BALANCING PIPELINE PARALLELISM WITH VOCABULARY PARALLELISM

Man Tsung Yeung [† 1 2] **Penghui Qi** [1 2] **Min Lin** [1] **Xinyi Wan** [1]

ABSTRACT

Pipeline parallelism is widely used to scale the training of transformer-based large language models, various works have been done to improve its throughput and memory footprint. In this paper, we address a frequently overlooked issue: the vocabulary layers can cause imbalanced computation and memory usage across pipeline stages, worsening pipeline bubbles and the memory bottleneck. To tackle this, we partition the vocabulary layers evenly across pipeline devices and group the computation into pipeline passes. To reduce the activation memory overhead, we propose several algorithms to reduce communication barriers within vocabulary layers. Additionally, we utilize a generalizable method to integrate Vocabulary Parallelism with existing pipeline schedules. By combining these techniques, our methods effectively balance the computation and parameter memory, with only a small constant activation memory overhead. Notably, when combined with activation memory-balanced schedules like *V-Half*, our approach achieves perfect balance in both memory and computation. Extensive evaluations demonstrate that our method achieves computation and memory balance regardless of the vocabulary size, resulting in a 5% to 51% improvement in throughput compared to naïve approaches, meanwhile significantly reducing peak memory usage especially for large vocabulary scenarios. Our implementation is open-sourced at https://github.com/sail-sg/VocabularyParallelism.

1 INTRODUCTION

As the scale of transformer models (Vaswani et al., 2017; Brown et al., 2020) continues to grow, model parallelism has garnered significant attention within the deep learning community. Several model parallel techniques have been proposed to address the challenges associated with training large models, including Zero Redundancy Optimizer (ZeRO) (Rajbhandari et al., 2020; Zhao et al., 2023), Tensor Parallelism (TP) (Shoeybi et al., 2019), and Pipeline Parallelism (PP) (Huang et al., 2019; Harlap et al., 2018; Narayanan et al., 2021; Qi et al., 2023; 2024). Each of these methods has its own advantages and limitations. For instance, ZeRO is effective in reducing memory by eliminating redundant parameter storage, but suffers from high communication overhead when gathering partitioned parameters and gradients for scenarios with limited network bandwidth or requiring frequent parameter updates. TP can efficiently handle large model parameters by splitting them across devices, but often faces low arithmetic intensity and requires significant inter-device communication. Among these techniques, PP shows distinct advantages due to its low communication cost and high arithmetic intensity, making it particularly attractive for training large-scale models. However, PP faces two significant challenges: pipeline bubbles and high memory consumption. Pipeline bubbles occur when there are idle periods in the pipeline stages, leading to suboptimal utilization of computational resources. Various strategies have been proposed to mitigate pipeline bubbles, such as token-level PP (Li et al., 2021) and interleaved 1F1B (Narayanan et al., 2021). An exceptional advancement is zero-bubble pipeline parallelism (Qi et al., 2023; 2024), which achieves almost zero bubble in many scenarios through splitting backward pass into activation gradient computation and weight gradient computation. In most PP schedules, the activations of several microbatches are stored to reduce pipeline bubbles, making memory a critical bottleneck to scale large models. Previous work has explored activation recomputation (Chen et al., 2016; Korthikanti et al., 2023), memory transferring (Kim et al., 2023) and memory-efficient V-Shape scheduling (Qi et al., 2024) to mitigate this issue. Despite various effort, the memory bottleneck still poses the largest limitation for PP.

In this paper, we focus on an imbalance issue in PP caused by vocabulary-related layers, which is often overlooked in practice but can significantly degrade the performance in both throughput and memory. Typically, transformer layers are uniformly distributed across pipeline stages, while the first stage contains an additional input layer and the last stage contains an additional output layer. Such imbalanced setup greatly hurts the performance in both computation

[†] Work was done during an internship at Sea AI Lab. [1]Sea AI Lab [2]National University of Singapore. Correspondence to: Xinyi Wan <wanxy@sea.com>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2024. Copyright 2024 by the author(s).

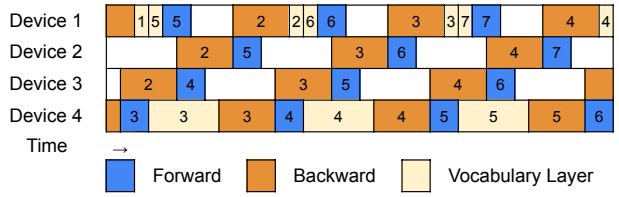

Figure 1. Repeating pattern in an imbalanced pipeline. Bubbles are incurred due to an extra output layer in the last pipeline stage.

and memory. Firstly, pipeline bubbles are introduced in other pipeline stages due to their less workload, as shown in Figure 1. Additionally, the additional input layer in the first stage exacerbates the memory bottleneck for most PP schedules like 1F1B (Harlap et al., 2018). Finally, as the vocabulary size grows larger (Tao et al., 2024), this imbalance becomes more pronounced, as shown in Figure 2. For instance, in the case of Gemma2 9B with a vocabulary size of 256k (Team et al., 2024), both the computation and parameter memory of the output layer are approximately 5 times those of the transformer layers, highlighting the severity of the issue.

To address this imbalance issue, we propose a novel Vocabulary Parallelism approach to balance the computation and memory in PP. By partitioning the vocabulary layers across all pipeline devices, we introduce several methods to group the computation and communication barriers together with a generalizable scheduling approach in PP, with only a small constant memory overhead. Extensive experiments demonstrate our approach significantly outperforms naïve layer redistribution and other existing methods, resulting in up to 51% improvement in throughput.

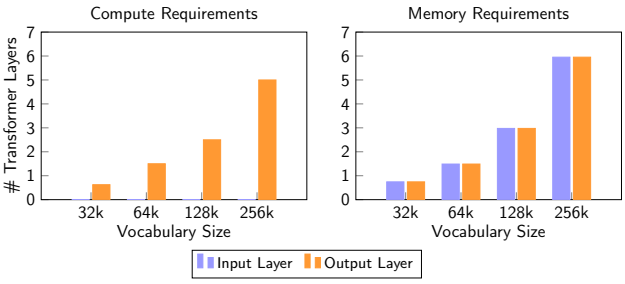

Figure 2. Ratio of compute and memory of vocabulary layers compared to transformer layers in Gemma2-9B.

2 Related Work

Balancing Activation Memory A line of research addresses another aspect of imbalance in PP, the activation memory with the 1F1B schedule. For instance, BPipe (Kim et al., 2023) transfers activations between devices, trading communication for reduced peak memory. Another approach uses V-Shape scheduling to create a pipeline schedule with balanced and efficient memory usage. These methods are orthogonal to our work, and combining them can achieve fully balanced pipeline parallelism in both computation and memory (activations and parameters).

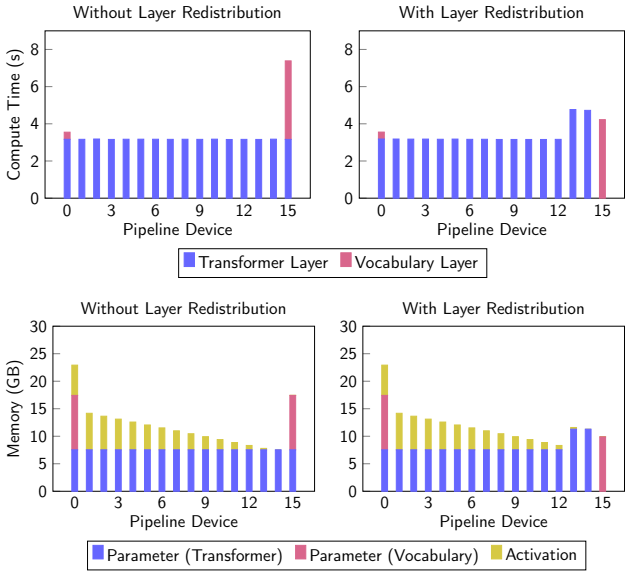

Figure 3. Transformer Layer Redistribution for a 7B GPT-like model with vocabulary size 128k. In this case, each stage has 2 transformer layers, while output layer is equivalent to 2.4x of transformer layer on compute and 2.6x on parameter memory.

Balancing Vocabulary Layers Some existing training systems try to mitigate the imbalance caused by vocabulary layers by redistributing transformer layers across different stages. DeepSpeed (Smith et al., 2022) uses a greedy method to automatically re-balance the workload between stages at the layer level. Similar strategies are employed in the training of Skywork-MoE (Wei et al., 2024). However, simply redistributing transformer layers faces several disadvantages. Firstly, even after redistribution, compute imbalance can still persist since only a subset of pipeline stages receive additional layers. An example is shown in Figure 3. This is particularly evident when the number of layers on each stage is small. Secondly, different layer types have varying compute-to-memory ratios, meaning that the re-balancing can only be based on either compute or memory but not both. In practice, the re-balancing is typically performed based on compute, leaving the memory imbalance still significant, particularly for input vocabulary layers that require minimal compute but substantial memory. Lastly, the effectiveness and planning of redistribution heavily depend on both the model settings and pipeline parallel settings. This makes it less flexible and challenging to adopt in various scenarios.

It's also worth noting that some models pretrained from scratch like Llama 3 (Dubey et al., 2024) reduce one transformer layer each from the first and the last stages, respectively. This method requires changes to architecture of models, which is out of the scope of this paper. Also, it has limitations if the training starts from a checkpoint where the number of transformer layers of model is fixed.

Another method to mitigate the imbalance problem in pipeline parallelism (PP) is the interlaced pipeline, reported by the automatic parallelism framework nnScaler (Lin et al., 2024). This approach distributes the input and output vocabulary layers across different pipeline devices using a tensor parallel (TP) style (Narayanan et al., 2021). By alternating between TP for vocabulary layers and PP for transformer layers, it aims to balance compute and memory overhead. However, TP requires frequent synchronization between devices, leading to two major drawbacks. First, the peak activation memory for 1F1B increases to 1.5 times of its original value (see Appendix B.1), which may make the critical memory bottleneck even worse. Second, the synchronized all-reduce during the output vocabulary layer introduces additional pipeline bubbles for each microbatch. Our ablation study in Appendix B.2 shows these all-reduce along slows down the end to end training by approximately 11% on 32 GPUs. These significant overheads in both activation memory and pipeline bubbles render the interlaced pipeline impractical in real-world scenarios.

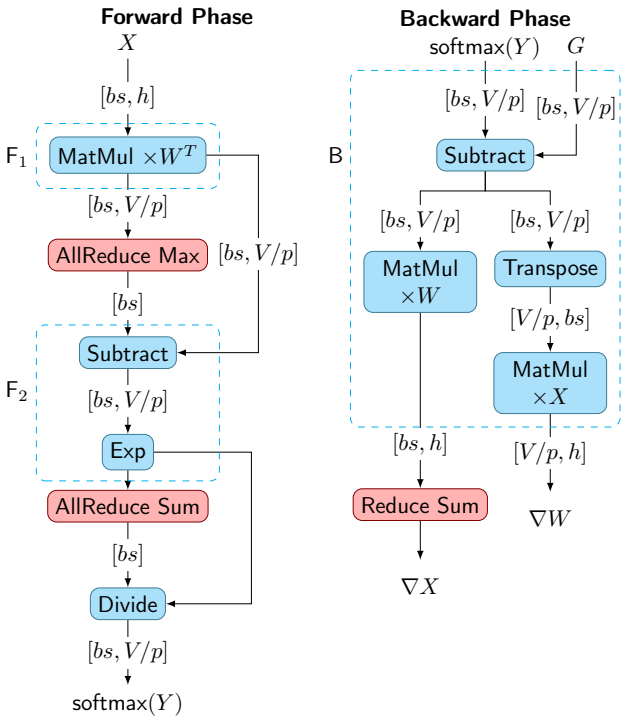

Figure 4. Computation graph of the output layer after partitioning across the vocabulary dimension. There are three all-reduce / reduce communications across all devices.

3 Vocabulary Parallelism in Pipeline Parallelism

To completely address the imbalance issue in PP, we propose Vocabulary Parallelism under the following design principles:

- We partition the vocabulary layers across the vocabulary dimension, and distribute it evenly to all pipeline devices.
- To be native to pipeline parallelism, the computation of vocabulary layers should be represented as passes similar to forward/backward passes of transformer layers.
- Integrating vocabulary passes into pipeline schedules should not drastically affect the memory and efficiency of the original pipeline schedule.

Intuitively, after partitioning the vocabulary layers to all pipeline devices, computations on each device can be scheduled independently by inserting them cleverly into the existing pipeline passes, as long as the dependencies are still satisfied. However, it is worth noting that partitioning the softmax computation creates several all-reduce operations. These communication barriers create cross-device dependencies.

Driven by this intuition, in Section 4, we discuss how to partition the computation in vocabulary layers to multiple devices and group them into pipeline passes. We observe that the communication barriers (e.g. the all-reduces in softmax) within the vocabulary layers increases the activation memory consumption of the pipeline schedule. As an improvement, we propose two novel algorithms to reduce the number of communication barriers, which reduces the activation memory overhead to minimum.

In Section 5, we discuss how to integrate these vocabulary passes into existing pipeline schedules. Inspired by the framework presented in Qi et al. (2024), we insert the vocabulary passes into the building blocks of existing schedules and simply repeat building blocks to construct pipeline schedules. This relieves us from the hassle of deciding the ordering of vocabulary passes of every microbatch and is naturally generalizable to other schedules.

4 Vocabulary Passes Construction

In this section, we introduce how to split the vocabulary layers into several computation passes after partitioning

them across all pipeline devices, and how to optimize the number of communication barriers.

4.1 A Naïve Approach

In the input layer, each device can perform forward and backward computations independently. We provide details on the input layer in Appendix C, and focus on the output layer for the remainder of this paper.

Figure 4 shows the computation graph of the output layer after partitioning the layer across p devices. We denote the microbatch size as b, sequence length as s, hidden dimension as h and vocabulary size as V.

The partitioned output layer can be grouped into three computation passes F_1, F_2 and B, separated by three all-reduce / reduce communications involving the maximum of logits, the sum of logits and the input gradients respectively. We can overlap these all-reduce communications with transformer layer computation by placing them in a separate stream, as shown in Figure 5.

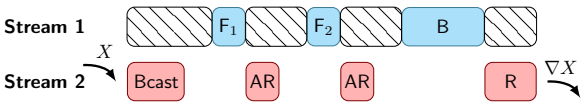

Figure 5. Overlapping all-reduce communication with transformer layer computation.

Figure 6 shows the computation and communication dependencies for a single microbatch. Notably, each of these all-reduce communications will introduce a communication barrier across all pipeline devices, which complicates the pipeline scheduling. As shown later in Section 5.2, the number of communication barriers also increases the activation memory consumption of the pipeline schedule. Therefore, we aim to reduce the number of communication barriers by reordering the operations in the output layer.

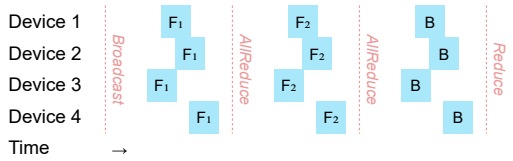

Figure 6. Scheduling dependencies in the naïve output layer implementation.

4.2 Definitions

We detail the computation in the output layer. Given the output of the last transformer layer X and the embedding weights W, we first compute Y:

$$Y = XW^T \tag{1}$$

Then, the (safe) softmax of each sequence is computed as follows:

$$\text{softmax}(Y_{ij}) = \frac{e^{Y_{ij}-m_i}}{\text{sum}_i} \tag{2}$$

where $m_i = \max_k Y_{ik}$ is the maximum of the logits and $\text{sum}_i = \sum_k e^{Y_{ik}-m_i}$ is the sum of logit exponents.

Assuming the cross entropy loss is used, in the backward phase, we have

$$\nabla X = (\text{softmax}(Y) - G)\, W \tag{3}$$

$$\nabla W = (\text{softmax}(Y) - G)^T\, X \tag{4}$$

where G is the ground truth matrix with $G_{ig_i} = 1$ and $G_{ij} = 0$ otherwise, where g_i is label for token i.

4.3 Forward Phase Optimization

Inspired by online softmax (Milakov & Gimelshein, 2018; Dao et al., 2022), we observe that the all-reduce communication for m_i and sum_i can be done after computing the softmax. Instead of using the global maximum and sum, each device instead computes $\text{softmax}'(Y_i)$ using the local maximum and sum from its vocabulary partition. We then have

$$\text{softmax}(Y_{ij}) = \text{softmax}'(Y_{ij}) \times \frac{\text{sum}'_i \times e^{m'_i - m_i}}{\text{sum}_i} \tag{5}$$

where m'_i and sum'_i are the locally computed maximum and sum, respectively.

Using equation 5, we have Algorithm 1 that reduces the 3 communication barriers to 2, which are denoted as C_1 and C_2 respectively.

Algorithm 1 Output layer with 2 communication barriers

function forward_and_backward(W)
  $X \leftarrow$ Receive Broadcast ◁ C_0
  $Y \leftarrow XW^T$
  $m'_i \leftarrow \max_{k=1}^{V/p} Y_{ik}$
  $\text{sum}'_i \leftarrow \sum_{k=1}^{V/p} e^{Y_{ik}-m'_i}$ ◁ S
  $\text{softmax}'(Y_{ij}) \leftarrow \frac{e^{Y_{ij}-m'_i}}{\text{sum}'_i}$
  $m_i \leftarrow$ AllReduce m'_i
  $\text{sum}'_i \leftarrow \text{sum}'_i \times e^{m'_i - m_i}$ ◁ C_1
  $\text{sum}_i \leftarrow$ AllReduce sum'_i
  $\text{softmax}(Y_i) \leftarrow \text{softmax}'(Y_i) \times \frac{\text{sum}'_i}{\text{sum}_i}$
  $\nabla X' \leftarrow (\text{softmax}(Y) - G)W$ ◁ T
  $\nabla W \leftarrow (\text{softmax}(Y) - G)^T X$
  $\nabla X \leftarrow$ Reduce $\nabla X'$ ◁ C_2
end function

The elementwise operations in C_1 only involves tensors of size $[bs]$ as opposed to size $[bs, V/p]$, which greatly reduces the computation pressure when overlapped with transformer layer computation.

4.4 Backward Phase Optimization

We also observe that all three all-reduce / reduce communications can be done after computing the matrix multiplications for the input gradients. Note that

$$\nabla X = \text{softmax}'(Y)W \times \frac{\text{sum}'_i \times e^{m'_i - m_i}}{\text{sum}_i} - GW \quad (6)$$

We can compute $\text{softmax}'(Y)W$ and GW beforehand, and all-reduce ∇X after we obtain m_i and sum_i. Since the matrix multiplications in equation 6 are already computed, computing ∇X within the communication barrier only involves lightweight operations.

This allows us to complete both phases in the output layer with only a single communication barrier C_1, as shown in Algorithm 2.

Algorithm 2 Output layer with 1 communication barrier

function forward_and_backward(W)
  $X \leftarrow$ Receive Broadcast ◁ C_0
  $Y \leftarrow XW^T$
  $m'_i \leftarrow \max_{k=1}^{V/p} Y_{ik}$
  $\text{sum}'_i \leftarrow \sum_{k=1}^{V/p} e^{Y_{ik} - m'_i}$
  $\text{softmax}'(Y_{ij}) \leftarrow \frac{e^{Y_{ij} - m'_i}}{\text{sum}'_i}$ ◁ S
  $A \leftarrow \text{softmax}'(Y)W$
  $B \leftarrow GW$
  $m_i \leftarrow$ AllReduce m'_i
  $\text{sum}'_i \leftarrow \text{sum}'_i \times e^{m'_i - m_i}$
  $\text{sum}_i \leftarrow$ AllReduce sum'_i ◁ C_1
  $\nabla X \leftarrow$ Reduce $A \times \frac{\text{sum}'_i}{\text{sum}_i} - B$
  $\text{softmax}(Y) \leftarrow \text{softmax}'(Y) \times \frac{\text{sum}'_i}{\text{sum}_i}$
  $\nabla W \leftarrow (\text{softmax}(Y) - G)^T X$ ◁ T
end function

Note that the weight gradient step T can be arbitrarily delayed since no other operations depend on it, similar to the idea in zero-bubble strategy (Qi et al., 2023).

We compare the two algorithms with the naïve implementation in Figure 7. By placing the operations in the communication barrier in a separate stream, they will be able to overlap with transformer layer computation. Compared to Algorithm 1, Algorithm 2 introduces a bit more computation overhead (shown in Section 6.5). However, it is still beneficial to reduce the number of communication barriers. As shown in section 5.2, a reduction in the number of communication barriers will help to save activation memory.

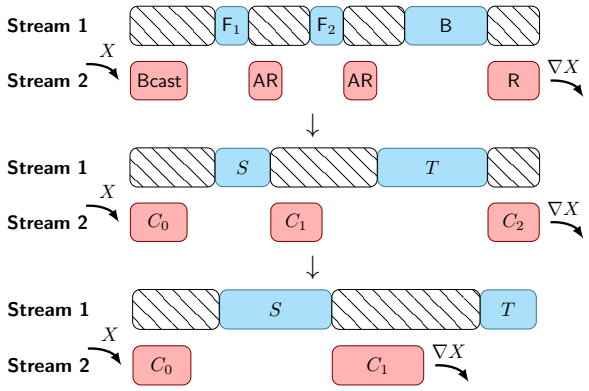

Figure 7. Computation order in the output layer for a single microbatch, corresponding to the naïve implementation, Algorithm 1 and Algorithm 2 respectively.

5 Pipeline Scheduling

In this section, we show a systematic method about how to make minimal changes to typical pipeline schedules to include the output layer passes. We apply our method on two different schedules, 1F1B (Harlap et al., 2018) and *V-Half* (Qi et al., 2024). Despite its popularity, an inherent problem of the 1F1B schedule is an imbalanced activation memory across pipeline devices. In contrast, *V-Half* balances out the activation memory by a V-Shape device placement, reducing the activation memory requirement to half of that of 1F1B. By integrating Vocabulary Parallelism into *V-Half*, we aim to achieve a completely memory-balanced pipeline.

5.1 Scheduling Dependencies

In Algorithms 1 and 2, we perform output layer computation with 2 and 1 communication barriers, respectively. For each microbatch, we have to integrate the output layer passes, S and T, into the pipeline schedules. The pipeline schedule has to adhere to the following constraints:

- All S passes must be scheduled after the forward pass of the last transformer layer completes.
- All T passes must be scheduled after all S passes complete.
- For Algorithm 1 only, the backward pass of the last transformer layer must be scheduled after all T passes complete. In contrast, the T passes can be arbitrarily delayed in Algorithm 2.

For example, Figure 8 shows the scheduling dependencies for a single microbatch in Algorithms 1 and 2 respectively.

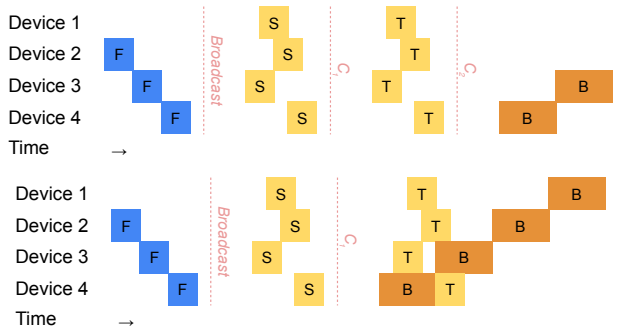

Figure 8. Scheduling Dependencies in Algorithms 1 and 2.

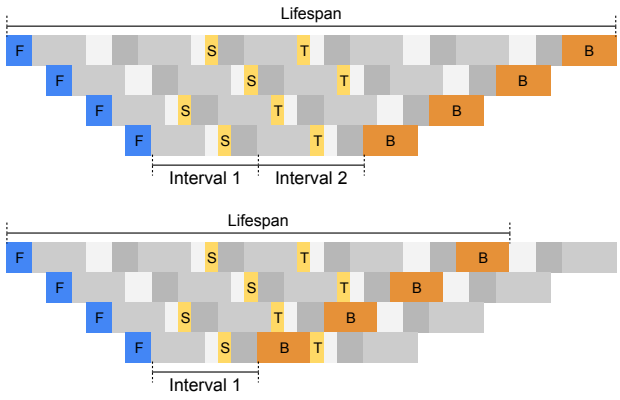

Figure 9. Modified building blocks for the 1F1B schedule corresponding to Algorithm 1 and Algorithm 2. The output layer passes are inserted.

5.2 Methodology

To elegantly integrate these S and T passes into typical pipeline schedules while adhering to the constraints, we follow Qi et al.'s framework (2024) to construct pipeline schedules. In this framework, each pipeline schedule can be structured by its building block, which is defined by the scheduling pattern of each microbatch. By uniformly repeating a building block, we can construct a pipeline schedule, with peak activation memory calculated by dividing its lifespan by its interval. The lifespan is the time between a forward and its corresponding backward, while the interval is the workload of a single microbatch on each device, as illustrated in Figure 9. This approach greatly simplifies dependency management for each microbatch and facilitates memory consumption analysis.

Considering the building block of the schedule, by inserting 2 or 1 intervals (for Algorithms 1 and 2 respectively) between the forward and backward pass of the last transformer layer, we can create space to schedule the output layer computation. Within the repeating interval in the building block, we can schedule output layer passes (S and T) arbitrarily in each pipeline device. We show an example based on 1F1B in Figure 9, where a *one-forward-one-backward-one-output* pattern is strictly followed. The final 1F1B schedules are presented in Figure 10, which is produced by uniformly repeating the building blocks. Additionally, the building block for *V-Half* can be found in Appendix D.

For the peak activation memory, as we insert at most 2 intervals to the lifespan, the peak activation memory only increases by at most 2 microbatches, which is a small constant overhead. This is a remarkable improvement compared to synchronous pipeline schedules, which multiplies the activation memory requirement by 1.5 (see Appendix B.1). Furthermore, the memory savings from balancing the vocabulary parameters outweighs the increase in activation memory.

Notably, as shown in Figure 9, the activation memory increased in terms of microbatches is equivalent to the number of communication barriers, which motivates our optimization of communication barriers in Section 4.

6 Experiments

We construct our experiments to verify: a) Our schedules with Vocabulary Parallelism can bring acceleration; b) Our methods can achieve a balanced memory usage when combined with memory-balanced schedules like *V-Half* (Qi et al., 2024); c) The partitioning of vocabulary layers has a reasonable scaling factor compared to linear scaling.

6.1 Implementation

We implement the pipeline scheduler and the partitioned vocabulary layers based on the open-source Megatron-LM project (Narayanan et al., 2021).

Scheduling under the assumption that backward takes twice time as forward might introduce unnecessary bubbles, especially when these values differ significantly. As a result, we profile the run time of the passes and schedule the S and T passes accordingly[1].

We handle the communication groups in separate streams, allowing the communication barrier to overlap with the transformer layer passes. We map the CUDA streams to separate GPU work queues to achieve this overlapping[2].

[1]The profiling verifies whether the backward pass runs approximately twice the time as the forward pass (unrelated to the vocabulary layer), in order to insert the vocabulary passes appropriately. The pipeline schedule generated will remain unchanged unless this ratio deviates by a certain threshold. In practice, we find that the difference is negligible in most transformer networks, and these differences would not change the pipeline schedule. Hence, this profiling could be viewed as optional.

[2]See https://docs.nvidia.com/deploy/mps/index.html#cuda-device-max-connections.

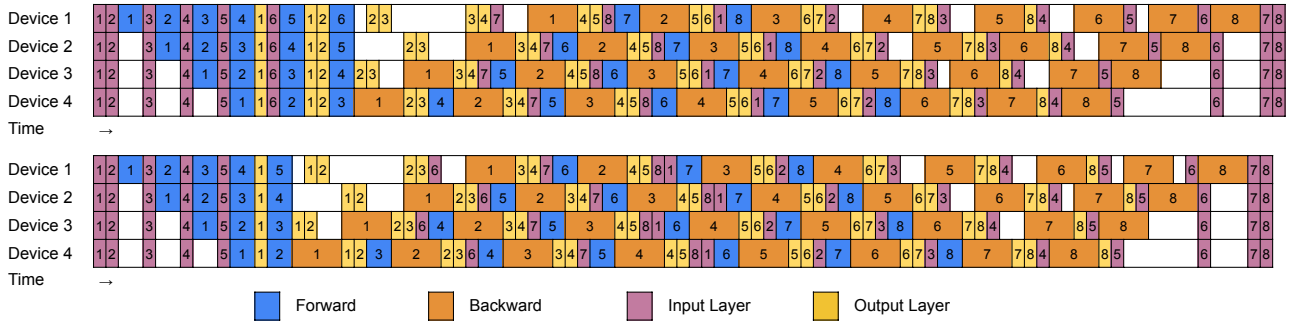

Figure 10. Full 1F1B schedules with Vocabulary Parallelism, corresponding to (a) Algorithm 1 and (b) Algorithm 2 respectively. Algorithm 1 requires activation memory for $p + 2$ microbatches while Algorithm 2 only requires $p + 1$, where p is the number of devices.

However, this would affect communication-computation overlap performance of tensor parallelism (Narayanan et al., 2021) as it relies on single work queues. To mitigate this problem, we set all model parallel communication groups to use high-priority streams. Additionally, both AllReduce and Reduce mentioned on Algorithm 1 and 2 are implemented as NCCL (NVIDIA) AllReduce to avoid imbalanced communication volume across devices.

We also pad the vocabulary size to be a multiple of $2p$ to improve memory alignment in the vocabulary layers, where p is the number of devices. In particular, we observe an approximate 8% increase in performance if our method is applied to 24 devices with padded size 256032 (a multiple of 48), compared to the original value 256008.

Note that our method makes tying input and output embedding weights easier as the input and output embedding weights now have the same device placement and can use the shared weight tensor. This saves GPU memory and avoids the additional all-reduce to synchronize gradients. However, in all our experiments, we adapted the more difficult setting, untying the input and output embedding weights, since it is adapted by some open source models like Llama 3 (Dubey et al., 2024).

6.2 Setup

We compare the following methods implemented on the 1F1B schedule (Harlap et al., 2018).

- Baseline: The naïve implementation in Megatron-LM. It distributes the transformer layers equally to all pipeline stages, while assigning the input and output layers to the first and last pipeline devices. This leads to highly imbalanced compute and memory.
- Redis: Redistributes the transformer layers across pipeline stages to balance out the computation as much as possible. We follow the derivation by Narayanan et al. (2021) to estimate the number of floating point operations in each pipeline stage, and minimize the length of the longest stage.
- Vocab-1: Implements Vocabulary Parallelism with only forward phase optimization, as described in Algorithm 1.
- Vocab-2: On top of Vocab-1, applies backward phase optimization, as described in Algorithm 2.
- Interlaced: Our implementation of the fully synchronous interlaced pipeline proposed by Lin et al. (2024).

Table 1. Settings used in experiments on 1F1B schedule.

PIPELINES (GPUS)	8	16	32
MODEL SIZE	≈ 4B	≈ 10B	≈ 21B
LAYERS	32	48	64
ATTENTION HEADS	24	32	40
HIDDEN SIZE	3072	4096	5120
SEQUENCE LENGTH	2048 / 4096		
MICROBATCH SIZE	1		
NUMBER OF MICROBATCHES	128		
VOCABULARY SIZE	32K / 64K / 128K / 256K		

We experiment the implementations by pretraining GPT-like models of varying model and vocabulary sizes with up to 32 NVIDIA A100 SXM 80G GPUs distributed across 4 nodes, interconnected by a RoCE RDMA network. The running time of each iteration is recorded after several warm-up iterations. We compare the 5 methods under each fixed setting of model and vocabulary size, shown in Table 1.

Our experiments use pure pipeline parallelism to verify that our method improves pipeline parallelism as expected. Given that our method is orthogonal to tensor and data parallelism, the conclusions can be generalized to the production environment where pipeline parallelism is used together with tensor and data parallelism.

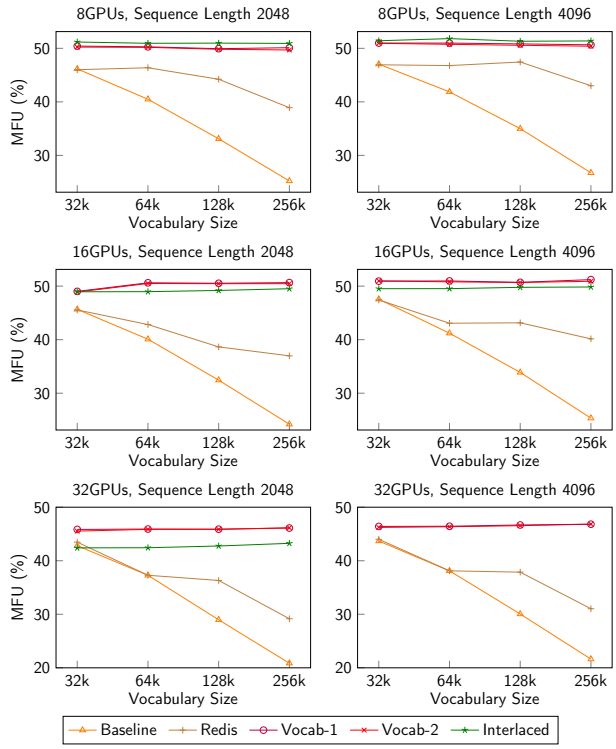

Figure 11. Throughput of different methods on 1F1B. Interlaced OOMs when training with 32 GPUs and sequence length 4096.

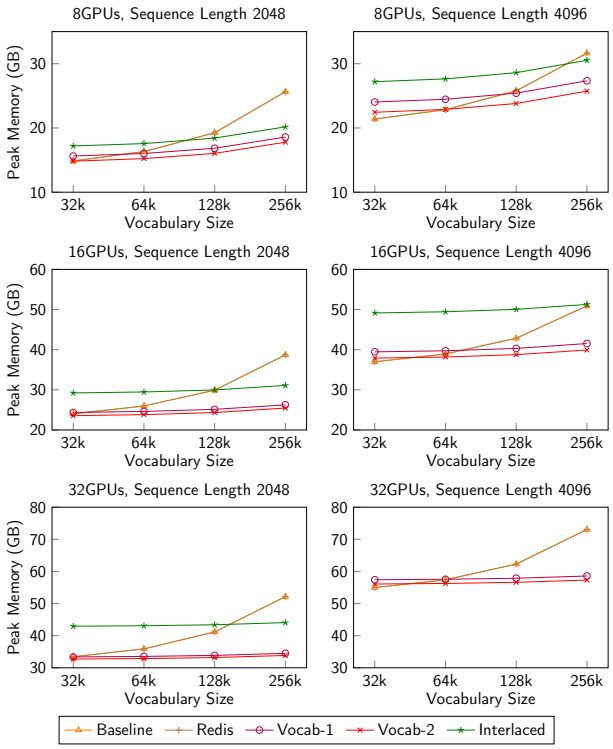

Figure 12. Peak memory of different methods on 1F1B

6.3 Comparison of Methods

We present comparisons of the throughput measured in FLOPs utilization (MFU) and peak memory in Figures 11 and 12, respectively. As shown in the figures, the layer redistribution approach suffers from a significant performance degradation of 8% to 33% for large vocabulary sizes, since the output layer alone already has a higher computation cost than that in the other pipeline devices. Its performance is also highly dependent on the model configuration, or more specifically, the ratio of compute between the vocabulary layer and transformer layers. For example, there is a 9.7% drop in MFU when increasing the vocabulary size from 64k to 128k for the 10B model with sequence length 2048, but that is not observed with sequence length 4096. In contrast, the Vocab and Interlaced approaches have a consistent MFU when scaling up the vocabulary sizes. Vocabulary Parallelism outperforms the interlaced pipeline under a multi-node setup, due to its overlapped communication. For the 21B model, Vocabulary Parallelism outperforms the interlaced pipeline by 6.7% to 8.2% in MFU.

For peak memory usage, the naïve implementation and layer redistribution approaches have an imbalanced parameter memory, leading to high peak memory for large vocabulary sizes. Although the Vocabulary Parallelism methods require extra activation memory, it is effectively negligible when we scale up the pipeline parallel size. However, the interlaced pipeline requires 1.5 times activation memory compared to 1F1B. This resulted in out-of-memory when training the 21B model with sequence length 4096.

6.4 Memory-Balanced Schedule

We show that our method can achieve a balanced memory usage by applying Vocab-1 on the *V-Half* schedule (Qi et al., 2024), a memory-balanced schedule. The implementation is based on the open-sourced *V-Half* implementation by Qi et al. (2024). To support division into virtual pipeline chunks, we adopt different configurations in the experiments, as shown in Table 2.

Table 2. Settings used in experiments on *V-Half* schedule.

Pipelines (GPUs)	16	24	32
Model Size	≈ 7B	≈ 16B	≈ 30B
Layers	32	48	64
Attention Heads	32	40	48
Hidden Size	4096	5120	6144
Sequence Length	2048 / 4096		
Microbatch Size	1		
Number of Microbatches	128		
Vocabulary Size	32k / 64k / 128k / 256k		

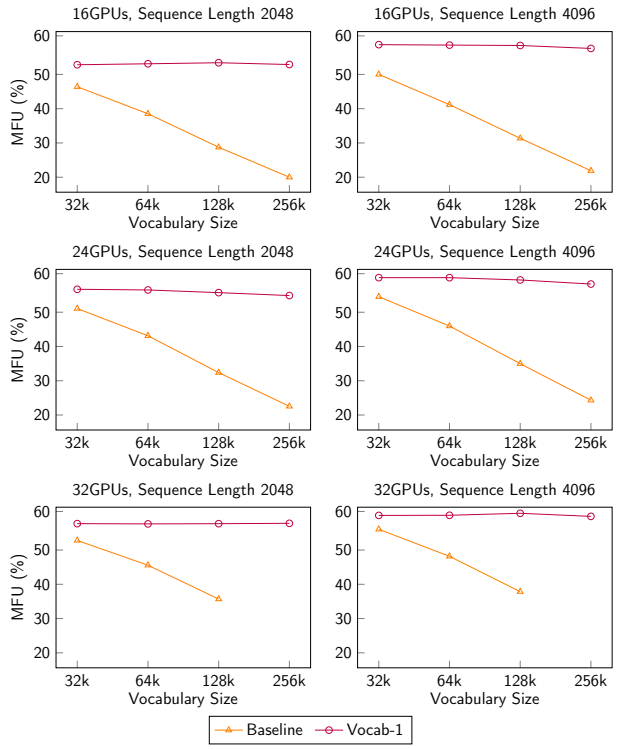

Figure 13. Throughput of different methods on *V-Half*. Baseline OOMs when training with 32 GPUs and vocabulary size 256k.

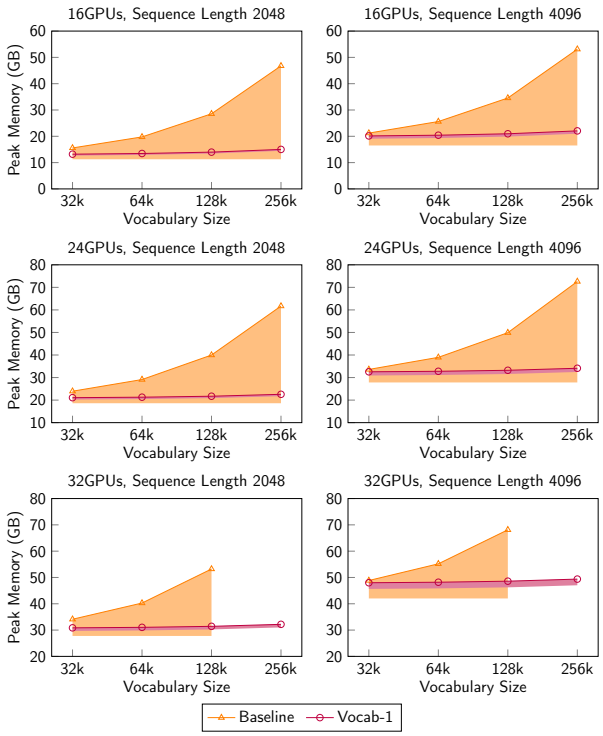

Figure 14. Peak memory of different methods on *V-Half*. The shaded area denotes the range of maximum allocated memory for all devices.

We compare the naïve *V-Half* schedule implementation and that incorporated with Vocab-1. The throughput and peak memory for each pipeline device are shown in Figures 13 and 14 respectively. The naïve implementation resulted in out-of-memory in cases with 32 GPUs and a 256k vocabulary size.

Similar to the previous experiments, the baseline suffers from a huge performance drop when we increase the vocabulary size, while Vocab-1 maintains a steady MFU, consistently outperforming the baseline by 7.2% to 143%. Besides, the baseline has a significant memory imbalance across pipeline devices with up to 45GB difference, while Vocab-1 achieves a balanced memory usage across pipeline devices. Although the first pipeline device still holds slightly more parameters due to positional and token type embedding, the extra memory required is a small constant. In our experiments, this is less than 2.5GB.

6.5 Scaling Analysis of Vocabulary Layers

We analyze the scalability of vocabulary layers in our Vocabulary Parallelism implementation. Using a vocabulary size of 256k, we measure the average throughput of all S and T passes across all devices in our implementation. We compare this with the ideal scenario where the vocabulary layers linearly scale (i.e. p times of the original throughput when distributed to p devices).

The output layers for Vocab-1 and Vocab-2 are considered separately. The time used for the communications is not included since it overlaps with other computation. The results are shown in Table 3.

Table 3. The scaling factor of vocabulary layer computation relative to linear scaling on sequence lengths 2048 and 4096.

SEQ	LAYER	8GPU	16GPU	32GPU
2048	OUTPUT-VOCAB-1	91.29%	84.22%	80.59%
	OUTPUT-VOCAB-2	86.72%	79.84%	75.93%
	INPUT	39.99%	28.85%	15.18%
4096	OUTPUT-VOCAB-1	93.21%	88.02%	85.24%
	OUTPUT-VOCAB-2	88.36%	83.42%	79.66%
	INPUT	27.69%	15.52%	8.35%

Parallelizing the vocabulary layers comes with some computation overhead, which can be attributed to two causes. Firstly, partitioning the vocabulary layers will reduce the model FLOPs utilization (MFU) of GPU kernels as the operations are smaller and hence less parallelized. Secondly, this brings extra computation, especially for the input layer where all devices have to construct the output tensor, whose

size is independent of the size of the vocabulary partition. However, it's worth noting that both input and output still only take a small portion of the computation of the entire model after being partitioned.

7 Conclusion and Future Work

In this work, we identified the problem that when training LLMs with pipeline parallelism, vocabulary layers causes non-negligible imbalance for both compute and memory. Existing methods either fails to achieve a balance or introduce significant performance overhead to the original pipeline schedule. To address this issue, we proposed Vocabulary Parallelism, a method that partitions vocabulary layers evenly to pipeline devices and integrates them into existing pipeline schedules. Our method achieves compute and memory balance for the vocabulary layers. As a result, experiments shows that it improves the throughput by up to 51% while also reduces peak memory consumption compared to existing methods.

Although our implementation of the vocabulary layers are pure python-based, we find that similar optimizations to Algorithm 2 opens an opportunity of fusing the forward and backward pass in CUDA kernels to avoid writes/reads of the softmax results, which can be huge in long-context large-vocabulary settings, to main memory, similar to the rationale of FlashAttention (Dao et al., 2022). Also, while our work focuses on the imbalanced vocabulary layers for pure text-based LLMs, we believe the embedding layers for multimodal LLMs suffer from the same problem and can be further explored.

References

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Kim, T., Kim, H., Yu, G.-I., and Chun, B.-G. Bpipe: Memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*, pp. 16639–16653. PMLR, 2023.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pp. 6543–6552. PMLR, 2021.

Lin, Z., Miao, Y., Zhang, Q., Yang, F., Zhu, Y., Li, C., Maleki, S., Cao, X., Shang, N., Yang, Y., Xu, W., Yang, M., Zhang, L., and Zhou, L. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 347–363, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. GitHub - NVIDIA/nccl: Optimized primitives for collective multi-GPU communication — github.com. https://github.com/NVIDIA/nccl. [Accessed 31-10-2024].

Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

Qi, P., Wan, X., Amar, N., and Lin, M. Pipeline parallelism with controllable memory. *arXiv preprint arXiv:2405.15362*, 2024.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

Tao, C., Liu, Q., Dou, L., Muennighoff, N., Wan, Z., Luo, P., Lin, M., and Wong, N. Scaling laws with vocabulary: Larger models deserve larger vocabularies. *arXiv preprint arXiv:2407.13623*, 2024.

Team, G., Riviere, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., Mesnard, T., Shahriari, B., Ramé, A., et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wei, T., Zhu, B., Zhao, L., Cheng, C., Li, B., Lü, W., Cheng, P., Zhang, J., Zhang, X., Zeng, L., et al. Skywork-moe: A deep dive into training techniques for mixture-of-experts language models. *arXiv preprint arXiv:2406.06563*, 2024.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.