BPIPE: Memory-Balanced Pipeline Parallelism for Training Large Language Models

Taebum Kim [1 2] **Hyoungjoo Kim** [1] **Gyeong-In Yu** [1] **Byung-Gon Chun** [1 2]

Abstract

Pipeline parallelism is a key technique for training large language models within GPU clusters. However, it often leads to a memory imbalance problem, where certain GPUs face high memory pressure while others underutilize their capacity. This imbalance results in suboptimal training performance, even when the overall GPU memory capacity is sufficient for more efficient setups. To address this inefficiency, we propose BPIPE, a novel approach for achieving memory balance in pipeline parallelism. BPIPE employs an activation balancing method to transfer intermediate activations between GPUs during training, enabling all GPUs to utilize comparable amounts of memory. With balanced memory utilization, BPIPE enhances the training efficiency of large language models like GPT-3 by eliminating redundant recomputations or increasing the micro-batch size. Our evaluation conducted on 48 A100 GPUs across six nodes interconnected with HDR InfiniBand shows that BPIPE accelerates the training of GPT-3 96B and GPT-3 134B models by 1.25x-2.17x compared to Megatron-LM, a state-of-the-art framework for training large language models.

1. Introduction

After the advent of the Transformer architecture (Vaswani et al., 2017), there has been a dramatic increase in the size of language models (Brown et al., 2020; Smith et al., 2022; Zhang et al., 2022; Chowdhery et al., 2022; Ouyang et al., 2022; Thoppilan et al., 2022; OpenAI, 2023). These models show astonishing results in a wide range of applications by exploiting more than a hundred billion parameters. Such an overwhelming number of parameters incurs high memory pressure, making large language model (LLM) training challenging. When training a model in mixed precision (Micikevicius et al., 2017) with the Adam (Kingma & Ba, 2014) optimizer, we need 20 bytes of memory for each model parameter (Smith et al., 2022). Hence, training a GPT-3 175B model needs more than 3,000 GiB to store the model parameters and optimizer states. Yet, no GPU exists whose memory capacity satisfies the requirement.

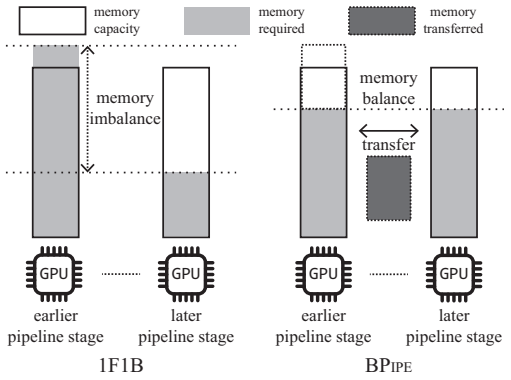


Figure 1. An illustration of how BPIPE deals with a memory imbalance problem. With the 1F1B pipeline schedule (left), the memory requirement of an earlier pipeline stage could exceed the memory capacity of a GPU. In that case, training is not feasible even if the total aggregated memory is sufficient for the memory requirement. BPIPE (right) balances the imbalanced memory requirement by transferring intermediate activations between earlier and later stages.

A few methods, such as model parallelism and activation recomputation (Griewank & Walther, 2000; Kirisame et al., 2021), alleviate the memory pressure to satisfy the requirement. Model parallelism partitions the model parameters and optimizer states across multiple GPUs so that each GPU stores a subset of the model parameters. It is further classified into tensor parallelism (Shazeer et al., 2018; Shoeybi et al., 2019) and pipeline parallelism (Huang et al., 2019), where tensor parallelism splits the operations across GPUs and pipeline parallelism splits the layers across GPUs. On the other hand, activation recomputation releases intermediate activations from memory right after forward compu-

[1]FriendliAI Inc., Seoul, Korea [2]Department of Computer Science and Engineering, Seoul National University, Seoul, Korea. Correspondence to: Byung-Gon Chun <bgchun@friendli.ai>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

tation and recomputes them during backward computation. Since model parallelism and activation recomputation add communication or computation overhead, how we configure them affects training performance significantly. Therefore, finding the configuration that achieves maximum performance and then scaling up training with data parallelism is essential for efficient LLM training.

However, due to its nature, pipeline parallelism could hinder finding the optimal configuration. Unlike tensor parallelism, pipeline parallelism assigns each GPU to handle a separate pipeline stage that computes different layers in a model. Accordingly, each pipeline stage has data dependency on others and results in computation stalls until the required data have arrived, commonly known as a pipeline bubble. To minimize the bubble, a 1F1B (one-forward and one-backward) pipeline schedule (Narayanan et al., 2019; Fan et al., 2021) splits an input batch into micro-batches and processes forward computation and backward computation alternately. In order to saturate all pipeline stages, earlier stages should reserve more memory for computing more forward micro-batches than later stages. Consequently, a memory imbalance exists across the pipeline stages, and executing the model fails if the earlier stages run out of memory, as illustrated in Figure 1.

Our key observation is that the later stages cannot utilize the same amount of GPU memory as the earlier stages require to precompute forward micro-batches. Therefore, if we can exploit the spare memory of later stages as extra memory of the earlier stages, the memory pressure will be relieved with a balanced memory load. In addition, reduced memory pressure allows us to utilize more memory to accelerate training by avoiding redundant recomputations, increasing the micro-batch size, or decreasing the model parallelism degree.

To this end, we propose BPIPE, a memory-balanced pipeline parallelism approach with an activation balancing method to resolve the memory imbalance. While training, BPIPE flattens the memory usage of pipeline stages by transferring activations between earlier and later stages. We propose a transfer scheduling algorithm to minimize the number of transfers while preserving the computation correctness. Furthermore, we design a pair-adjacent stage assignment to make transfers not affect the training time. As a result, BPIPE achieves a balanced GPU memory usage and facilitates efficient LLM training.

We have implemented BPIPE on Megatron-LM (Korthikanti et al., 2022). Our evaluation on six HPE Apollo 6500 8-GPU A100 nodes with 800 Gbps cross-node bandwidth shows that BPIPE can accelerate training GPT-3 96B and GPT-3 134B models by 1.25x-2.17x by executing more efficient training configurations with fewer recomputations and larger micro-batch sizes.

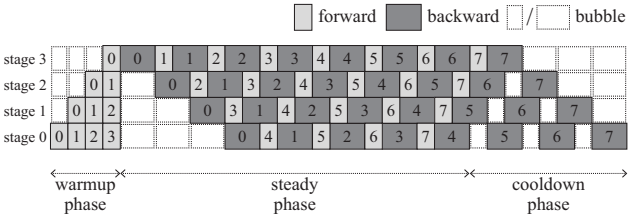


Figure 2. An illustration of a 4-way 1F1B pipeline schedule with eight micro-batches. The memory pressure of each pipeline stage increases during the warmup phase, stays constant at the steady phase, and decreases within the cooldown phase. After the cooldown phase, parameters are updated with accumulated gradients of each micro-batch. A number in either forward or backward denotes the micro-batch index.

2. Background & Motivation

When training LLMs with limited GPU resources, model parallelism is necessary to split the model into multiple partitions and make each part fit into a single GPU. Depending on how the model is divided, we can classify model parallelism into two categories: tensor parallelism and pipeline parallelism.

Tensor parallelism partitions an operation so that each GPU executes the same operation with partial inputs. An additional synchronize operation, usually collective communication such as all-reduce and all-gather, follows the partitioned operation. Owing to its costly synchronization, tensor parallelism is known to be practical when used within a machine boundary (Narayanan et al., 2021b).

On the contrary, pipeline parallelism partitions the layers of a model into multiple stages and distributes them across the GPUs. While training, two consecutive pipeline stages exchange intermediate activations or gradients with point-to-point communication. Since the point-to-point communication of pipeline parallelism adds small overheads compared to the synchronization of tensor parallelism, the pipeline parallelism degree can increase along with the model size.

However, increasing the degree of pipeline parallelism yields a *memory imbalance problem* between pipeline stages. Using the 1F1B pipeline schedule, as illustrated in Figure 2, the first stage has to store activations as many micro-batches as the pipeline parallelism degree during the warmup phase. For the other stages, the number of micro-batches in the warmup phase decreases linearly, so the last stage holds activations of a single micro-batch. Consequently, the imbalance of memory usage exists across the pipeline stages, and its magnitude amplifies with an escalation in the pipeline parallelism degree.

Figure 3 shows the memory usage of each pipeline stage when training a GPT-3 13B (Brown et al., 2020) model

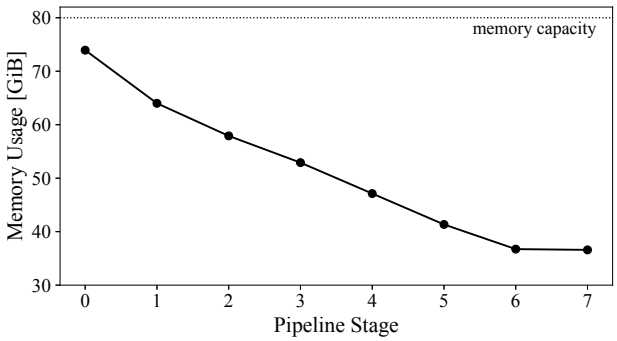


Figure 3. Memory imbalance among pipeline stages when training a GPT-3 13B model with 8-way pipeline parallelism. The micro-batch size is 1, and recomputation is not applied. The memory difference between the first and last stages is 37 GiB.

with 8-way pipeline parallelism using 8 NVIDIA A100 80 GiB GPUs. In 8-way pipeline parallelism, the first stage computes eight micro-batches during the warmup phase. Therefore, the difference in the number of micro-batches between stage 0 and stage 7 is seven, causing a 37 GiB memory imbalance. In other words, stage 7 only utilizes up to half of the memory due to the imbalance. Even worse, attempting to accelerate training by utilizing the unused memory of the last pipeline stage (e.g., increasing the micro-batch size) might fail because the first stage no longer has enough memory.

Asymmetric partitioning of the pipeline stage (i.e., later stages contain more layers than earlier stages) might eliminate the imbalance, but it incurs inefficiency because the pipeline latency is minimized when each pipeline stage has the same computation time (Narayanan et al., 2019; Fan et al., 2021; Li et al., 2021b; Zheng et al., 2022). Alternatively, Chimera (Li & Hoefler, 2021) relieves the imbalance by replicating model parameters so that each GPU holds two pipeline stages. However, since replicating model parameters doubles the memory usage, the overall memory usage increases for the same pipeline degree. To the best of our knowledge, BPIPE is the first work that speeds up training LLMs by addressing the memory imbalance with smart activation transfer across GPUs.

3. Method

In this section, we describe an *activation balancing* method that solves the memory imbalance problem by transferring activations between pipeline stages. First, we formalize the memory imbalance and provide a high-level view of the activation balancing. Then, we define the balanced memory objective and demonstrate an activation transfer scheduling algorithm that achieves the objective. Finally, we describe how we assign pipeline stages to GPUs to minimize the transfer time.

3.1. Pipeline Memory Imbalance

On p-way pipeline parallelism with m micro-batches, the memory usage of the pipeline stage s can be expressed as $M(s) = W(s)+A(s)$, where $A(s)$ is the maximum amount of saved activations and $W(s)$ is the size of model parameters including optimizer states. To simplify, assume that a model has repetitive layers which account for almost all of the model parameters, such as Transformer models (Devlin et al., 2018; Raffel et al., 2020; Brown et al., 2020). If all pipeline stages have an even number of layers, $W(s) = W_0$ is constant. Then, $A(s)$ that varies with the pipeline stage s determines the memory usage. If we define $\mu(s)$ as the maximum number of saved micro-batches on stage s, $A(s)$ becomes $A(s) = A_0\mu(s)$, where A_0 is the size of saved activations per micro-batch, which is also a constant value when each pipeline stage has the same number of layers. As a result, $M(s)$ can be written as the following.

$$M(s) = W_0 + A_0\mu(s) \tag{1}$$

According to Figure 2, each pipeline stage in the 1F1B pipeline schedule possesses at most $\mu(s) = \min(p - s, m)$. In the practical case, $m \gg p$ is satisfied because such a case fully saturates all pipeline stages. Therefore, $\mu(s)$ of the 1F1B pipeline schedule satisfies the following relation.

$$\mu(s) = p - s \tag{2}$$

Substituting $\mu(s)$ in Eq. 1 with $p - s$, Eq. 1 denotes that the difference in the number of saved micro-batches is a cause of the imbalance of $M(s)$.

3.2. Activation Balancing

BPIPE eliminates the memory imbalance by reducing the difference in the number of saved micro-batches for all pipeline stages. Eq. 1 and Eq. 2 show that the memory usage of the pipeline stage decreases linearly as the pipeline stage increases. Accordingly, we pair each stage s with stage $p - s - 1$ to balance $\mu(s)$ and $\mu(p - s - 1)$. The memory imbalance within each pair can be written as the following equation.

$$M(s) - M(p - s - 1) = A_0(p - 2s - 1) \tag{3}$$

Eq. 3 implies that the pipeline stages with $s < \frac{p-1}{2}$ occupy more memory than the stages with $s > \frac{p-1}{2}$. We define the former stages as *evictor*s and the latter stages as *acceptor*s. Each pair then consists of an evictor and an acceptor, where the evictor evicts activations to its pair acceptor to balance $\mu(s)$ and $\mu(p - s - 1)$.

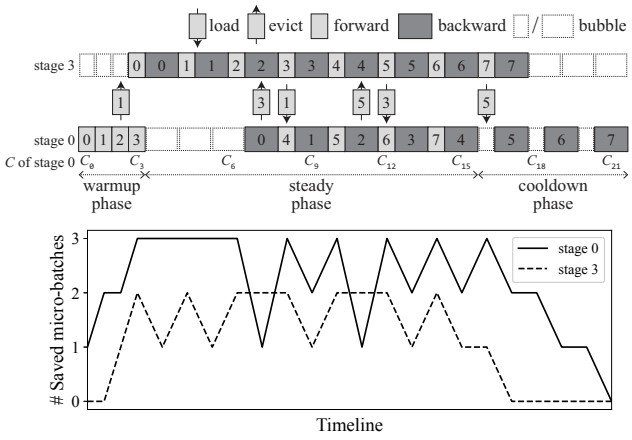


Figure 4. An illustration of the activation balancing and the corresponding changes in the number of saved micro-batches within a 4-way 1F1B pipeline schedule with eight micro-batches. Stage 0 and stage 3 are the pair evictor and acceptor, so stage 0 evicts activations to stage 3 and loads them before the backward computation. All transfers are running in parallel to the forward or backward computation.

Figure 4 illustrates the activation balancing within a 4-way 1F1B pipeline schedule with eight micro-batches. Now, assume we want to make $\mu(0)$ as 3. While stage 0 proceeds the forward computation of the third micro-batch, stage 0 evicts the saved activations to stage 3, a pair acceptor of stage 0. The number of saved micro-batches does not increase after the forward computation due to the eviction. BPIPE always evicts the latest micro-batch among the saved to minimize the number of transfers. Therefore, the second micro-batch is evicted instead of the first micro-batch. The evicted activations should be loaded by stage 0 before processing the corresponding backward computation at the steady phase. Stage 0 evicts another micro-batch before loading because each forward computation and loading increase the number of saved micro-batches by 1, respectively. For example, evicting the micro-batches whose indices are 3 and 5 in Figure 4 represent it. At the cooldown phase, stage 0 loads the activations without the additional eviction because no forward computation is executed along with the loading. As a result, $\mu(0)$ becomes three, as shown in the graph of Figure 4.

3.3. Balanced Memory Objective

From Eq. 2, the sum of the maximum number of saved micro-batches for any evictor-acceptor pair is $\mu(s) + \mu(p - s - 1) = p + 1$. Including a buffer for the additional evictions at the steady phase, the value becomes $p + 2$. Now, the optimally balanced number of micro-batches μ_{opt} is derived as $\mu_{opt} = \lceil \frac{p+2}{2} \rceil$, and the optimal memory balance M_{opt} is $W_0 + A_0\mu_{opt}$.

An objective of the activation balancing is accomplishing $\mu(s) \leq \mu_{opt}$ for all pipeline stages throughout the training. An evictor with pipeline stage s achieves the objective by evicting at most $(p - s) - \mu_{opt} + 1$ micro-batches to its pair acceptor. Simultaneously, the pair acceptor saves at most $s + 1$ micro-batches following the pipeline schedule and $(p-s)-\mu_{opt}+1$ micro-batches from the evictor. $\mu(p-s-1)$ then becomes $p + 2 - \mu_{opt}$, where the value is less or equal to μ_{opt}. Therefore, both the evictor and acceptor achieve the objective.

The memory objective implies that BPIPE does not trigger any transfer of activations for the evictors whose stages already satisfy the objective. In other words, the activation balancing operates when $p \geq 4$, and only the evictors whose pipeline stage satisfies $s \leq \lfloor \frac{p-4}{2} \rfloor$. For example, the innermost pair of pipeline stages, which is stage 1 and stage 2 in Figure 4, does not transfer activations because they already satisfy the objective.

3.4. Transfer Schedule

To achieve the memory objective, we propose a transfer scheduling algorithm. The algorithm gets a pipeline parallelism degree p, a pipeline stage s of an evictor, the number of micro-batches m, and a computation schedule C of 1F1B pipeline as inputs and returns a transfer schedule T for stage s. A computation schedule is an array of *computation decisions*. Each computation decision C_i comprises the type of computation made and the index of the micro-batch being processed. For example, C of pipeline stage 0 in Figure 4 has 22 computation decisions, including pipeline bubbles. C_0 has 0 as a micro-batch index with a $FORWARD$ computation type, C_9 has 1 as a micro-batch index with a $BACKWARD$ computation type, and C_{18} has an empty micro-batch index because the computation type of C_{18} is a $BUBBLE$.

Algorithm 1 describes the details of scheduling for the given inputs. The algorithm traverses the computation schedule, deciding when to evict or load. Within the warmup phase, n_evict micro-batches are evicted to make $\mu(s)$ equal to μ_{opt} (lines 9-13). When the algorithm encounters a $BACKWARD$ type computation decision at i and finds that the micro-batch required to compute the backward (i.e., $C_i.idx$) has been evicted, it loads the micro-batch at $i - 1$ (lines 14-18). However, this load would make $\mu(s)$ exceed μ_{opt} if C_i belongs to the steady phase. In other words, when C_{i-1} is a $FORWARD$ type, both C_{i-1} and the load at $i - 1$ increase $\mu(s)$ so that $\mu(s)$ exceeds μ_{opt}. The algorithm prevents it by evicting an additional micro-batch that will be needed at the furthest future at $i - 2$, whose index is $C_{i-3}.idx$ (lines 19-23). On the other hand, if C_i belongs to the cooldown phase, C_{i-1} is the $BUBBLE$ type which

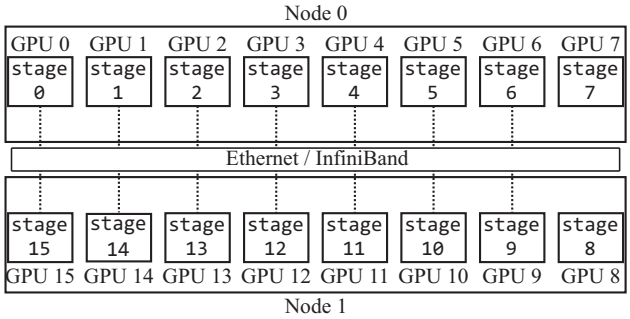
Algorithm 1 Transfer Scheduling Algorithm

Input: p, s, m, 1F1B computation schedule C
Output: transfer schedule T
$\mu_{opt} \leftarrow \lceil \frac{p+2}{2} \rceil$
$n\_evict \leftarrow \min(p - s, m) - \mu_{opt}$
$evicted \leftarrow \emptyset$
$T_i \leftarrow None \ ^\forall 0 \leq i < |C|$
for $i = 0$ **to** $|C| - 1$ **do**
  if $C_i.type = FORWARD$ **then**
    if $\mu_{opt} - 1 \leq C_i.idx < \mu_{opt} - 1 + n\_evict$ **then**
      /* warmup phase */
      $T_i \leftarrow Evict(C_{i-1}.idx)$
      $evicted \leftarrow evicted \cup \{C_{i-1}.idx\}$
    end if
  else if $C_i.type = BACKWARD$ **then**
    /* steady or cooldown phase */
    if $C_i.idx \in evicted$ **then**
      $T_{i-1} \leftarrow Load(C_i.idx)$
      $evicted \leftarrow evicted \setminus \{C_i.idx\}$
      if $C_{i-1}.type = FORWARD$ **then**
        /* steady phase */
        /* evict to keep $\mu(s) \leq \mu_{opt}$ */
        $T_{i-2} \leftarrow Evict(C_{i-3}.idx)$
        $evicted \leftarrow evicted \cup \{C_{i-3}.idx\}$
      end if
    end if
  end if
end for

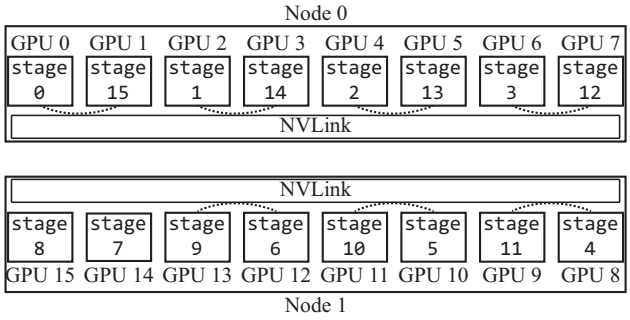
does not increase $\mu(s)$. Thus, $\mu(s)$ does not exceed μ_{opt}, and the algorithm does not evict an additional micro-batch.

The generated transfer schedule T contains an array of *transfer decisions* that interleaved with the 1F1B computation schedule. While processing the 1F1B schedule, BPIPE simultaneously processes the transfer schedule. BPIPE evicts the saved activations of the j-th micro-batch if T_i is $Evict(j)$ and loads them if T_i is $Load(j)$. When T_i is $None$, a default decision in line 6, BPIPE does not process any transfer.

Following the computations with the corresponding transfer decisions, BPIPE achieves the memory objective for all pipeline stages with the minimum number of transfers. The number of transfers is the summation of the number of $Evict$s and $Load$s. We only consider the number of $Evict$s because each eviction requires exactly one corresponding $Load$ before processing the backward computation. Subsequently, we can interpret the micro-batch eviction as analogous to cache eviction. In other words, an evictor manages a cache storage whose capacity is μ_{opt}. It evicts an item (i.e., forward micro-batch) to the memory space of its pair acceptor when the cache is full. Deciding which item to evict is then determined based on a cache eviction policy.



(a) A standard assignment



(b) A pair-adjacent assignment.

Figure 5. An illustration of standard assignment and pair-adjacent assignment for 16-way pipeline parallelism on two nodes, each with 8 GPUs. The dotted lines represent the communication between evictor-acceptor pairs. The standard assignment makes each pair communicate over the slow inter-node link. In contrast, the pair-adjacent assignment lets them transfer their activations over the fast intra-node link.

Unlike common caching scenarios, we know exactly when each item is required in the future. Therefore, as evicting the item that will be needed in the furthest future is optimal (Belady, 1966), our scheduling algorithm minimizes the number of transfers.

Although the algorithm assumes a 1F1B computation schedule as an input, we can extend it to other variants of 1F1B (Narayanan et al., 2021a;b; Yang et al., 2021; Athlur et al., 2022; Zhuang et al., 2022) because they all have a memory imbalance. Appendix A describes how we can extend the algorithm to the interleaved 1F1B pipeline schedule (Narayanan et al., 2021b).

3.5. Pair-Adjacent Assignment

As BPIPE processes a transfer schedule simultaneously with a 1F1B computation schedule, each transfer should take less time than $FORWARD$ or $BACKWARD$ not to affect the training time. To minimize the transfer time, we propose a *pair-adjacent assignment* that locates each evictor-acceptor pair in the same node in a cluster. Within the same node,

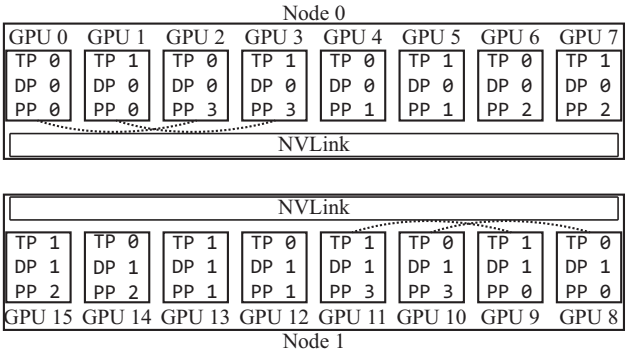


Figure 6. An illustration of how the pair-adjacent assignment assigns pipeline stages to GPUs when 4-way pipeline parallelism with 2-way tensor parallelism and 2-way data parallelism on two nodes, each with 8 GPUs. 'TP' and 'DP' denote the rank of tensor parallelism and data parallelism. 'PP' represents the pipeline stage.

each pair can exploit a high-bandwidth intra-node communication link like NVLink rather than using a relatively slower inter-node communication link like Ethernet or InfiniBand.

Figure 5 illustrates how pipeline stages are assigned to GPUs using 16-way pipeline parallelism on a cluster with two nodes, each of which has 8 GPUs. A standard assignment assigns pipeline stages to GPUs in order, as Figure 5(a) shows. However, it assigns all pipeline pairs to different nodes. Instead, BPIPE assigns pair stages to adjacent GPUs, as shown in Figure 5(b). Each pair then resides in the same node and utilizes fast intra-node communication. Although the pair-adjacent assignment entails additional inter-node communication of dependent data between consecutive pipeline stages, it is beneficial because the bytes transferred between pairs are much bigger than the bytes exchanged between consecutive stages.

When pipeline parallelism is used with either tensor or data parallelism, the pair-adjacent assignment assigns pipeline stages considering other parallelism methods. Figure 6 illustrates how the pair-adjacent assignment operates with data and tensor parallelisms. BPIPE prioritizes pipeline parallelism over data parallelism because the gradient synchronization of data parallelism is performed only once for each training step. On the other hand, BPIPE prioritizes tensor parallelism over pipeline parallelism to ensure that the frequent collective communication of tensor parallelism always takes place within a node. Hence, when the tensor parallelism degree is equal to the number of GPUs in a single node, BPIPE transfers activations across the node boundary. Nevertheless, the activation balancing is feasible across the node boundary in our evaluation setup. In Appendix B, we provide formalized details on the network bandwidth requirements of the activation balancing.

Table 1. Model configurations for evaluation. L denotes the number of layers, D denotes hidden dimension size, and H denotes the number of attention heads. Finally, G and B are the numbers of GPUs used to execute the model and batch size, respectively.

Model	L	D	H	G	B
GPT-3 13B	40	5120	40	8	32
GPT-3 96B	80	9984	104	32	128
GPT-3 134B	84	11520	120	48	192

4. Evaluation

To evaluate BPIPE, we ask the following questions.

- Does BPIPE facilitate faster training of large language models? (§ 4.2)
- Does BPIPE flatten the memory usage of each pipeline stage? (§ 4.3)
- Does BPIPE efficiently evict and load activations without performance degradation? (§ 4.4)

4.1. Implementation and Environment Setup

We have implemented BPIPE on Megatron-LM v3 (Korthikanti et al., 2022). We utilize separate CUDA streams for evicting and loading activations so that activations can be transferred concurrently with either forward or backward computation. In addition, we manually manage CUDA memory for activations that are needed for backward computation and reuse pre-allocated memory after the eviction to avoid unexpected memory fragmentation.

Our evaluations are conducted on a cluster of six HPE Apollo 6500 Gen10 Plus nodes, each of which is equipped with 8 NVIDIA 80 GiB A100 GPUs connected over NVLink and 4 Mellanox 200 Gbps HDR InfiniBand HCAs for communication. All experiments are executed on the NVIDIA PyTorch NGC 22.09 container. We evaluate GPT-3 (Brown et al., 2020) throughout the experiments, one of the most representative LLMs. We use three different model configurations, as shown in Table 1, in which the largest model has 134 billion parameters in total. Sequence length and vocabulary size are 2,048 and 51,200 for all models, respectively, and we use mixed precision training (Micikevicius et al., 2017). Although we evaluate only GPT-3 models, BPIPE is applicable to any model as long as pipeline parallelism is used to train the model.

4.2. Training Performance

To evaluate whether BPIPE can accelerate training, we find the fastest configuration for training GPT-3 96B and GPT-3 134B models, with and without BPIPE. Our baseline is

Table 2. Training configurations of GPT-3 96B and GPT-3 134B models. 'tensor' and 'pipeline' represent the tensor and pipeline parallelism degrees, respectively. The remaining GPUs are used for data parallelism, and we use ZeRO stage-1 data parallelism (Rajbhandari et al., 2020) that splits optimizer states. Moreover, tensor parallelism includes partitioning layer normalization and dropout, as introduced in Korthikanti et al. 2022. 'mb' denotes the micro-batch size, and each value corresponds to a different training configuration.

Model	Model Parallelism ID	tensor	pipeline	mb	Recompute scope
GPT-3 96B	(1)	1	16	1	layer
	(2)	2	8	1,2	layer
	(3)	4	4	1	attention
				2	layer
	(4)	8	2	1	attention
				2,4	layer
	(5)	2	16	1	attention
				2,4	layer
	(6)	4	8	1,2	attention
				4,8	layer
	(7)	8	4	1,2	attention
				4,8	layer
GPT-3 134B	(1)	2	12	1,2	layer
	(2)	4	6	1	attention
				2,4	layer
	(3)	8	3	1	attention
				2,4	layer
	(4)	4	12	1,2	attention
				4	layer
	(5)	8	6	1,2	attention
				4,8	layer

Megatron-LM v3 (Korthikanti et al., 2022), a state-of-the-art LLM training framework. We perform a grid search to find the best configuration as follows. In addition to the notations in Table 1, we define tensor parallelism degree as t, pipeline parallelism degree as p, data parallelism degree as d, and micro-batch size as mb. Then, the following constraints exist.

- $H \bmod t = 0$: The number of attention heads should be divisible by the tensor parallelism degree because tensor parallelism for Transformer layers splits attention heads.
- $8 \bmod t = 0$ (8 is the number of GPUs in a single node): Tensor parallelism is practical when used within a node boundary due to its costly synchronization.
- $L \bmod p = 0$: The number of layers should be divisible by the pipeline parallelism degree, assuming that pipeline parallelism evenly divides the layers.
- $B \bmod (mb \times d) = 0$: The total batch size should be divisible by the micro-batch size times data parallelism degree. The number of micro-batches is $B/(mb \times d)$.
- $t \times d \times p = G$: Multiplication of tensor, data, and pipeline parallelism degrees should be equal to the total number of GPUs.

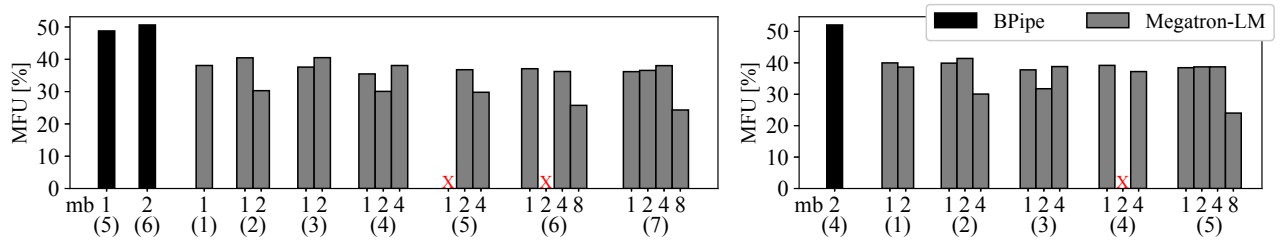
Under the constraints, we enumerate all possible tuples of (t, p, d, mb). We evaluate them with Megatron-LM for each recomputation scope in the order of none, attention, and layer, where the none scope does not recompute any activation, the attention scope recomputes only the self-attention of the Transformer layer, which is known as the selective recomputation (Korthikanti et al., 2022), and the layer scope recomputes the entire Transformer layer. We apply early stopping when succeeding to execute with fewer recomputations because carrying out more recomputations for the same (t, p, d, mb) is inefficient. Then, each (t, p, d, mb) with a recomputation scope composes a single training configuration. For BPIPE, we evaluate the configurations that Megatron-LM cannot execute since the activation balancing does not accelerate training. If BPIPE fails due to the out-of-memory error, we exclude those configurations. As a result, Table 2 lists all feasible training configurations.

We use model FLOPS utilization (MFU) as an evaluation metric, a ratio of the observed throughput to the hardware maximum throughput (Korthikanti et al., 2022). Figure 7 shows that BPIPE can execute the configuration that achieves 52.06% MFU, though Megatron-LM cannot execute it due to the memory imbalance problem. Consequently, BPIPE accelerates training the models by 1.25x compared to the fastest training configuration that Megatron-LM can execute and 2.17x than the most inefficient configuration of Megatron-LM. The raw MFU numbers of Figure 7 are listed in Appendix C.

4.3. Memory Balancing

Figure 8 presents the change in memory usage when using BPIPE. Without the activation balancing, the maximum memory usage difference between the pipeline stages is larger than 25 GiB. The model cannot be executed because the first stage runs out of memory. However, the difference sharply reduces to 10 GiB with BPIPE because BPIPE flattens the memory usage of each evictor-acceptor pair. As a result, BPIPE facilitates faster training with more efficient resource utilization, as Figure 7 shows.

If the model size grows, the pipeline parallelism degree should increase because increasing the tensor parallelism degree is bounded up to the number of GPUs in a single node. Then, the memory imbalance would also increase following the pipeline parallelism degree. For example, Narayanan et al. (2021b) and Korthikanti et al. (2022) use



(a) GPT-3 96B (b) GPT-3 134B

Figure 7. Normalized training throughputs of GPT-3 96B and GPT-3 134B models. Labels of the bars represent the training configurations of Table 2. For example, a bar whose label is (1)-mb1 denotes training configuration with model parallelism ID (1) and the micro-batch size as 1. 'X' denotes that Megatron-LM fails to execute due to the out-of-memory (OOM). On the other hand, BPIPE makes executing all of them possible with the activation balancing. Moreover, they are faster than all the configurations that Megatron-LM can execute.

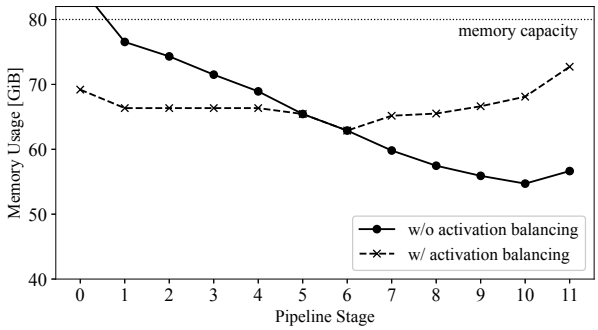


Figure 8. Memory usage of each pipeline stage with the activation balancing compared to without the activation balancing when training a GPT-3 134B model with configuration (4)-mb2 of Table 2. To estimate the memory usage without the activation balancing, we allow the activation balancing only for stage 0 and stage 11 since stage 0 has insufficient memory to execute.

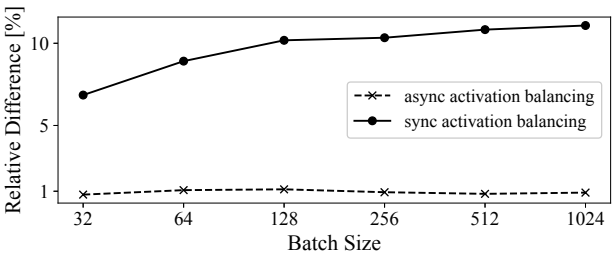


Figure 9. The relative difference in iteration time with various batch sizes when training a GPT-3 13B with 8-way pipeline parallelism. No recomputation is applied, and the number of micro-batches is equal to the batch size.

64-way pipeline parallelism for scaling up GPT-3 model to 1 trillion parameters. It implies that BPIPE can dramatically flatten the memory imbalance, paving the way for more efficient training of LLMs.

4.4. Performance Analysis

We inspect the efficiency of BPIPE by comparing the iteration time before and after applying the activation balancing. Figure 9 shows the additional time cost of the activation balancing, varying the batch size from 32 to 1,024 when training a GPT-3 13B model with 8-way pipeline parallelism. The cost occupies 1.1% even when the number of micro-batches is sufficiently large.

BPIPE achieves a low time cost by virtue of the asynchronous activation transfer. If we transfer activations synchronously, the time overhead shoots up to 11%, as Figure 9 shows. On the contrary, asynchronous transfer overlaps with the computation because the computation time takes far longer than the transfer time. The transfer overlaps even if the model size grows because the computation time increases more steeply than the transfer time, as Table 3 shows. Furthermore, the size of activations to transfer increases with fewer recomputations. However, the transfer time does not exceed the forward time even for the no recomputation, as Table 4 shows.

5. Related Work

Data parallelism. Data parallelism replicates model parameters and optimizer states across GPUs (Li et al., 2020). An input batch is divided into multiple mini-batches for each training step, so each GPU conducts forward-backward computation with a different mini-batch. After each GPU finishes a single training step, all GPUs carry out an all-reduce collective communication to synchronize gradients and then update the model parameters. Thereby, all GPUs always have the same parameters throughout the training. Since collective communication occurs only once in a single

Table 3. Time breakdown of transfer, forward, and backward. All times are the elapsed time for processing a single micro-batch. For GPT-3 13B, 8-way pipeline parallelism is used with the attention recomputation scope and the micro-batch size is 1. For GPT-3 96B and GPT-3 134B, we use configuration (6)-mb2 and (4)-mb2 of Table 2, respectively.

Model	transfer	forward	backward
GPT-3 13B	7.63 ms	36.32 ms	83.57 ms
GPT-3 96B	15.63 ms	143.37 ms	287.90 ms
GPT-3 134B	12.06 ms	126.87 ms	256.30 ms

Table 4. Time breakdown by varying the recomputation scope of a GPT-3 13B model with 8-way pipeline parallelism.

Recompute scope	transfer	forward	backward
none	22.48 ms	36.32 ms	74.12 ms
attention	7.63 ms	36.32 ms	83.57 ms
layer	0.58 ms	36.32 ms	110.33 ms

training step, data parallelism is the best way to scale up training. Beyond replicating model parameters, ZeRO (Rajbhandari et al., 2020) and Fully Sharded Data Parallelism (FSDP) (Zhao et al., 2023) reduce the memory pressure by splitting optimizer states. They propose more aggressive splits, including gradients and model parameters, with more frequent collective communication.

Tensor parallelism. Mesh-TensorFlow (Shazeer et al., 2018) introduces an abstraction that can express arbitrary operation partitioning. It also presents that data parallelism is one of the cases of tensor parallelism that splits tensors across the batch dimension. Moreover, some model-specific tensor parallel strategies exist that efficiently reduce memory pressure with moderate synchronization costs. For example, Megatron-LM (Shoeybi et al., 2019) proposes an efficient partitioning strategy of two consecutive matrix multiplications in the Transformer layer. Additionally, partitioning along the sequence dimension of the Transformer layer is also feasible (Li et al., 2021a; Korthikanti et al., 2022).

Pipeline parallelism. To further minimize the pipeline bubble, several pipeline schedules (Narayanan et al., 2021a;b; Li & Hoefler, 2021; Yang et al., 2021; Athlur et al., 2022; Zhuang et al., 2022) have been proposed. Such schedules stem from the 1F1B schedule but sacrifice memory usage or even affect model correctness. Furthermore, token-level scheduling (Li et al., 2021b) is proposed for an auto-regressive language model (Brown et al., 2020). However, it is effective when the batch size is much smaller than the pipeline parallelism degree, which is unlikely in practical LLM training scenarios.

Activation recomputation. Since recomputation incurs an overhead equal to the forward computation time, activations to discard and reserve should be well-decided. Chen et al. (2016) proposed an algorithm that can reduce memory consumption to sub-linear costs for a linear computation chain. Checkmate (Jain et al., 2020) solves a mixed integer linear program to find an optimal recomputation strategy for an arbitrary model structure. When models have repetitive layers, recomputing activations layer by layer is widely adopted in practice. In addition to the layerwise strategy, recent research (Korthikanti et al., 2022) has shown that recomputing only the attention of the Transformer layer can efficiently reduce the memory pressure with small computation overhead.

CPU offloading. In general, a CPU has orders of magnitude larger, cheaper, and slower memory than a GPU. Accordingly, several approaches have been proposed that exploit CPU memory as a swap storage of limited GPU memory (Rhu et al., 2016; Wang et al., 2018; Pudipeddi et al., 2020; Peng et al., 2020; Huang et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021). However, utilizing CPU memory is not scalable because communication between CPU and GPU is slow due to the limited communication bandwidth of PCIe.

Automatic search for the optimal training configuration. Given a deep learning model and a cluster environment, various systems exist that automatically search for the optimal execution plan (Wang et al., 2019; Jia et al., 2019; Lepikhin et al., 2020; Rasley et al., 2020; Tarnawski et al., 2021; Xu et al., 2021; Bian et al., 2021; Karakus et al., 2021; Jia et al., 2022; Zheng et al., 2022; Unger et al., 2022). Such systems configure a search space with a cost model and explore it to find the best result. As BPIPE rewrites the memory usage of pipeline parallelism, BPIPE can expand the search space and help find better configurations.

6. Conclusion

We propose BPIPE, a memory-balanced pipeline parallelism method for training LLMs. With the activation balancing that transfers intermediate activations between pipeline stages, BPIPE resolves the memory imbalance problem of pipeline parallelism. While training, all pipeline stages utilize a comparable amount of memory by storing a balanced amount of activations in the unit of micro-batch. Our evaluation shows that BPIPE speeds up training large-scale GPT-3 models by executing faster training configurations that are not feasible without BPIPE.

References

Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 472–487, 2022.

Belady, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2): 78–101, 1966.

Bian, Z., Liu, H., Wang, B., Huang, H., Li, Y., Wang, C., Cui, F., and You, Y. Colossal-ai: A unified deep learning system for large-scale parallel training. *arXiv preprint arXiv:2110.14883*, 2021.

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.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 431–445, 2021.

Griewank, A. and Walther, A. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1341–1355, 2020.

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.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

Jia, X., Jiang, L., Wang, A., Xiao, W., Shi, Z., Zhang, J., Li, X., Chen, L., Li, Y., Zheng, Z., et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 673–688, 2022.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

Karakus, C., Huilgol, R., Wu, F., Subramanian, A., Daniel, C., Cavdar, D., Xu, T., Chen, H., Rahnama, A., and Quintela, L. Amazon sagemaker model parallelism: A general and flexible framework for large model training. *arXiv preprint arXiv:2111.05972*, 2021.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic tensor rematerialization. In *ICLR*. OpenReview.net, 2021.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Li, S. and Hoefler, T. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. *arXiv e-prints*, pp. arXiv–2105, 2021a.

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, 2021b.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021a.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. 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, 2021b.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/arXiv.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 891–905, 2020.

Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., and Bharadwaj, S. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P. J., et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

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.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.

Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13. IEEE, 2016.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

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.

Tarnawski, J. M., Narayanan, D., and Phanishayee, A. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34: 24829–24840, 2021.

Thoppilan, R., Freitas, D. D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H. S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhou, Y., Chang, C., Krivokon, I., Rusch, W., Pickett, M., Meier-Hellstern, K. S., Morris, M. R., Doshi, T., Santos, R. D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Aguera-Arcas, B., Cui, C., Croak, M., Chi, E. H., and Le, Q. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022.

Unger, C., Jia, Z., Wu, W., Lin, S., Baines, M., Narvaez, C. E. Q., Ramakrishnaiah, V., Prajapati, N., McCormick, P., Mohd-Yusof, J., et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 267–284, 2022.

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.

Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 41–53, 2018.

Wang, M., Huang, C.-c., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–17, 2019.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C., and De Sa, C. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M. T., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Nguyen, B., Chauhan, G., Hao, Y., and Li, S. Pytorch FSDP: experiences on scaling fully sharded data parallel. *CoRR*, abs/2304.11277, 2023.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

Zhuang, Y., Zhao, H., Zheng, L., Li, Z., Xing, E. P., Ho, Q., Gonzalez, J. E., Stoica, I., and Zhang, H. On optimizing the communication of model parallelism. *arXiv preprint arXiv:2211.05322*, 2022.

A. Transfer Scheduling for Interleaved 1F1B Pipeline Schedule

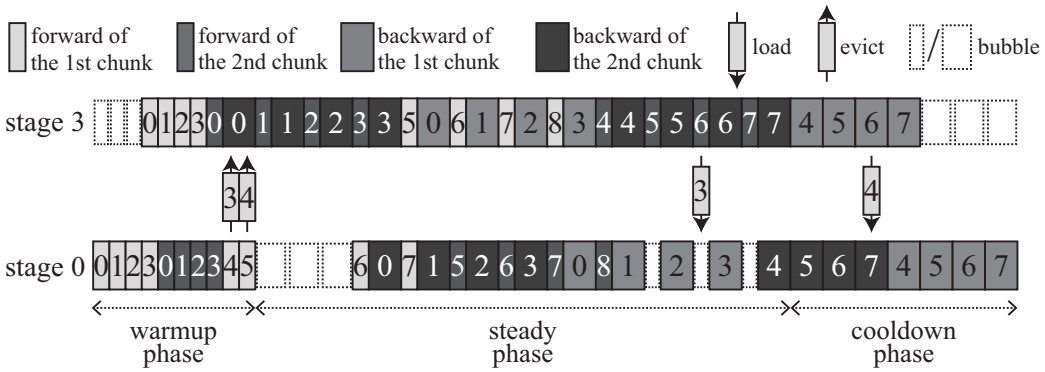


Figure 10. An illustration of the activation balancing within a 4-way interleaved 1F1B pipeline schedule with eight micro-batches and two model chunks for each pipeline stage.

Narayanan et al. (2021b) propose an interleaved 1F1B pipeline schedule that reduces the pipeline bubble. BPIPE can apply the activation balancing to the interleaved 1F1B schedule as the following. In the interleaved schedule, each pipeline stage computes model chunks, which are non-contiguous subsets of layers (Narayanan et al., 2021b). Assume a pipeline stage s of p-way pipeline parallelism. When each pipeline stage has v model chunks, it processes $\mu(s) = p(v-1) + 2(p-s-1) + 1$ micro-batches before the first backward computation. Then, μ_{opt} is derived as the following.

$$\begin{aligned}\mu_{opt} &= \lceil \frac{\mu(s) + \mu(p-s-1) + 1}{2} \rceil \\ &= \lceil \frac{p(v-1) + 2(p-s-1) + 1 + p(v-1) + 2(p-(p-s-1)-1) + 1 + 1}{2} \rceil \\ &= pv + 1\end{aligned}$$

The transfer scheduling of the interleaved 1F1B pipeline schedule is similar to that of the vanilla 1F1B schedule. Figure 10 illustrates how the activation balancing operates with a 4-way interleaved 1F1B pipeline schedule with two model chunks. μ_{opt} becomes 9, so stage 0 evicts two micro-batches to stage 3. In the steady or cooldown phase, it loads the evicted micro-batches to perform the backward computations.

B. Network Bandwidth Requirement for Activation Balancing

Table 5. Definition of the variables.

Variable	Definition
L	number of layers
H	number of attention heads
D	hidden dimension size
l	sequence length
p	pipeline parallelism degree
t	tensor parallelism degree
b	micro-batch size
f	forward computation time of a single micro-batch [sec]

Since each transfer of the activation balancing should finish earlier than the forward computation of a single micro-batch, we can calculate the required network bandwidth of a single GPU with the forward computation time. Using the variables in Table 5, we can derive the bandwidth requirements as the following. According to Korthikanti et al. (2022), the bytes of activation memory per each forward micro-batch are:

$$\text{No recomputation: } \frac{Llbh}{pt}(34 + \frac{5Hl}{h}) \text{ bytes} \tag{4}$$

Table 6. MFU numbers of the GPT-3 96B model, when the tensor parallelism degree is 8.

Model	Model Parallelism ID	tensor	pipeline	mb	Recompute scope	Megatron-LM MFU [%]	BPIPE MFU [%]
GPT-3 96B	(7)	8	4	1	attention	36.18	35.80
				2	attention	36.56	36.52
				4	layer	38.04	38.00
				8	layer	24.31	27.47

$$\text{Attention recomputation scope: } 34\frac{Llbh}{pt} \text{ bytes} \tag{5}$$

$$\text{Layer recomputation scope: } 2\frac{Llbh}{p} \text{ bytes} \tag{6}$$

Dividing Eq (4) to (6) by f, bandwidth requirements are derived as the following equations.

$$\text{No recomputation: } \frac{1}{10^9}\frac{Llbh}{ptf}(34+\frac{5Hl}{h}) \text{ GB/s} \tag{7}$$

$$\text{Attention recomputation scope: } \frac{34}{10^9}\frac{Llbh}{ptf} \text{ GB/s} \tag{8}$$

$$\text{Layer recomputation scope: } \frac{2}{10^9}\frac{Llbh}{pf} \text{ GB/s} \tag{9}$$

If a single node has 8 GPUs, two GPUs within the node where an evictor-acceptor pair resides should have larger bandwidth than Eq (7) to (9) when the tensor parallelism degree is 1, 2, or 4. For the tensor parallelism degree of 8, inter-node bandwidth divided by 8 should be larger than the derived bandwidths.

In our evaluation, the fastest configuration of BPIPE for the GPT-3 96B model uses 4-way tensor parallelism, 8-way pipeline parallelism, attention recomputation scope, and the micro-batch size of 2. Then the total bytes to transfer are 3.48 GB. Since each forward computation takes 143.37 ms in Table 3, the required bandwidth is 24.25 GB/s. As the forward computation time does not change by the recomputation scope, we can derive that NVLink is also sufficient for no recomputation in which the required bandwidth is 100.31 GB/s.

When the tensor parallelism degree is equal to the number of GPUs in a single node, BPIPE transfers activations across the node boundary. However, the inter-node transfer is feasible if (1) a cluster has a fast inter-node network such as InfiniBand and (2) recomputation is used for training. Moreover, we alleviate the bandwidth requirements with the following optimization. In general, backward computation takes twice as long as forward computation. Therefore, we allow the consecutive $Evict$-$Load$ to be performed over the timespan of consecutive $BACKWARD$-$FORWARD$ computation within the steady phase. For example, in Figure 4, evicting and loading the micro-batches whose indices are 3 and 1 can be completed within the sum of the time required for backward and forward computations. The bandwidth requirements are then relieved as the following equations.

$$\text{No recomputation: } \frac{1}{10^9}\frac{2Llbh}{3ptf}(34+\frac{5Hl}{h}) \text{ GB/s} \tag{10}$$

$$\text{Attention recomputation scope: } \frac{68}{3\cdot 10^9}\frac{Llbh}{ptf} \text{ GB/s} \tag{11}$$

$$\text{Layer recomputation scope: } \frac{4}{3\cdot 10^9}\frac{Llbh}{pf} \text{ GB/s} \tag{12}$$

Table 6 presents the MFU values when the tensor parallelism degree is 8 for the GPT-3 96B model. The results show that BPIPE is feasible even when the tensor parallelism degree is 8. Furthermore, when the micro-batch size is 8, the MFU value of BPIPE is higher than the MFU value of Megatron-LM. For such a large micro-batch size, we observed that PyTorch periodically vacates the cache allocator due to high memory pressure, resulting in performance degradation. BPIPE partially avoids such inefficiency with the activation balancing.

C. Raw MFU Numbers

Table 7. Raw MFU numbers of Figure 7. The numbers are the values of Megatron-LM, except those annotated with **(BPIPE)**. For those configurations, Megatron-LM fails to run due to the out-of-memory error.

Model	Model Parallelism ID	tensor	pipeline	mb	Recompute scope	MFU [%]
GPT-3 96B	(1)	1	16	1	layer	38.08
	(2)	2	8	1	layer	40.46
				2	layer	30.29
	(3)	4	4	1	attention	37.59
				2	layer	40.51
	(4)	8	2	1	attention	35.47
				2	layer	30.07
				4	layer	38.08
	(5)	2	16	1	attention	**48.78 (BPIPE)**
				2	layer	36.78
				4	layer	29.79
	(6)	4	8	1	attention	37.09
				2	attention	**50.64 (BPIPE)**
				4	layer	36.23
				8	layer	25.74
	(7)	8	4	1	attention	36.18
				2	attention	36.56
				4	layer	38.04
				8	layer	24.31
GPT-3 134B	(1)	2	12	1	layer	39.98
				2	layer	38.64
	(2)	4	6	1	attention	39.90
				2	layer	41.40
				4	layer	30.06
	(3)	8	3	1	attention	37.77
				2	layer	31.76
				4	layer	38.81
	(4)	4	12	1	attention	39.19
				2	attention	**52.06 (BPIPE)**
				4	layer	37.22
	(5)	8	6	1	attention	38.45
				2	attention	38.72
				4	layer	38.72
				8	layer	24.01