Pipette: Automatic Fine-grained Large Language Model Training Configurator for Real-World Clusters

Jinkyu Yim[†,1], Jaeyong Song[†,1], Yerim Choi[§], Jaebeen Lee[§], Jaewon Jung[†], Hongsun Jang[†] and Jinho Lee[†,*]
[†]*Department of Electrical and Computer Engineering, Seoul National University*
[§]*Samsung Electronics*
{skyson00, jaeyong.song}@snu.ac.kr, {yr9.choi, jbeen.lee}@samsung.com, {hongsun.jang, jungjaewon, leejinho}@snu.ac.kr

***Abstract*—Training large language models (LLMs) is known to be challenging because of the huge computational and memory capacity requirements. To address these issues, it is common to use a cluster of GPUs with 3D parallelism, which splits a model along the data batch, pipeline stage, and intra-layer tensor dimensions. However, the use of 3D parallelism produces the additional challenge of finding the optimal number of ways on each dimension and mapping the split models onto the GPUs. Several previous studies have attempted to automatically find the optimal configuration, but many of these lacked several important aspects. For instance, the heterogeneous nature of the interconnect speeds is often ignored. While the peak bandwidths for the interconnects are usually made equal, the actual attained bandwidth varies per link in real-world clusters. Combined with the critical path modeling that does not properly consider the communication, they easily fall into sub-optimal configurations. In addition, they often fail to consider the memory requirement per GPU, often recommending solutions that could not be executed. To address these challenges, we propose *Pipette*, which is an automatic fine-grained LLM training configurator for real-world clusters. By devising better performance models along with the memory estimator and fine-grained individual GPU assignment, Pipette achieves faster configurations that satisfy the memory constraints. We evaluated Pipette on large clusters to show that it provides a significant speedup over the prior art.**

***Index Terms*—Large Language Model Training, 3D Parallelism, Automatic Configuration, Distributed Training**

I. Introduction

It is evident that both the industry and academia are experiencing a remarkable surge in large language models (LLMs) [1]–[3]. As the model size grows, the performance (i.e., accuracy) of an LLM is known to continuously improve for various tasks. To provide its huge computational power and memory capacity, researchers often use 3D parallelism, which divides a model along three dimensions (tensor-wise, layer-wise, and batch-wise) as depicted in Figure 1. This has been demonstrated to be efficient at utilizing a massive number of GPUs [3]–[5]. However, the exact configuration of the GPUs toward the optimal performance still remains to be solved.

Heuristic approaches [4], [6] rely on manual rules for the configuration, derived from insights gained by domain experts through numerous trials. To reduce the effort to manually find an adequate configuration, a recent trend [7], [8] has been to automatically search for near-optimal configurations. Most of them profile the computation time and then integrate it into their pipeline latency models to get latencies of configurations.

[1]Co-first authors. [*]Corresponding author.

Unfortunately, we diagnose that these methods tend to have three main limitations that restrict their practical use in the field.

1) *Static Interconnect Assumption.* The existing methods simply assume that the interconnects between the servers are static, with a fixed bandwidth and latency. However, the actual communication latency in a real cluster exhibits heterogeneity among the links [9]–[11], which could yield unexpected straggler effects.
2) *Hidden Critical Path.* The existing methods construct latency models on the 3D parallelism but miss some critical paths. This comes from the discrepancy between the latency model and modern scheduling. While the state-of-the-art latency models [8], [12] assume outdated scheduling methods to achieve maximal throughput, the de facto standard is to use memory-efficient scheduling [5], [13] to relieve the memory capacity requirements.
3) *Out-of-Memory Configurations.* The configurations recommended by the automated tools often require more memory per GPU than what is physically available. This is because those methods do not consider the memory usage of LLM [8] at all or fail to estimate it [12].

To address the above limitations, we propose *Pipette*, an automatic fine-grained 3D parallelism configurator for real-world clusters. We devise the following three novel schemes. First, to fully take the heterogeneous interconnect latency into account, Pipette performs *fine-grained worker dedication* to determine the location of individual workers in the cluster. Second, Pipette uses an elaborately designed *latency estimator* based on the refined latency model, which addresses the aforementioned hidden critical path problem of prior works. For the communication terms in the latency model, latency estimator profiles the actual latency of each interconnect in a real-world cluster. Last, to only produce configurations that meet the per-GPU memory constraint, we introduce accurate *memory estimator*, based on a simple ML model. We evaluated Pipette on clusters with up to 128 NVIDIA V100 or A100 GPUs connected via NVLink and Infiniband, and it provided significant up to 1.46× speedup over the prior art AMP [8].

II. Background and Related Work

A. 3D Parallelism and Pipeline Schedule

3D parallelism [4], [5], [14] is a common parallelization technique used to train LLMs with tens to thousands of GPUs in a cluster. It splits a model into several parts and distributes them

979-8-3503-4859-0/DATE24/© 2024 EDAA

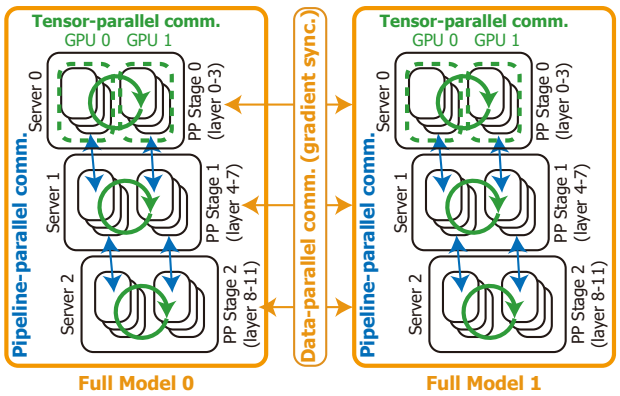

Fig. 1. Example 3D parallelism configuration with 12 GPUs.

to each worker (i.e., GPU) using *pipeline (PP)*, *tensor (TP)*, and *data (DP) parallelism*. Figure 1 depicts an example 3D parallelism configuration with a 3-way pipeline, 2-way tensor, and 2-way data parallelism. Because the entire model comprises tens to hundreds of layers, pipeline parallelism divides the layers into several stages. The peer-to-peer pipeline parallel communications (blue arrows) fulfill the dependency among the pipeline stages. Tensor parallelism further splits the layers at an intra-layer level. It utilizes all-reduce communication (green arrows) inside each layer. Because this all-reduce communication introduces a severe overhead, tensor parallel ways are usually placed within a server. Data parallelism, a widely used technique to parallelize deep learning models, can be orthogonally applied on top of the above two parallelization methods. It requires parameter gradient synchronization (orange arrows), which again uses all-reduce communication.

Figure 2 illustrates one training iteration of the example configuration in Figure 1 with six microbatches. Usually, the key is to hide the communication latency by overlapping it with the computation and reducing the idle resource (i.e., bubbles) by using microbatches. One overhead from such a schedule is the high memory capacity requirement, because each GPU has to store activations and gradients of all six microbatches for backward passes. Therefore, recent studies [3]–[5], [15] on LLM training adopted the memory-efficient schedule (i.e., 1F1B) to reduce the memory usage. As shown in Figure 2b, interleaving the forward and backward one by one dramatically reduces the memory capacity requirements because the microbatch data can be dropped from the memory after performing backpropagation.

B. Automatic Configuration and Latency Model

Some researchers [7], [8] have attempted to automatically tune the training configuration. Typically, they have established a first-order pipeline latency model and performed an exhaustive search. Let C be the computational latency to process one microbatch, while $T_{com}^{(\cdot)}$ is the communication latency of each parallelization, n_mb is the number of microbatches, and pp is the number of pipeline stages. Then, the latency model from [8] is as follows:

$$T_{prev} = (n\_mb - 1) \cdot (C + T_{com}^{TP}) + pp \cdot (C + T_{com}^{TP}) + (pp - 1) \cdot T_{com}^{PP} + T_{com}^{DP} \quad (1)$$

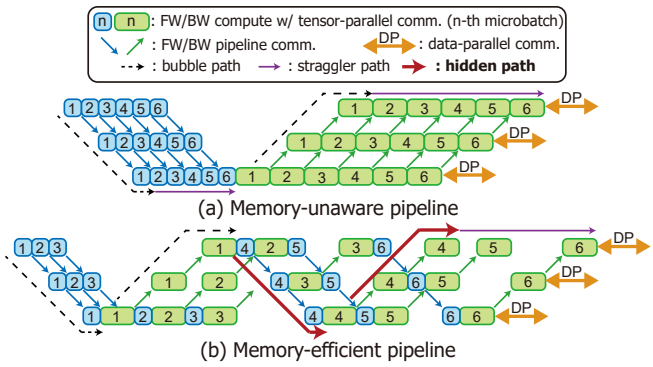

Fig. 2. Pipeline scheduling.

The first term represents the runtime of stragglers in the pipeline (purple arrows). The second and third terms capture the pipeline bubble computation and communication time (black dotted arrows), respectively. The last term adds the data parallel communication time (orange arrow). Usually, profiled values are used for computation C, and document-specified bandwidth values are used for T_{com}. However, this does not fit well with the modern memory-efficient schedules of Figure 2b, and does not consider the heterogeneous nature of real-world clusters.

III. Pipette Overview

Algorithm 1 Pipette Procedure

Input:
G:#GPUs, bs_{global}: global batch size, M_{limit}: memory limit per GPU
Output:
$Conf$: parallelization configuration,
Map: $Conf \rightarrow$ GPU mapping, T: execution latency

1: $BW \leftarrow network\_profile()$ // Profile actual bandwidth matrix
2: $Conf_{best} \leftarrow None$, $Map_{best} \leftarrow None$, $T_{best} \leftarrow \infty$
3: **for** $Conf \in \{(pp, tp, dp) \mid pp \cdot tp \cdot dp = G, \quad pp, tp, dp \in \mathbb{N}\}$ **do**
4: $bs_{mini} = bs_{global}/dp$ // Minibatch size
5: **for** $bs_{micro} \in divisors(bs_{mini})$ **do**
6: // Exclude OOM configurations (Section VI)
7: **if** $MemEstimator(Conf, bs_{micro}) > M_{limit}$ **then** continue
8: // Fine-grained Worker Dedication (Section IV)
9: **while** $Map \leftarrow SA\_NextMap(Map)$ **do**
10: // Estimate Latency (Section V)
11: $T \leftarrow LatencyEstimator(Conf, Map, bs_{mini}, bs_{micro}, BW)$
12: **if** $T < T_{best}$ **then**
13: $Conf_{best}, Map_{best}, T_{best} \leftarrow Conf, Map, T$
14: **end if**
15: **end while**
16: **end for**
17: **end for**
18: **return** $Conf_{best}, Map_{best}, T_{best}$

Algorithm 1 shows the overall Pipette procedure. Similar to previous methods [7], [8], Pipette finds the best configuration by examining the possible pipeline/tensor/data parallel combinations. However, instead of relying on the document-specified link bandwidth, Pipette considers the heterogeneous bandwidths of a real-world cluster by profiling them (line 1). At the inner for loop, for each selected configuration ($Conf$) with each microbatch size (bs_{micro}), Pipette checks the memory capacity constraint using *Memory Estimator* (line 7, Section VI). If it is runnable, Pipette finds the best configuration for GPU mapping (Map) by *Fine-grained Worker Dedication* with the simulated

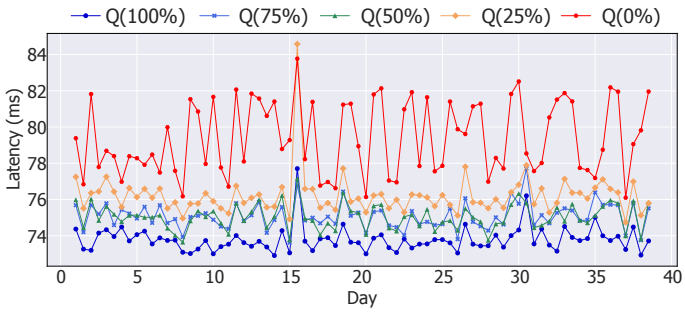

Fig. 3. Inter-stage communication latency in a real-world cluster for 40 days.

annealing (SA) algorithm (lines 9-15, Section IV). SA keeps iterating the while loop for a specific time limit (e.g., 10s). For each iteration, *Latency Estimator* of Pipette estimates the latency of the configuration of the current mapping (line 11, Section V). If the latency is the smallest one ever found, we store the current configuration/mapping/latency as the best case (line 13). As a result, Pipette returns the best configuration, mapping, and latency (line 18).

IV. Fine-grained Worker Dedication

- **Key observation**: Real-world clusters have heterogeneous link bandwidths.

In Figure 3, we provide a 40-day continuous profiling result of a commercial industry cluster ('high-end' environment from §VII-A) to measure the actual peer-to-peer bandwidth of interconnect using mpiGraph [16]. Each line shows the latency corresponding to the order combination of 8 nodes. We made sure that only the measurements were running to ensure that the measurements were isolated and reliable. From the plot, it is clear that the pairs of nodes exhibit different latency even though they are designed to be equal. Such phenomenon has been reported from multiple commercial clouds [9]–[11], which indicates the practical need for considering the differences.

To embrace such heterogeneity in the parallel configuration, we advocate for dedicating individual logical workers to the physical GPUs in a fine-grained manner. There are several rationales to this. First, the communications between the pipeline stages directly affect the execution time, but they take place only on a subset of the cluster network. By steering the traffic to travel through higher-speed links, the execution time can be reduced. Furthermore, certain DP (i.e., all-reduce) traffic does not affect the execution time. As shown in Figure 4, only the DP communication of the earlier stages (a $\leftrightarrow$ d in (a) and f $\leftrightarrow$ c in (b)) are on the critical path.

In Figure 4, we provide a toy example of a six-node cluster based on the pipeline schedule example in Figure 2b of $pp = 3$ and $dp = 2$. The bandwidth variance between the six servers is slightly exaggerated to have $2\times$ difference between the slow links (thin lines) and fast links (bold lines). As illustrated in Figure 4a, a naive configuration (alphabetical ordering) could result in a longer pipeline schedule. The inter-stage communications between nodes (a $\rightarrow$ b, b $\rightarrow$ c, and d $\rightarrow$ e) are slower than other inter-stage communications. The data parallel communication in the first stage (a $\leftrightarrow$ d) is much slower than in the last stage (c $\leftrightarrow$ f) due to the heterogeneity.

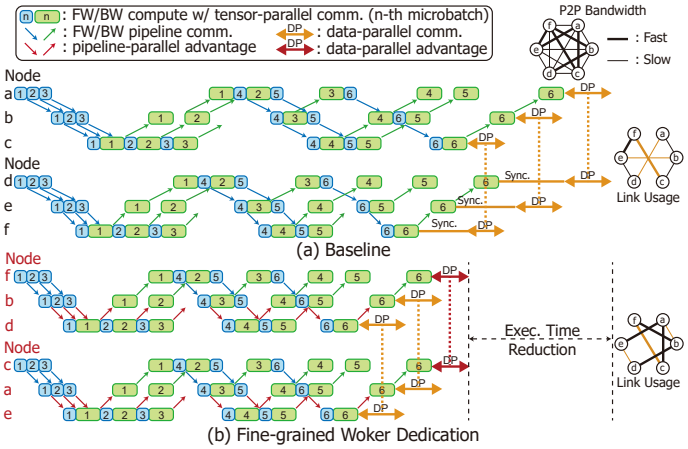

Fig. 4. Baseline and fine-grained worker dedication schedule.

On the other hand, Figure 4b shows the pipeline from the fine-grained worker dedication. We reordered/regrouped the nodes in the pipeline and changed the data parallel grouping. For example, the pipeline group of node (a, b, c) is changed into node (f, b, d). This minimizes the inter-stage communication and reduces the data parallel communication on the critical path as depicted with red arrows.

This problem essentially becomes finding a 1:1 mapping f between logical workers and the GPUs. Given a parallel configuration pp, tp, and dp,

$$f : W \rightarrow G, \quad W = \mathbb{N}^{[1,pp]} \times \mathbb{N}^{[1,tp]} \times \mathbb{N}^{[1,dp]} \\ \forall w_i \neq w_j,\ f(w_i) \neq f(w_j), \tag{2}$$

where W is the set of logical workers and G is the set of GPUs, where $|W| = |G| = pp \times tp \times dp$. This problem is analogous to the classic multi-core mapping [17], [18], and thus we use the popular simulated annealing (SA) to find f, similar to [18]. We used three movements for the SA. Regarding f as a string, the first two movements are *migration* (remove a single element to a random position), *swap* (exchange two elements), and *reverse* (take a substring and reverse its order). Notably, the reverse move is based on our observation that the bidirectional bandwidths between a pair of nodes are often almost symmetric. The objective is to minimize the execution latency (Section V). We used 10 seconds for the SA time limit, and set $\alpha = .999$ for the temperature reduction coefficient. The overall time of the simulated annealing takes a few minutes.

V. Latency Estimator

- **Key observation**: Modern memory-efficient pipeline schedule contains hidden critical paths.

A latency model is a necessary component to evaluate the mapping functions recommended by fine-grained worker dedication through SA. Actually running the model in the target cluster consumes too much time and is not practical, especially in a shared cloud with long waiting queues.

Existing approaches [8], [12] model the pipeline latency as Equation (1), whose critical path comprises the bubble path and the straggler path. However, we identify an additional hidden critical path that appears on the modern memory-efficient

schedule, as depicted with red arrows in Figure 2b. As the key to the memory-efficient schedule is to interleave the forward and backward blocks (i.e., 1F1B), such hidden paths occur $(n\_mb/pp - 1)$ times. To reflect this, we divide the memory-efficient pipeline into bubbles (T_{bubble}), a straggler ($T_{straggler}$), and data parallel communication time (T_{com}^{DP}). With those, the total pipeline latency model of Pipette ($T_{Pipette}$) is as follows:

$$T_{Pipette} = T_{bubble} \cdot (n\_mb/pp) + T_{straggler} + T_{com}^{DP}, \tag{3}$$

where T_{bubble} and $T_{straggler}$ can be calculated by following:

$$\begin{aligned} T_{bubble} &= pp \cdot (C + T_{com}^{TP}) + (pp-1) \cdot T_{com}^{PP}, \\ T_{straggler} &= (pp-1) \cdot (C + T_{com}^{TP}). \end{aligned} \tag{4}$$

For the individual terms, we model them as follows: First, for the intra-node microbatch computation (C) and tensor parallel communication (T_{com}^{TP}), we use the profiled values as in previous works [8], [12]. Optionally, we provide an extrapolated latency estimation model for other cluster sizes that have not been profiled, similar to our memory estimator (Section VI).

Second, for the pipeline parallel communication latency (T_{com}^{PP}), we define $B(g_1, g_2)$ as the pairwise bandwidth between g_1 and g_2 in G. We set the pipeline parallel message size as msg_{PP}. With a slight abuse of notation, we model T_{com}^{PP} as:

$$T_{com}^{PP} = \max_{y,z} \sum_{x=1}^{pp-1} \frac{2 \cdot msg_{PP}}{B(G_{f(x,y,z)}, G_{f(x+1,y,z)})}. \tag{5}$$

The msg_{PP} is doubled to account for the forward and backward passes. x, y, and z represent the ids of the pipeline-, tensor-, and data-parallel groups, respectively. Thus, the denominator represents the bandwidth between adjacent pipeline stages, and T_{com}^{PP} takes the slowest of the end-to-end pipelines.

Third, for the data parallel communication term (T_{com}^{DP}), we assume the hierarchical-ring all-reduce algorithm, which contains two intra-node all-reduces and a single inter-node all-reduces. We set the group of intra-node communicators (GPUs) on pipeline stage x, tensor group y as $W_{x,y}^{intra}$ and inter-node communicators (nodes) as $W_{x,y}^{inter}$. Following the well-known all-reduce communication latency equation from [19], the data parallel communication latency becomes:

$$\begin{aligned} T_{com}^{DP} = &\max_{y} \frac{4 \cdot (|W_{1,y}^{intra}| - 1) \cdot msg_{DP}}{(|W_{1,y}^{intra}|) \cdot \min\limits_{w_1, w_2 \in W_{1,y}^{intra}} (B(G_{f(w_1)}, G_{f(w_2)}))} \\ &+ \max_{y} \frac{2 \cdot (|W_{1,y}^{inter}| - 1) \cdot msg_{DP}}{(|W_{1,y}^{inter}|) \cdot \min\limits_{w_1, w_2 \in W_{1,y}^{inter}} (B(G_{f(w_1)}, G_{f(w_2)}))}. \end{aligned} \tag{6}$$

Only the DP communication of stage 1 on the critical path is considered, and the all-reduce delay depends on the slowest link in participation. To verify the effect of the new latency model and bandwidth profiling, we compared the estimation results of latency estimator with [8] in Figure 5a. [8] shows 23.18% mean absolute percentage error (MAPE), whereas Pipette shows 5.87%. [8] fails to estimate latencies accurately, because it does not consider the memory-efficient schedule and uses the ideal bandwidth in the communication-related terms.

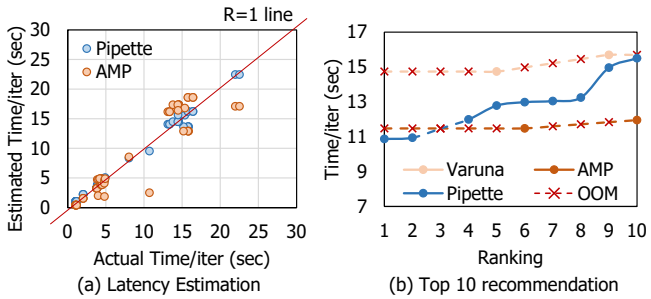

Fig. 5. Latency estimation and top 10 recommendation of baselines and Pipette.

VI. Memory Estimator

- **Key observation**: Prior art often recommends OOM solutions, hindering them from being practical.

When the existing approaches recommend configurations, they fail to consider the memory requirements. To demonstrate this, we plot the top 10 configurations from AMP [8] and Varuna [12] in Figure 5b. The experiment was conducted on the mid-range cluster in Section VII-A. Eight of the top 10 suggested configurations incur OOM in both methods, including the top recommendations. This significantly harms the practicality because users need to try the solutions one by one until they find a feasible configuration. Therefore, when searching for configurations, we integrate an accurate memory estimator. A common way to estimate the memory requirement is by dividing the model size by the number of stages and tensor-parallel ways and then approximating the activation size by considering the layer structures [20].

However, with pipelining, such a simple model significantly underestimates the memory usage, especially the auxiliary structures of the training framework and external libraries [21]. Taking advantage of the fact that LLMs are composed of the same transformer blocks, we model the memory requirement as a function of several system and model parameters. When we approximate them with a multi-layer perceptron (MLP):

$$\begin{aligned} M_{max} = MLP(&n\_gpus, n\_layers, n\_hiddens, n\_heads, \\ &tp, pp, dp, bs_{micro}, bs_{mini}, bs_{global}). \end{aligned} \tag{7}$$

The inputs include the configuration parameters (Algorithm 1) in addition to the number of GPUs (n_gpus), layers (n_layers), hidden dimensions (n_hidden), and attention heads (n_heads). To train the model, we use the profiled data from all possible configurations using up to four cluster nodes (32 GPUs), and validate the model extrapolation up to 128 GPUs. The MLP model has five layers with 200 hidden sizes and is trained for 50,000 iterations with the profiled data. This training process is required for each cluster only once because the trained model can be used afterward for all other configurations. When memory estimator decides whether a configuration is runnable, it sets a soft margin to stably recommend runnable configurations. As a result, Pipette recommends more runnable results than baselines, as shown in Figure 5b.

TABLE I
EXPERIMENTAL ENVIRONMENT

Mid-range Cluster (16 Nodes)	GPU	8× NVIDIA V100
	CPU	2× Xeon Gold 6142, 16 cores
	Memory	768GB DDR4 ECC
	Inter-node	Infiniband EDR (100Gbps)
	Intra-node	NVLink (300GBps)
High-end Cluster (16 Nodes)	GPU	8× NVIDIA A100
	CPU	2× EPYC 7543, 32 cores
	Memory	1TB DDR4 ECC
	Inter-node	Infiniband HDR (200Gbps)
	Intra-node	NVLink (600GBps)

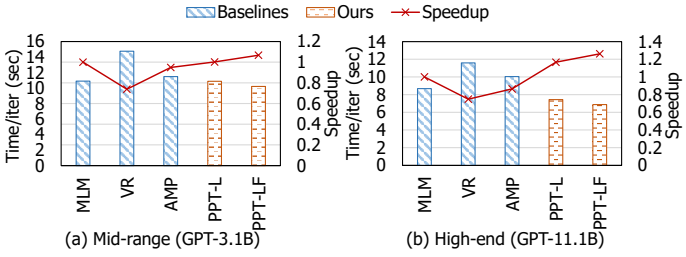

Fig. 6. Training time and speedup of Pipette and the baselines.

VII. EVALUATION

A. Environments

Clusters. Table I shows the experimental environment we used for evaluation. We used two clusters: The V100 ('Mid-range') cluster has an inter-node connection with Infiniband EDR (100Gbps) and an intra-node connection with NVLink (300GBps). The A100 ('High-end') cluster has an inter-node connection with Infiniband HDR (200Gbps) and an intra-node connection with NVSwitch (600GBps). We set a cluster with 128 GPUs (16 nodes) as default in both cases.

Framework and models. We used the publicly available source code of Megatron-LM [6] as our baseline LLM training framework. For interconnect profiling, we ran NCCL-tests [22]. We tested GPT models of sizes up to 3.1B and 11.1B parameters, which reach the GPU memory limit in the mid-range and high-end clusters, respectively. We evaluated the total minibatch size from 128 to 512 and the microbatch size from 1 to 8 for experiments and adopted other hyperparameters from [14].

Baselines. We chose manually tuned method Megatron-LM (MLM) [14], Varuna (VR) [12] and AMP [8] as baselines. Megatron-LM generally tunes the number of GPUs per node as a tensor parallel way ($tp = 8$). Therefore, if not stated, we fixed $tp = 8$ in the MLM baseline. Varuna emphasizes using the pipeline parallel-only configuration for LLM training and provides successful speedup over [5]. We use the default configuration suggested by its open-sourced code. AMP is the state-of-the-art automatic configurator for 3D parallelism, so we tested it as a baseline. Because AMP often recommends out-of-memory cases, we manually tested them one by one from the top recommendation until we reached a runnable configuration.

B. Speedup and Ablation

Figure 6 shows the training time and speedup of Pipette compared to the baselines. MLM provides better training throughput than other baselines because it is tuned for the memory-efficient pipeline, while other baselines recommend configurations based on the memory-unaware one. Therefore, we normalized the speedup to MLM. We also give ablation in Figure 6 to illustrate the effect of each scheme. 'PPT-L' means Pipette only with the elaborately designed latency estimator with memory estimator applied. 'PPT-LF' denotes Pipette with both latency estimator and fine-grained worker dedication.

In mid-range and high-end clusters, PPT-L provides 1.36×, 1.56× speedup over Varuna (VR). VR uses a configuration without tensor parallelism. However, in both our configurations, VR induces too much network overhead, especially due to the existence of the hidden critical path (Section V). AMP is much faster than VR, but PPT-L still shows 1.06× and 1.35× speedup over it. This comes from the ability of Pipette to estimate the execution latency correctly. Pipette with fine-grained worker dedication (PPT-LF) further exploits the heterogeneity of a real-world cluster, showing 1.12×, 1.46× speedup over AMP in mid-range and high-end clusters, respectively. As a result, Pipette provides 1.07× and 1.26× speedup over the manually tuned baseline (MLM), which requires manual trials to find a fast configuration. Additionally, Pipette tends to show further speedup on the high-end cluster, which handles larger models.

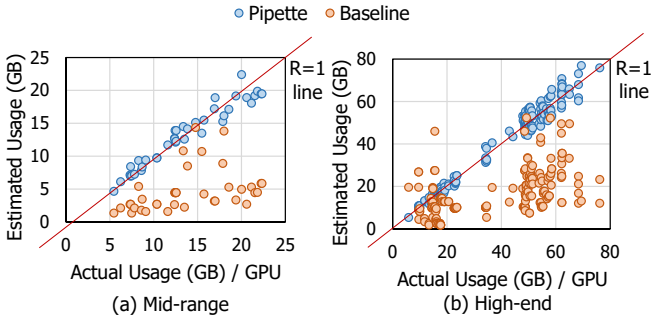

Fig. 7. Memory estimation results of Pipette and the baseline.

C. Analysis of Memory Estimator

When searching for the best configuration, Pipette checks whether the configuration does not exceed the GPU memory limit. To check the practicality of memory estimator, we compare it with [20]. Figure 7 plots how well Pipette and the baseline estimate the actual maximum memory usage. We collected 215 data points, representing the estimated and actual GPU memory usage with various model and parallel configurations. The baseline underestimates the maximum memory usage because it does not consider the usage from the training framework and external libraries [21]. Therefore, the baseline provides the mean absolute percentage error (MAPE) of 65.71% and 59.49%, while Memory estimator of Pipette only shows 7.39% and 6.42% in mid-range and high-end clusters, respectively. This demonstrates the importance of a dedicated memory estimator for an automatic configurator.

D. Analysis of Configuration Overhead

Compared to other works, Pipette introduces bandwidth profiling, simulated annealing-based worker mapping, and memory estimator. Therefore, analyzing the overhead of Pipette is meaningful, while it is negligible compared to LLM training time. In

TABLE II
CONFIGURATION OVERHEAD OF PIPETTE

Cluster	Mid-range		High-end	
#Nodes (Model)	8 (1.1B)	16 (3.1B)	8 (8.1B)	16 (11.1B)
Bandwidth Profiling	58.13 sec.	119.62 sec.	113.67 sec.	239.21 sec.
Simulated Annealing	640.38 sec.	790.51 sec.	640.23 sec.	769.91 sec.
Memory Estimation	0.03 sec.	0.04 sec.	0.03 sec.	0.05 sec.
Total Conf. Time	10.71 min.	13.23 min.	10.71 min.	16.85 min.
Overhead	0.03%	0.03%	0.02%	0.05%
AMP (300K)	30.10 days	37.74 days	36.86 days	34.85 days
Pipette (300K)	29.13 days	35.42 days	31.61 days	23.89 days
Time Saving	0.97 days	2.33 days	5.25 days	10.97 days

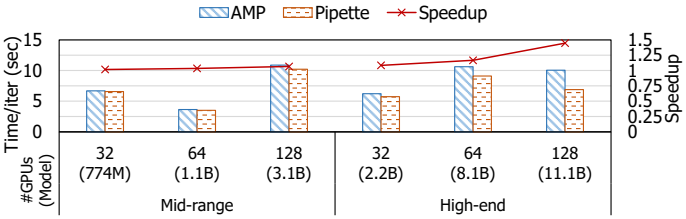

Fig. 8. Cluster and model size scalability of Pipette.

Table II, we demonstrate the configuration overhead of Pipette. The simulated annealing takes most of the total overhead. The total number of nodes is the main factor in determining the problem size of the simulated annealing algorithm. Therefore, the total time is similar in both mid-range and high-end clusters when the number of nodes is the same. The overhead means the portion of configuration time when training the full iteration (300K) following [14]. The overhead is less than 0.05% of the total training time, and the benefit is 0.97-10.97 days over the configuration recommended by AMP.

E. Sensitivity Studies

Cluster/Model size. While we use a cluster with 128GPUs as the default, it is essential to check Pipette still provides speedup with different numbers of GPUs (from 32 to 128). Figure 8 shows the training time and speedup of Pipette over AMP with various numbers of GPUs. We weak-scaled the model size with the number of GPUs, following [4], [14]. In clusters with smaller numbers of GPUs than the default setting, the heterogeneity appears less, so Pipette shows a smaller speedup but still brings 1.02-1.17× speedup.

Micro/Minibatch size. Recent works use microbatch sizes from 1 to 8 and minibatch sizes from 64 to 1K, so we checked the micro/minibatch size sensitivity of Pipette, in Figure 9. For the experiment, we ran Pipette and AMP to find configurations when the micro/batch size is fixed. For the microbatch size sensitivity, we fixed minibatch size to 256. In minibatch size sensitivity, we used microbatch size with 8. In all settings, Pipette provides stable 1.14-1.44× speedup over AMP, which shows the practicality of Pipette.

VIII. CONCLUSION

We propose Pipette, an automatic fine-grained LLM training configurator for real-world clusters. Pipette provides significant speedup over the baselines by devising an accurate, critical path-based latency model and dedicating each worker in a fine-grained manner. In addition, to the best of our knowledge, Pipette is only the configurator that recommends configurations meeting the memory capacity constraints. This greatly enhances the practicality of Pipette.

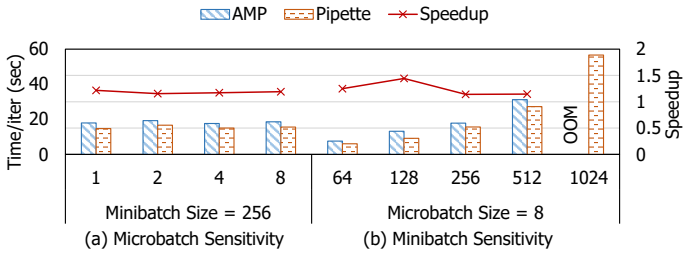

Fig. 9. Microbatch and minibatch size sensitivity of Pipette.

ACKNOWLEDGEMENT

This work was supported by Samsung Electronics Co., Ltd (IO221111-03540-01), and the National Research Foundation of Korea (NRF) grants (2022R1C1C1011307).

REFERENCES

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language Models Are Unsupervised Multitask Learners," *OpenAI blog*, 2019.
[3] T. Brown, B. Mann, N. Ryder, *et al.*, "Language Models are Few-Shot Learners," *NeurIPS*, 2020.
[4] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," *SC*, 2020.
[5] D. Narayanan, M. Shoeybi, J. Casper, *et al.*, "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM," *SC*, 2021.
[6] J. Casper, M. Patwary, B. Fomitchev, *et al.*, *Nvidia/megatron-lm: V2.5*, version v2.5, Aug. 2021. DOI: 10.5281/zenodo.5181820.
[7] J. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional Planner for DNN Parallelization," *NeurIPS*, 2021.
[8] D. Li, H. Wang, E. Xing, and H. Zhang, "AMP: Automatically Finding Model Parallel Strategies with Heterogeneity Awareness," *NeurIPS*, 2022.
[9] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze, "PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the public Cloud," *MLSys*, 2020.
[10] A. Moody, "Contention-free routing for shift-based communication in mpi applications on large-scale infiniband clusters," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2009.
[11] S. S. Vazhkudai, B. R. De Supinski, A. S. Bland, *et al.*, "The design, deployment, and evaluation of the coral pre-exascale systems," *SC*, 2018.
[12] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models," *EuroSys*, 2022.
[13] D. Narayanan, A. Harlap, A. Phanishayee, *et al.*, "PipeDream: Generalized Pipeline Parallelism for DNN Training," *SOSP*, 2019.
[14] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.
[15] J. Song, J. Yim, J. Jung, *et al.*, "Optimus-CC: Efficient large NLP model training with 3D parallelism aware communication compression," *ASPLOS*, 2023.
[16] *LLNL mpiGraph Tests*, https://github.com/LLNL/mpiGraph.
[17] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," *DATE*, 2004.
[18] J. Hu and R. Marculescu, "Energy-and performance-aware mapping for regular NoC architectures," *IEEE TCAD*, 2005.
[19] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *IJHPCA*, pp. 49–66, 2005.
[20] T. Bricken, *Transformer Memory Requirements*, https : / / www . trentonbricken.com/TransformerMemoryRequirements/, 2022.
[21] Y. Gao, Y. Liu, H. Zhang, *et al.*, "Estimating GPU Memory Consumption of Deep Learning Models," *ESEC/FSE*, 2020.
[22] *NVIDIA NCCL Tests*, https://github.com/NVIDIA/nccl-tests.