordingly, it is critical to decide the *GPU allocation plan* and the *stage placement plan* (*i.e.,* how to partition model layers to stages and place them on the sorted GPUs), so as to balance the computation and communication.

B. Heterogeneous GPU Allocation

To examine the training performance and cost of heterogeneous GPU allocations, we conduct a set of experiments by training LLaMA-3B [21] with the Alpaca dataset [22]



(a) PP Size = 3 (b) PP Size = 4

Fig. 3: Comparison of training iteration time of LLaMA-1.8B for different stage placement plans in terms of GPU sequences (x-axis labels) and model partitioning strategies (Whale and `Optimal`). A, B, C, and D denote NVIDIA A6000, A30, RTX3090, and A4000 GPUs, respectively.

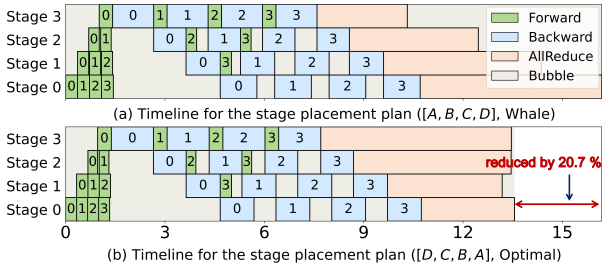


Fig. 4: Timelines of one training iteration of LLaMA-1.8B with Whale and `Optimal` stage placement plans, where A, B, C, and D denote NVIDIA A6000, A30, RTX3090, and A4000 GPUs, respectively.

on a cluster of three GPU types including NVIDIA A6000, A30, and RTX3090 as listed in Table I. Specifically, we set the training job SLO as 40 seconds, the sequence length as $4,096$, and the global batch size as 32. We identify 35 effective GPU allocation plans, among which only 2 are homogeneous plans. In particular, we employ the stage placement strategy in Whale [15] for heterogeneous GPU plans.

As illustrated in Fig. 2, we observe that there exists significant variability in cost-effectiveness among different GPU allocation plans, which are scattered in the four quadrants. Specifically, the GPU allocation plans in quadrant-③ exhibit the highest cost-effectiveness. The most cost-efficient allocation plan (*i.e.,* `Min`) in quadrant-③ can save up to $1.73\times$ in monetary cost and reduce training time by $1.05\times$, as compared to the least cost-efficient plan (*i.e.,* `Max`) in quadrant-①. Furthermore, the optimal heterogeneous GPU allocation plan can reduce the monetary cost by up to $1.41\times$ while guaranteeing training performance, compared to the best homogeneous plan. This provides an opportunity to optimize heterogeneous GPU resource allocations for large model training, aiming to minimize the monetary cost while guaranteeing the model training performance in the cloud.

C. Inadequate Stage Placement

We investigate the performance impact of stage placement on large model training. Specifically, we configure 12 different stage placement plans in terms of 6 GPU sequences and 2 model partitioning strategies, including Whale [15] and the `Optimal` partitioning strategy by the exhaustive search. We set the DP size as 2, and the PP size as 3 and 4, respectively. For each PP size, we select 3 different GPU sequences to illustrate the training performance variance. As shown in Fig. 3, we observe that the `Optimal` plan (*i.e.,* 4 stages placed with $[D, C, B, A]$) can achieve the training performance gains by up to 31.9% as compared to an inadequate plan (*i.e.,* 4 stages placed with $[B, A, D, C]$) obtained by Whale. Even under the same GPU sequences, the `Optimal` partitioning strategy still yields a moderate training speedup by 6.9%-15.2% compared to Whale in heterogeneous clusters.

We further look into the stage execution timelines for training LLaMA-1.8B with Whale and `Optimal` stage placement plans. As shown in Fig. 4, Whale experiences longer training time by up to 20.7% as compared to the `Optimal` plan, due to the two following facts. *First,* Whale solely focuses on the GPU memory requirements of stages [15]. As the front stages (*e.g.,* `stage0`, `stage1`) of a model exhibit high memory requirements to store the activation values, Whale tends to allocate GPUs with large memory capacity to these stages, thereby leading to prolonged communication time for the front stages and thus the ineffective overlap of stage communication and computation time. *Second,* the `Optimal` plan jointly considers the computing capability and memory capacity of GPUs in the stage placement. It deliberately allocates GPUs with inferior computing capability and large memory capacity to the front stages, so that fewer layers are put to these front stages while more layers are allocated to the rear stages (*e.g.,* `stage2`, `stage3`), as compared to Whale. Such an arrangement allows the long communication time of the rear stages to overlap with the computation time of the front stages, greedily minimizing the maximum stage execution time.

Summary. *First,* provisioning a cluster of heterogeneous GPU resources for large model training has a *substantial optimization space* in reducing monetary cost while guaranteeing the large model training performance in the cloud. *Second,* an adequate stage placement plan can greedily overlap the stage computation and communication time by jointly considering the computing capability and memory capacity of GPUs, thereby decreasing the model training time significantly.

III. System Design

In this section, we design *Espresso* illustrated in Fig. 5, a cost-efficient GPU provisioning framework to minimize the monetary cost while guaranteeing the performance of large model training in the cloud. Once users submit a training job (*i.e.,* a large model, datasets, SLOs, and training epochs), the *Profiler first* acquires the training time and GPU memory consumption of different layers on heterogeneous GPUs from a lightweight profiling tool ①. Leveraging the profiled job statistics, the *GPU Allocator then* employs a cost tree-based provisioning strategy to generate a cost-efficient GPU allocation plan ② (Sec. III-B). It further utilizes a resource-aware stage placement strategy in *Stage Placer* to properly partition model layers into stages and place them on the provisioned GPUs with the objective of minimizing the training time ③ (Sec. III-C). *Finally,* the training job is deployed with the GPU allocation and stage placement plans in the cloud ④.

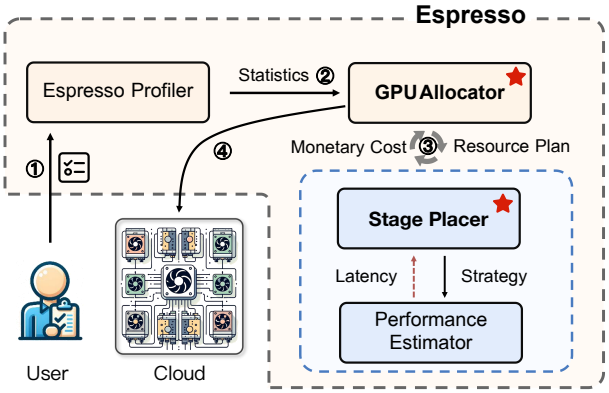


Fig. 5: Overview of *Espresso* architecture.

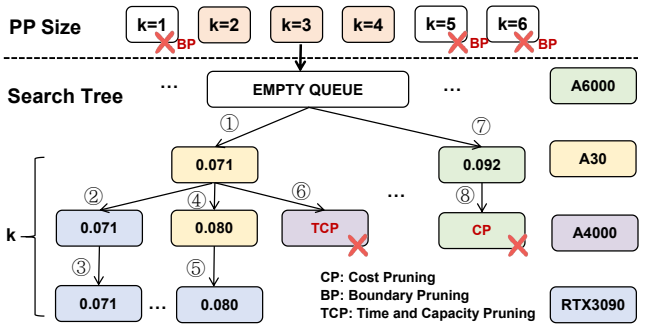


Fig. 6: An illustration of cost tree-based GPU provisioning strategy, where the numbers in a circle indicate the priorities of searching determined by the upper-bound cost of a partial plan (*i.e.,* Eq. (4)). Path [A30, RTX3090, RTX3090] and Path [A30, A30, RTX3090] are valid paths, which are input to *Stage Placer* to compare the cost efficiency.

A. Problem Formulation

Given a set of heterogeneous GPU types (or instance types) $\mathcal{T}$, quotas Lim_t (*i.e.,* the available GPUs for each type $t \in \mathcal{T}$), and a job with its training SLO (*i.e.,* T_{SLO}), *Espresso* provisions a set of GPUs and generates a stage placement plan, aiming to minimize the training cost C while guaranteeing T_{SLO}. Such an optimization problem can be formulated as

$$\min_{\mathcal{N},\mathcal{G}} \quad C = z \times T_{iter}^{\mathcal{N},\mathcal{G}} \times \sum_{t\in\mathcal{T}}(c_t \times n_t) \tag{1}$$

$$\text{s.t.} \quad z \times T_{iter}^{\mathcal{N},\mathcal{G}} \leq T_{SLO}, \tag{2}$$

$$n_t \leq Lim_t, \quad \forall t \in \mathcal{T}, n_t \in \mathbb{Z} \tag{3}$$

where z denotes the number of training iterations. $\mathcal{N}$ denotes the GPU allocation plan, specifying the allocated GPU types and their quantities. $\mathcal{G}$ represents the stage placement plan, indicating the model partitions (*i.e.,* stages) and the mapping of stages to allocated GPUs. Given $\mathcal{N}$ and $\mathcal{G}$, $T_{iter}^{\mathcal{N},\mathcal{G}}$ denotes the training iteration time, while c_t and n_t denote the unit price and provisioned number for each GPU type t, respectively. Constraint (2) guarantees the model training time, while Constraint (3) indicates that n_t is below the user quota.

Problem analysis. Our optimization problem in Eq. (1) can be decomposed into two sub-problems: *GPU allocation* and *stage placement*. For the first one, the diversity of GPU types and the exponential increase in possible combinations produce an enormous search space for the optimal allocation plan. This results in the low efficiency of existing exhaustive search algorithms [23]. For the second one, the stage placement requires partitioning model layers into a number of stages and placing them on the allocated GPUs, which can be reduced to the form of a *traveling salesman problem*, already known as an NP-hard problem [24].

B. GPU Allocator

To speed up the heterogeneous GPU allocation while minimizing the monetary cost, we prioritize searching for the allocation plans with lower costs in the large solution space. To this end, we propose a *cost tree-based* provisioning strategy, where the allocation plans are represented by the paths from the root node to child nodes in a tree structure (*i.e.,* cost tree). During the construction of cost trees, we keep tracking the training cost of each path and employ multi-dimensional pruning methods to reduce unnecessary branches.

Cost tree construction. As illustrated in Fig. 6, each node in the cost tree represents the type and number of allocated GPUs where a stage is deployed. In particular, the GPU number is determined by the DP size and the number of stages is determined by the PP size k. The path from the root node to a leaf node represents a complete GPU allocation plan $\mathcal{N}$ with the length of k, while the other paths correspond to partial plans $\widehat{\mathcal{N}}$. To simplify the tree construction, the duplicate paths (*e.g.,* $[B, A, C]$) are discarded given a path (*e.g.,* $[A, B, C]$) in the tree. To guide the search, we assign a value to each path that denotes the *upper-bound* cost of the corresponding allocation plan. For a *partial* plan $\widehat{\mathcal{N}}$, its upper-bound cost is calculated as the minimum value among the training costs of homogeneous GPUs for the remaining stages, given by

$$C_{upper}(\widehat{\mathcal{N}}) = \min_{t\in\mathcal{T}}(\texttt{HomoConfig}(\widehat{\mathcal{N}}, t)), \tag{4}$$

where $\texttt{HomoConfig}(\widehat{\mathcal{N}}, t)$ denotes the training cost of one iteration using the GPU type t for all subsequent stages. For simplicity, we denote the allocation plan for $\texttt{HomoConfig}(\widehat{\mathcal{N}}, t)$ as $\mathcal{N}_h = \widehat{\mathcal{N}} \cup \{\underbrace{g_t, g_t, \ldots, g_t}_{k-|\widehat{\mathcal{N}}|}\}$, where g_t denote a GPU of the type t.

Cost model. The calculation of the lower-bound cost of $\mathcal{N}_h$ (*i.e.,* $\texttt{HomoConfig}(\widehat{\mathcal{N}}, t)$) relies on the stage placement of a job, which brings non-negligible computation overhead. To mitigate that, we build a cost model by neglecting the training communication overhead. Specifically, the lower bound of an iteration time T_{iter}^{lower} can be calculated by dividing the total computational load $W(l_{total})$ across all model layers l_{total} by the total computing capability of $\mathcal{N}_h$ [25], which is given by

$$T_{iter}^{lower} = \frac{W(l_{total})}{p \times \sum_{t\in\mathcal{N}_h} r_t}, \tag{5}$$

where r_t is the actual GPU computing capability, profiled by running an arbitrary training job on a GPU of the given type t. p is the GPU active time ratio, which can be calculated as

$$p = \frac{m}{m + k - 1}, \tag{6}$$

where k is the PP size and m is the number of micro-batches. This is because the GPU active time of each stage only consists of m forward and backward passes for each iteration, while the bubble time of each stage is $k-1$ forward and backward passes in one iteration [19], due to data dependencies between stages. As a result, we can predict the lower-bound cost of $\mathcal{N}_h$ according to Eq. (1) and Eq. (5), which is given by

$$\texttt{HomoConfig}(\widehat{\mathcal{N}}, t) = z \times T_{iter}^{lower} \times \sum_{t' \in \mathcal{N}_h} (c_{t'} \times n_{t'}). \quad (7)$$

Multi-dimensional pruning methods. To speed up the searching efficiency of allocation plans, we further propose multi-dimensional pruning methods as illustrated in Fig. 6.

(1) Boundary pruning (BP). As the number of GPUs is determined by the PP size k and DP size d, BP calculates the boundary values for k and d to narrow down the search space. *First*, we set the lower bound k_{lower} and the upper bound k_{upper} as the number of stages required to greedily place model layers on GPUs with the largest and smallest memory, respectively. *Second* is determining the lower bound d_{lower} and the upper bound d_{upper} for d. As the batch size is typically set as a power of two, we typically set the d in the range of $[d_{lower} = 2^0, d_{upper} = 2^{\lfloor \log_2(x) \rfloor}]$ for x GPUs.

(2) Cost pruning (CP). The branches with high costs can be *pruned early* to reduce the search space, by calculating the lower-bound cost c_{lower} of a partial allocation plan. To align with real-world scenarios, we first introduce the GPU memory fragmentation coefficient β (typically set to 80% [26]) to estimate the number of model layers l placed on the provisioned GPUs given a partial plan. After inputting the computational load $W(l)$ into Eq. (5), we can obtain the lower-bound iteration time, thereby calculating the training cost of a partial plan. If such a training cost exceeds the current optimal complete plan $\mathcal{N}$, such a partial plan is discarded.

(3) Time and Capacity pruning (TCP). TCP leverages the lower-bound iteration time t_{lower} and the maximum allocated memory m_{max} for a plan $\mathcal{N}_h$, which enables us to discard the plans whose t_{lower} exceed the SLOs or whose m_{max} are less than the lower-bound GPU memory (*i.e.,* m_{lower}) of the training job [25]. Specifically, we calculate t_{lower} by placing the remaining stages on the GPUs with the highest computing capability. Meanwhile, we calculate m_{max} by placing the remaining stages on the GPUs with the largest memory.

Alg. 1 elaborates *Espresso*'s GPU provisioning strategy using color-coded differentiation. *First,* each pair of PP and DP sizes corresponds to the construction of a cost tree. Boundary values are then calculated by functions `GetPPSizeRange` and `GetDPSizeRange` based on BP, and cost trees that do not require further exploration are discarded (lines 2-5). *Next,* a priority queue Q manages the exploration of potential GPU allocation plans, sorting them in ascending order based on these partial plans' $C_{upper}(\widehat{\mathcal{N}})$ (see Eq. (4)) (lines 6-8). Upon identifying a complete allocation plan, Alg. 1 enters the optimization step. In more detail, `stagePlacer` determines a stage placement by Alg 2, and a low-cost provisioning plan (including $\mathcal{N}$ and $\mathcal{G}$) is selected (lines 9-13). Alg. 1

Algorithm 1: *Espresso*: a cost tree-based GPU provisioning strategy.

Input : Available GPUs types $\mathcal{T}$, GPU quotas Lim_t, a job J with its training SLO T_{SLO}.
Output: GPU allocation plan $\mathcal{N}$, stage placement plan $\mathcal{G}$.

1 **Initialize**: $\mathcal{N} \leftarrow \emptyset$, $\mathcal{G} \leftarrow \infty$;
2 $d_{lower}, d_{upper} \leftarrow$ `GetDPSizeRange`$(\mathcal{T})$;// BP Pruning
3 **for** $d \in [d_{lower}, d_{upper}]$ **do**
4 $k_{lower}, k_{upper} \leftarrow$ `GetPPSizeRange`$(\mathcal{T}, J, d)$;
5 **for** $k \in [k_{lower}, k_{upper}]$ **do**
6 **Initialize:** a priority queue $Q \leftarrow (0, \emptyset, 0, 0)$, m_{lower}; // Lower-bound memory demand
7 **while** $Q \neq \emptyset$ **do**
8 $\_, \widehat{\mathcal{N}}, s, g_t \leftarrow Q$.top(); // s denotes the current stage.
  // Optimizing the complete plan
9 **if** $s = k$ **then**
10 $\widehat{\mathcal{G}} \leftarrow$ `stagePlacer`$(\widehat{\mathcal{N}}, k, d, J.layer)$;
11 **if** $C(\widehat{\mathcal{N}}, \widehat{\mathcal{G}}) < C(\mathcal{N}, \mathcal{G})$ **then**
12 Update $\mathcal{G} \leftarrow \widehat{\mathcal{G}}$, $\mathcal{N} \leftarrow \widehat{\mathcal{N}}$;
13 **continue**;
14 $c_{lower}, t_{lower}, m_{max} \leftarrow$ `pruneInfo`$(\widehat{\mathcal{N}}, k, d)$;
15 **if** $t_{lower} > T_{SLO} || m_{max} < m_{lower}$ **then**
16 **continue**; // TCP Pruning
17 **if** $c_{lower} > C(\mathcal{N}, \mathcal{G})$ **then**
18 **continue**; // CP Pruning
19 **for** $y \in [0, \min(k-s, Lim_t)]$ **do**
  // Search the number of g_t
20 $\widehat{\mathcal{N}}_n \leftarrow \widehat{\mathcal{N}} \cup \{\underbrace{g_t, g_t, \ldots, g_t}_{y}\}$;
21 $Q.push((C_{upper}(\widehat{\mathcal{N}}_n), \widehat{\mathcal{N}}_n, s+y, g_{t+1}))$;
22 **return** $\mathcal{N}$, $\mathcal{G}$;

adopts the function `pruneInfo` to calculate the lower-bound training cost c_{lower} and iteration time t_{lower} as well as the maximum allocated memory m_{max}, facilitating CP and TCP pruning (lines 14-18). The exploration step further evaluates the number of GPUs g_t that can be added to the partial allocation plan $\widehat{\mathcal{N}}$ (lines 19-21). *Finally, Espresso* outputs the allocation plan $\mathcal{N}$ and the stage placement plan $\mathcal{G}$ (line 22).

C. Stage Placer

Given a GPU allocation plan $\mathcal{N}$, we proceed to partition model layers into stages and place them on the provisioned heterogeneous GPUs, aiming to minimize the training iteration time. This is equivalent to minimizing the completion time of the slowest stage. Accordingly, our optimization objective in Eq. (1) can be reduced to

$$\min_{\mathcal{G}} T_{iter}^{\mathcal{N},\mathcal{G}} = \max_{i \in [0,k)} \left(T_{\mathcal{G}_i}^{comp} + T_{\mathcal{G}_i}^{comm}\right) \quad (8)$$

$$= T_{\mathcal{G}_0}^{comp} + \max_{i \in [0,k)} \left(T_{\mathcal{G}_i}^{comm} - \sum_{j \in [0,i)} T_{\mathcal{G}_j}^{b}\right). \quad (9)$$

In Eq. (8), $\mathcal{G}_i$ denotes the stage placement plan for the i-th stage, which includes the model partitions and the mapping of stages to allocated GPUs. As illustrated in Fig. 7, $T_{\mathcal{G}_i}^{comp}$ represents the time for the i-th stage to complete both the

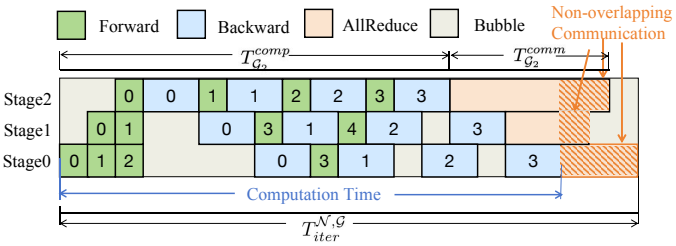


Fig. 7: Timeline of an iteration for a training job with 3 stages.

forward and backward passes on $\mathcal{G}_i$ in one iteration. $T^{comm}_{\mathcal{G}_i}$ is the gradient synchronization time for data parallelism. Due to the stage dependency, the pipeline computation (*i.e.,* $T^{comp}_{\mathcal{G}_0}$) finishes when `stage0` completes its final backward pass. As the communication of a stage can be overlapped with the computation of preceding stages, the non-overlapping communication time for a stage i is calculated as $T^{comm}_{\mathcal{G}_i} - \sum_{j=0}^{i-1} T^{b}_{\mathcal{G}_j}$, where $T^{b}_{\mathcal{G}_j}$ is the backward execution time of a stage j.

Compute-memory ratio (*CMR*). As motivated by Sec. II-C, an adequate stage placement should consider both the computing capability and memory capacity of GPUs. To minimize the training iteration time, we define a metric named *CMR* as

$$CMR(t,s) = \frac{F(t)}{M(t)} \times \alpha(t,s), \tag{10}$$

where $F(t)$ and $M(t)$ represent the computing capability in `TFLOPS` and memory capacity in `GB` of a GPU type t, respectively. As shown in Fig. 7, the communication time to be overlapped varies across different stages. To greedily minimize the non-overlapping communication time, we introduce the metric $\alpha(t,s)$ to characterize the computing and memory demands for each stage s. Specifically, we define $\alpha(t,s)$ as $\frac{\max(l^{t,s}_m, l^{t,s}_c)}{l^{t,s}_c}$, where $l^{t,s}_m$ is the maximum number of layers allocated to a GPU type t under the memory constraint, while $l^{t,s}_c$ is the maximum number of layers that cannot exceed the communication time of `stage0`. In general, a GPU with lower *CMR* is more suitable for the front stages.

Resource-aware stage placement. Based on the analysis above, we design a *resource-aware* stage placement algorithm in Alg. 2 using color-coded differentiation. *First, Espresso* initialize two lists. The first list L_m contains the maximum number $l^{t,s}_m$ of layers as elaborated before. The second list L_f consists of the number l^t_f of layers to be placed on a GPU type t by considering load balancing among stages [15] (line 3). The maximum layers l_{max} for `stage0` is then determined based on $l^{t,0}_m$ (line 4). *Next, Espresso* enumerates the number of layers l_{s_0} for `stage0` to greedily balance the pipeline and overlap the computation and communication (line 5). In more detail, *Espresso* initializes a candidate stage placement plan $\widehat{\mathcal{G}}$ and the maximum communication time T^{ol}_s for stage s within the communication time of `stage0` (lines 6-7). For the subsequent stages, we update T^{ol}_s as the sum of T^{ol}_{s-1} and the backward execution time $T^{b}_{\widehat{\mathcal{G}}_{s-1}}$ and GPUs' *CMR* according to Eq. (10) (lines 9-10). *Espresso* further explores the candidate placement plan $\widehat{\mathcal{G}}$ by selecting the GPU with the minimum

Algorithm 2: `stagePlacer`: a resource-aware stage placement algorithm.

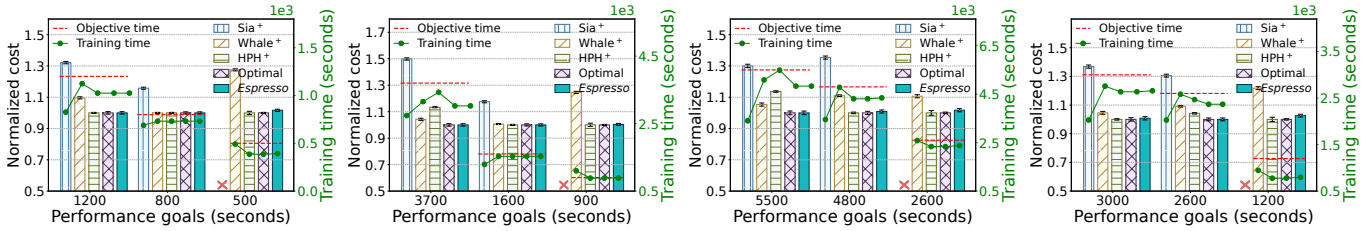
1 **Input:** GPU allocation plan $\mathcal{N}$, PP size k, DP size d, number of model layers l_{total}.
2 **Output:** Stage placement plan $\mathcal{G}$.
3 Calculate (L_m, L_f) based GPU load & memory in $\mathcal{N}$;
4 **Initialize** : $\mathcal{G} \leftarrow \emptyset$; $l_{max} \leftarrow \max_t(l^{t,0}_m)$;
// l_{s_0} denotes the number of layers for `stage0`
5 **for** $l_{s_0} \in [1, l_{max}]$ **do**
6 GPU type $t \leftarrow$ GPU with the *minimum CMR*;
7 **Initialize**: a candidate plan $\widehat{\mathcal{G}} \leftarrow ((g_t, l_{s_0}))$, $T^{ol}_0 \leftarrow T^{comm}_{\widehat{\mathcal{G}}_0}$;
8 **for** $s \in [1, k-1]$ **do**
9 $T^{ol}_s \leftarrow T^{b}_{\widehat{\mathcal{G}}_{s-1}} + T^{ol}_{s-1}$;
10 Update GPUs' *CMR* $\leftarrow$ Eq. (10);
11 GPU type $t \leftarrow$ GPU with the *minimum CMR*;
12 Set layer number of stage s as $l \leftarrow \min(l^{t,s}_m, l^t_f)$;
13 $\widehat{\mathcal{G}}$.add((g_t, l));// g_t is a GPU of the type t.
14 $\widehat{\mathcal{G}} \leftarrow$ `AdjustToCoverAllLayers`($\widehat{\mathcal{G}}$);
15 **if** `Estimator`($\widehat{\mathcal{G}}$) < `Estimator`($\mathcal{G}$) **then**
16 $\mathcal{G} \leftarrow \widehat{\mathcal{G}}$;
17 **return** $\mathcal{G}$;

CMR and assigning an appropriate number of layers to stages (lines 11-13). To address the potential issue of layer fragmentation (*i.e.,* few layers remaining to be assigned), *Espresso* adds the remaining layers to the stages (except `stage0`) in $\widehat{\mathcal{G}}$ one by one (function `AdjustToCoverAllLayers`), as long as the memory constraint satisfies (line 14). Finally, *Espresso* leverages the `Estimator` [27] to estimate the iteration time for $\widehat{\mathcal{G}}$ and then identifies an adequate placement plan $\mathcal{G}$ with the minimum iteration time (lines 15-17).

Remark. The complexity of Alg. 1 is $\mathcal{O}(\sum_{k=k_{lower}}^{k_{upper}} C^k_x \times \lfloor \log_2(x) \rfloor)$, where k_{lower} and k_{upper} denote the lower bound and the upper bound for the PP size k, respectively. C^k_x is a selection of k GPUs from the total number (*i.e.,* x) of GPUs limited by the user quota, and $\log_2 x$ indicates the number of possible DP sizes. The complexity of Alg. 2 is $\mathcal{O}(l_{max} \times k)$, where l_{max} denotes the maximum layers for `stage0`. As k is typically an integer within tens [20], its complexity can be roughly linear to the number of model layers. Accordingly, the computation overhead of *Espresso* is practically acceptable due to our multi-dimensional pruning methods, which will be validated by Sec. V-C.

IV. System Implementation

We implement a prototype of *Espresso* on JuChi Cloud using DeepSpeed [28] v0.13.1 and PyTorch [29] v2.0.1, with over 1,200 lines of Python and Linux Shell scripts. The source code is publicly accessible on GitHub (https://github.com/icloud-ecnu/Espresso). Specifically, we integrate our model partitioning method (in the *Stage Placer*) as a plug-in module into DeepSpeed to dynamically partition the large model for efficient training in heterogeneous clusters. Additionally, our lightweight profiling tool leverages `llm-analysis` [30] to estimate the memory usage and training time for various



(a) LLaMA-1.3B (b) LLaMA-3B (c) LLaMA-7B (d) ViT

Fig. 8: Comparison of training time and monetary cost achieved by the Sia$^+$, Whale$^+$, HPH$^+$, Optimal, and *Espresso* under different SLOs.

GPU types. It analyzes model parameters (*i.e.,* hidden size and model layers) and GPU specifications (*i.e.,* computing capability and memory capacity) without deploying model training jobs on GPUs, thereby significantly reducing our profiling cost. We further devise a *Performance Estimator* based on topological sorting and `llm-analysis` to predict the model training iteration time by analyzing task dependencies, which include intra-stage and inter-stage dependencies. Inter-stage dependencies include activation and gradient transmissions, while intra-stage dependencies are managed by the `schedule.TrainSchedule` in DeepSpeed.

Discussion. *Espresso* can be applied in both public clouds and private clouds, by extending the *cost* tree to other key metrics such as GPU scarcity or GPU utilization. By adequately defining the cost model, *Espresso* can guarantee the training performance of large models, while minimizing the usage of scarce GPUs or improving the GPU utilization in the cloud.

V. Performance Evaluation

In this section, we evaluate *Espresso* by carrying out a set of prototype experiments on JuChi Cloud. We seek to answer the two following questions:

- **Effectiveness:** Can our GPU provisioning strategy in *Espresso* reduce monetary cost while guaranteeing large model training performance in the cloud? (Sec. V-B)
- **Overhead:** How much runtime overhead does *Espresso* practically bring in GPU allocation?

A. Experimental Setup

Configurations of training environments and workloads. We provide four representative GPU types for heterogeneous environments as listed in Table I. The cluster provides network isolation with an inter-instance bandwidth of 2.5 Gbps. Considering that various popular models (*e.g.,* GPT [31] and LLaMA [21]) are based on the Transformer [32] architecture, we select four representative Transformer-based large models listed in Table II to evaluate the performance of *Espresso*. Specifically, we train the ViT [33] on the ImageNet-1K dataset for image classification. For language modeling, the LLaMA-1.3B employs the Wikipedia dataset, whereas the LLaMA-3B and LLaMA-7B models are trained on the Alpaca dataset [22] for question-answering tasks.

Baselines and metrics. We compare *Espresso* with three state-of-the-art strategies: (1) Sia$^+$ [34] selects 2^n ($n \in \mathbb{N}$) GPUs of the same type in heterogeneous environments. (2) Whale$^+$ [15] places stages on GPUs in decreasing order of GPU memory capacity and combines the model partitioning strategy based on load balancing. (3) HPH$^+$ [14] places stages on GPUs in ascending order of GPU computing capability and combines the model partitioning strategy based on integer programming. We integrate our GPU provisioning strategy with Whale$^+$ and HPH$^+$ to support GPU resource provisioning in heterogeneous environments. We also evaluate the performance gap between *Espresso* and the Optimal strategy obtained through the exhaustive search. We focus on three metrics: training time, monetary cost, and runtime overhead.

TABLE II: Large models and datasets evaluated in our experiments.

Workload	#Layer	Hidden size	#Param.	Dataset
LLaMA-1.3B [21]	26	2,048	1.3B	Wikipedia[2]
LLaMA-3B	45	2,560	3B	Alpaca [22]
LLaMA-7B	61	3,200	7B	
ViT [33]	70	2,048	3.4B	ImageNet-1K [35]

B. Effectiveness of Espresso

Can *Espresso* minimize monetary cost while ensuring the model training performance? We train the four models in Table II with Sia$^+$, Whale$^+$, HPH$^+$, Optimal, and *Espresso* for 100 iterations by setting different training time SLOs. As shown in Fig. 8, *Espresso* successfully meets the training time objectives while reducing monetary cost by up to 49.8% compared to other strategies. The Sia$^+$ strategy, while effective in reducing training time, primarily achieves this through an over-allocation of GPUs. It focuses on assigning a single type of GPUs to a job, neglecting the potential benefits of allocating heterogeneous GPUs. To meet the SLO, Sia$^+$ always necessitates allocating more resources than the model requires, thereby resulting in higher monetary costs inevitably. In particular, the mark "×" labeled in Fig. 8 represents the cases that Sia$^+$ cannot identify any suitable GPU allocation plans due to the user quota limitation and SLO constraints.

Despite utilizing our GPU provisioning strategy in *Espresso*, both Whale$^+$ and HPH$^+$ exhibit shortcomings in their stage placement strategies (as discussed in Sec. II-B). Specifically, Whale$^+$ tends to allocate more layers in the front stages, increasing the communication time for gradient synchronization. Conversely, HPH$^+$ fails to consider the intensive memory demands of the front stages, allocating GPUs with insufficient

[2]Downloads: https://huggingface.co/datasets/legacy-datasets/wikipedia

TABLE III: Comparison of GPU allocation plans achieved by Sia$^+$, Optimal, and *Espresso* for LLaMA-7B and ViT. The GPU allocation plan is defined as $[\#A6000, \#A30, \#3090, \#A4000]$.

	Workload/ Strategy	Performance SLO (seconds)		
		5,500	4,800	2,600
LLaMA	Sia$^+$	[8, 0, 0, 0]	[8, 0, 0, 0]	N/A
	Optimal	[4, 0, 0, 1]	[4, 1, 0, 0]	[8, 12, 0, 4]
	Espresso	[4, 0, 0, 1]	[4, 1, 0, 0]	[8, 12, 0, 4]
		3,000	2,600	1,200
ViT	Sia$^+$	[0, 4, 0, 0]	[0, 4, 0, 0]	N/A
	Optimal	[0, 1, 2, 0]	[0, 2, 1, 0]	[8, 8, 8, 16]
	Espresso	[0, 1, 2, 0]	[0, 2, 1, 0]	[8, 8, 8, 16]

TABLE IV: Comparison of stage placement plans employed by Whale$^+$, HPH$^+$, and *Espresso*. The stage placement plan is denoted as two tuples, where the upper denotes the allocated GPU type (*i.e.,* A, B, C, and D denote A6000, A30, RTX3090, and A4000, respectively) for each stage, while the lower denotes the number of layers partitioned to each stage.

Model (SLO)	Whale$^+$	HPH$^+$	*Espresso*
LLaMA-1.3B (500 seconds)	$[A, B, B, D]$ $[9, 6, 6, 5]$	$[D, A, B, B]$ $[5, 6, 7, 8]$	$[D, A, B, B]$ $[6, 6, 6, 8]$
LLaMA-3B (3,700 seconds)	$[A, A, C]$ $[18, 18, 9]$	$[C, A, A]$ $[9, 18, 18]$	$[A, A, C]$ $[19, 17, 9]$
LLaMA-7B (5,500 seconds)	$[A, A, A, A, D]$ $[14, 14, 14, 15, 4]$	$[D, A, A, A, A]$ $[1, 14, 15, 15, 16]$	$[A, A, A, A, D]$ $[14, 14, 15, 14, 4]$
ViT (1,200 seconds)	$[A, C, B, D, D]$ $[20, 15, 18, 8, 9]$	$[D, D, C, A, B]$ $[14, 13, 13, 14, 16]$	$[D, D, C, A, B]$ $[14, 14, 14, 14, 14]$

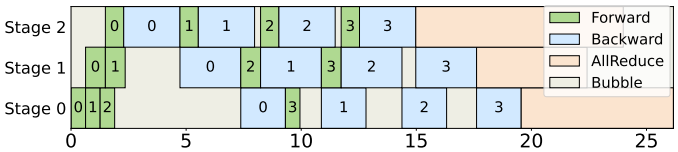


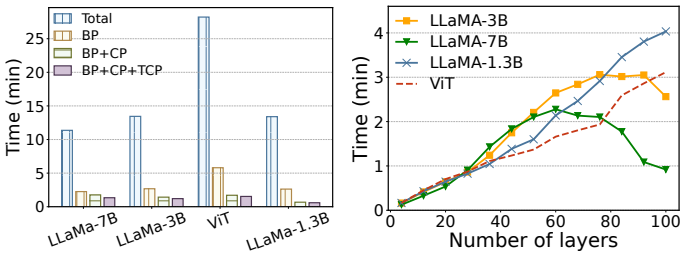
Fig. 9: Timeline of one iteration for ViT, partitioned into three stages denoted as the number of layers (*i.e.,* $[22, 16, 30]$) placed on the GPU allocation $[RTX3090, A4000, A6000]$.

memory to stages. This oversight leads to the short execution time at the front stages, resulting in the prolonged pipeline computation time. In comparison, *Espresso* maintains a cost variance within 4.5% of the Optimal strategy, showcasing its effective resource provisioning strategies. Furthermore, unlike the Optimal strategy, which incurs exponential computation overhead, *Espresso* generates GPU resource provisioning plans in just a few minutes.

Can *Espresso* provide a cost-efficient GPU allocation plan? To analyze the efficiency of our GPU provisioning strategy in *Espresso*, we look into the GPU allocation plans for LLaMA-7B and ViT under three strategies: Sia$^+$, Optimal, and *Espresso*. As elaborated in Table III, we observe that *Espresso* successfully identifies the best GPU allocation plan consistent with the Optimal strategy using our cost tree-based GPU provisioning strategy. In contrast, the Sia$^+$ strategy significantly narrows down the search space of GPU allocations by restricting the number of GPU selections to 2^n. As a result, it cannot fully exploit the benefits of a heterogeneous GPU environment and always leads to over-provisioning, thereby incurring unnecessary monetary costs. Moreover, under limited resources and/or high computational demands (*e.g.,* meeting stringent SLOs), Sia$^+$ fails to provide any allocation plans, resulting in job launch failures. In contrast, *Espresso* effectively leverages diverse GPU types, offering flexible plans that ensure job launches successfully.

Can *Espresso* generate an adequate stage placement plan? We compare the placement plans of four models under three strategies: Whale$^+$, HPH$^+$, and *Espresso*, as shown in Table IV. According to the results in Fig. 8, *Espresso* significantly reduces training time, with a maximum savings of up to 25.3% compared to Whale$^+$ and HPH$^+$. As illustrated in Table IV, Whale$^+$ assigns more layers to the front stages, balancing the pipeline but increasing the communication time for gradient synchronization. In scenarios with longer computation time, Whale$^+$ performs better (*i.e.,* LLaMA-3B) due to a more balanced pipeline. However, in scenarios with shorter computation time, the increased communication time leads to performance degradation (*i.e.,* ViT). Conversely, HPH$^+$ selects GPUs with smaller memory and allocates fewer layers to the front stages, increasing the number of layers in the rear stages, thus effectively overlapping communication time. In scenarios where communication time is dominant, HPH$^+$ performs better (*i.e.,* LLaMA-1.3B). Nevertheless, HPH$^+$ does not fully account for the high memory demand in the front stages, which can cause the pipeline imbalance and prolong the training time (*i.e.,* LLaMA-7B). When having the same GPU sequence (*i.e.,* the mapping of stages to allocated GPUs), the training time of *Espresso* is $0.8\% - 2.6\%$ longer than that of HPH$^+$. This is primarily due to HPH$^+$ employing integer programming to determine the optimal model partitioning plan. However, such an approach can result in substantial runtime overhead (*i.e.,* hours) for large models.

How the stage placement of *Espresso* minimize the large model training time? To look into the stage placement plan of *Espresso*, we conduct experiments on the ViT with three stages on three different types of GPUs. We set the DP size to 2. As shown in Fig. 9, we observe that the computation and communication time are well overlapped among the three stages. According to our resource-aware stage placement algorithm based on *CMR*, the generated placement plan is ([RTX3090, A4000, A6000], $[22, 16, 30]$). Initially, we obtain the best number of layers l_{s_0} for `stage0` as 22 layers. As the A4000 cannot support 22 layers for `stage0` due to the memory constraint, we compare the *CMR* values of RTX3090 and A6000, which are 6.99 and 7.91, respectively, thus selecting the RTX3090 with a lower *CMR* value for placing such 22 layers. For `stage1`, we compare the *CMR* values of A4000 and A6000 (*i.e.,* 4.75 and 6.92, respectively), and then select the A4000 with a lower *CMR* value for placing



(a) Ablation of pruning methods (b) Models with different layers

Fig. 10: Computation overhead of *Espresso* resource provisioning algorithm with (a) multi-dimensional pruning strategies and (b) four modified large models by varying model layers.

13 layers. Similar to the procedure of previous stages, the final stage (`stage2`) utilizes the A6000 for placing 28 layers. However, such a placement plan causes fewer assigned layers than the model layers. Consequently, we adjust the second stage (`stage1`) to 16 layers and maintain 30 layers for the final stage (`stage2`) using the A6000. By carefully adjusting the type of GPUs and the number of layers placed in each stage, *Espresso* not only optimizes the pipeline computation time but also overlaps the communication time effectively, reducing the overall model training time.

C. Overhead of Espresso

To evaluate the effectiveness of multi-dimensional pruning methods (*i.e.,* BP, CP, TCP) and runtime overhead of *Espresso*, we conduct experiments with the four models listed in Table II and 56 available GPUs (*i.e.,* the maximum user quota) in Table I. BP quickly establishes a reasonable search boundary, significantly reducing the vast search space caused by the grid search and reducing the search time up to 79.4%-83.2%. Building upon this, CP further narrows the search space by fast identifying feasible costs, leading to a decrease of 84.6%-95.2% in the search time. Though the marginal impact of TCP is the minimum due to substantial prior reductions, it still contributes to faster pruning speeds, reducing the average search time by 88.3%-96.2%. The combined use of the three pruning strategies reduces the search time by an average of 92.4%. The computation overhead of the four models is only 1.4, 1.3, 1.6, and 0.6 minutes, respectively. Such overhead is negligible compared to the tens to hundreds of hours typically required to train large models.

To evaluate the computation overhead of *Espresso* across varying model sizes, we adjust the number of layers in four models. As depicted in Fig. 10(b), the computation overhead of resource provisioning generally increases linearly with the layer numbers. Nonetheless, as illustrated for LLaMA-3B and LLaMA-7B, turning points in the curve occur as the number of layers increases, attributed to GPU resource limitations within the cluster which curtails the search space. Consequently, the computation overhead declines once exceeding a specific layer number threshold. In sum, the computation overhead of the *Espresso* remains within four minutes for models of varying sizes on a heterogeneous cluster of 56 GPUs.

VI. Related Work

GPU resource provisioning. Many existing works (*e.g.,* Pollux [36], Optimus [37]) focus on provisioning homogeneous GPUs for training jobs. They often adjust GPU allocations and batch sizes dynamically using a training performance model. Meanwhile, there have been works on scheduling training jobs in heterogeneous clusters to enhance GPU resource utilization [7] and job fairness [10]. However, these prior works overlook the key metric (*i.e.,* training cost efficiency) optimization. A recent work Sia [34] selects one GPU type from heterogeneous GPUs for training, failing to exploit the advantages of GPU heterogeneity fully. *SpotDNN* [5] boosts training cost-efficiency in heterogeneous clusters, yet it overlooks integrating diverse training parallelization strategies. In contrast, *Espresso* focuses on heterogeneous GPU provisioning for large models training and proposes a cost tree-based provisioning strategy to quickly identify a cost-efficient plan.

Stage placement optimization. Most existing techniques (*i.e.,* PipeDream [20], and Galvatron [38]) utilize dynamic programming to balance execution time across stages in homogeneous clusters. To optimize stage placement in heterogeneous clusters, Whale [15] allocates GPUs with a large memory capacity to front stages to accommodate their high memory requirements. HPH [14] allocates GPUs with a small computing capability to front stages to minimize communication time. They overlook the diverse computing and memory demands across different stages. A more recent work Metis [16] leverages the exhaustive search method to generate stage placement plans in heterogeneous clusters. In contrast, *Espresso* proposes a lightweight *resource-aware* stage placement strategy that leverages the *CMR* to jointly characterize the impact of GPU capability and memory capacity on large model training.

Large model training techniques. There have recently emerged many large model training parallelization techniques, such as optimizing GPU memory [39], improving the communication efficiency [18], [40], minimizing bubble time [41], [42], reducing activation values storage [20], [43], integrating parallelism dimensions and automating training strategies [38], [44], [45]. *Espresso* can work with these works above to further reduce the job training time and guarantee the training performance with cost-efficient heterogeneous GPU resources.

VII. Conclusion

This paper introduces *Espresso*, a cost-efficient GPU provisioning framework to facilitate large model training with heterogeneous GPUs in the cloud. *Espresso* proposes a *cost tree-based* provisioning strategy to prioritize searching GPU allocation plans with lower costs and reduce unnecessary branches by multi-dimensional pruning methods. To greedily overlap computation with communication across stages, *Espresso* further devises a resource-aware stage placement strategy that leverages the *compute-memory ratio* to adequately partition model layers into stages and place them to heterogeneous GPUs. Prototype experiments demonstrate that *Espresso* can guarantee the training performance while saving the budget by up to 49.8% compared to existing solutions.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan *et al.*, "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[2] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie *et al.*, "Swin transformer v2: Scaling up capacity and resolution," in *Proc. of IEEE CVPR*, 2022, pp. 12 009–12 019.

[3] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury, "Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates," in *Proc. of ACM SOSP*, Oct. 2023, pp. 382–395.

[4] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou *et al.*, "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning," in *Proc. of USENIX OSDI*, Nov. 2020, pp. 533–548.

[5] R. Shang, F. Xu, Z. Bai, L. Chen, Z. Zhou, and F. Liu, "SpotDNN: Provisioning Spot Instances for Predictable Distributed DNN Training in the Cloud," in *Proc. of IEEE IWQoS*, Jun. 2023, pp. 1–10.

[6] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang *et al.*, "iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2022.

[7] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee *et al.*, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," in *Proc. of USENIX OSDI*, Nov. 2020, pp. 481–498.

[8] Z. Mo, H. Xu, and C. Xu, "Heet: Accelerating Elastic Training in Heterogeneous Deep Learning Clusters," in *Proc. of ACM ASPLOS*, Apr. 2024, pp. 499–513.

[9] A. Sultana, F. Xu, X. Yuan, L. Chen, and N.-F. Tzeng, "Hadar: Heterogeneity-Aware Optimization-Based Online Scheduling for Deep Learning Cluster," in *Proc. of IEEE IPDPS*, 2024, pp. 681–691.

[10] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning," in *Proc. of ACM EuroSys*, Apr. 2020, pp. 1–16.

[11] Z. Mo, H. Xu, and W. C. Lau, "Optimal Resource Efficiency with Fairness in Heterogeneous GPU Clusters," *arXiv preprint arXiv:2403.18545*, 2024.

[12] Q. Wang, F. Wang, and X. Zheng, "Hops: Fine-grained Heterogeneous Sensing, Efficient and Fair Deep Learning Cluster Scheduling System," in *Proc. of ACM SoCC*, Nov. 2024, pp. 1–17.

[13] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λDNN: Achieving Predictable Distributed DNN Training With Serverless Architectures," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.

[14] Y. Duan, Z. Lai, S. Li, W. Liu, K. Ge *et al.*, "HPH: Hybrid Parallelism on Heterogeneous Clusters for Accelerating Large-scale DNNs Training," in *Proc. of IEEE CLUSTER*, Oct. 2022, pp. 313–323.

[15] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi *et al.*, "Whale: Efficient Giant Model Training over Heterogeneous GPUs," in *Proc. of USENIX ATC*, Jul. 2022, pp. 673–688.

[16] T. Um, B. Oh, M. Kang, W.-Y. Lee, G. Kim *et al.*, "Metis: Fast Automatic Distributed Training on Heterogeneous GPUs," in *Proc. of USENIX ATC*, Jul. 2024, pp. 563–578.

[17] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis *et al.*, "PyTorch Distributed: Experiences on Accelerating Data Parallel Training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.

[18] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary *et al.*, "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM," in *Proc. of IEEE SC*, Nov. 2021, pp. 1–15.

[19] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in Neural Information Processing Systems*, vol. 32, pp. 1–10, 2019.

[20] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur *et al.*, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *Proc. of ACM SOSP*, Nov. 2019, pp. 1–15.

[21] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv preprint arXiv:2307.09288*, 2023.

[22] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin *et al.* (2023) Stanford Alpaca: An Instruction-Following LLaMA Model. https://github.com/tatsu-lab/stanford_alpaca.

[23] L. Luo, P. West, P. Patel, A. Krishnamurthy, and L. Ceze, "SRIFTY: Swift and Thrifty Distributed Neural Network Training on the Cloud," in *Proc. of MLSys*, Aug. 2022, pp. 833–847.

[24] R. M. Karp, *Reducibility among Combinatorial Problems*. Springer, 2010.

[25] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proc. of MLSys*, vol. 5, pp. 341–353, 2023.

[26] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, and otheres, "Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent," in *Proc. of USENIX ATC*, Jul. 2023, pp. 995–1008.

[27] W. Liu, Z. Lai, S. Li, Y. Duan *et al.*, "AutoPipe: A Fast Pipeline Parallelism Approach with Balanced Partitioning and Micro-batch Slicing," in *Proc. of IEEE CLUSTER*, Oct. 2022, pp. 301–312.

[28] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters," in *Proc. of ACM KDD*, Aug. 2020, pp. 3505–3506.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer *et al.*, "Pytorch: An Imperative Style, High-performance Deep Learning Library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[30] C. Li. (2023) LLM-Analysis: Latency and Memory Analysis of Transformer Models for Training and Inference. https://github.com/cli99/llm-analysis.

[31] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI blog*, vol. 1, no. 8, pp. 1–24, 2019.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones *et al.*, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, pp. 1–11, 2017.

[33] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai *et al.*, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in *Proc. of ICLR*, Oct. 2020.

[34] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, and G. R. Ganger, "Sia: Heterogeneity-Aware, Goodput-Optimized ML-Cluster Scheduling," in *Proc. of ACM SOSP*, Oct. 2023, pp. 642–657.

[35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. of IEEE CVPR*, Aug. 2009, pp. 248–255.

[36] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho *et al.*, "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning," in *Proc. of USENIX OSDI*, Jul. 2021, pp. 1–18.

[37] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," in *Proc. of ACM EuroSys*, Apr. 2018, pp. 1–14.

[38] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie *et al.*, "Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 470–479, 2022.

[39] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," in *Proc. of IEEE SC*, Nov. 2020, pp. 1–16.

[40] S. Zhang, L. Diao, C. Wu, Z. Cao *et al.*, "HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis," in *Proc. of ACM EuroSys*, Apr. 2024, pp. 524–541.

[41] Z. Liu, S. Cheng, H. Zhou, and Y. You, "Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency," in *Proc. of IEEE SC*, Nov. 2023, pp. 1–13.

[42] S. Li and T. Hoefler, "Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines," in *Proc. of IEEE SC*, Nov. 2021, pp. 1–14.

[43] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng *et al.*, "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models," in *Proc. of ACM PPoPP*, Feb. 2021, pp. 431–445.

[44] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen *et al.*, "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning," in *Proc. of USENIX OSDI*, Jul. 2022, pp. 559–578.

[45] X. Zhang, H. Zhao, W. Xiao, X. Jia, F. Xu *et al.*, "Rubick: Exploiting job reconfigurability for deep learning cluster scheduling," *arXiv preprint arXiv:2408.08586*, 2024.