Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM

Deepak Narayanan[‡⋆], Mohammad Shoeybi[†], Jared Casper[†], Patrick LeGresley[†], Mostofa Patwary[†], Vijay Korthikanti[†], Dmitri Vainbrand[†], Prethvi Kashinkunti[†], Julie Bernauer[†], Bryan Catanzaro[†], Amar Phanishayee[*], Matei Zaharia[‡]
[†]*NVIDIA* [‡]*Stanford University* [*]*Microsoft Research*

ABSTRACT

Large language models have led to state-of-the-art accuracies across several tasks. However, training these models efficiently is challenging because: a) GPU memory capacity is limited, making it impossible to fit large models on even a multi-GPU server, and b) the number of compute operations required can result in unrealistically long training times. Consequently, new methods of model parallelism such as tensor and pipeline parallelism have been proposed. Unfortunately, naive usage of these methods leads to scaling issues at thousands of GPUs. In this paper, we show how tensor, pipeline, and data parallelism can be composed to scale to thousands of GPUs. We propose a novel interleaved pipelining schedule that can improve throughput by 10+% with memory footprint comparable to existing approaches. Our approach allows us to perform training iterations on a model with 1 trillion parameters at 502 petaFLOP/s on 3072 GPUs (per-GPU throughput of 52% of theoretical peak).

1 INTRODUCTION

Transformer-based language models [13, 27, 33–35, 42, 46] in Natural Language Processing (NLP) have driven rapid progress in recent years as computation at scale has become more available and datasets have become larger. Recent work [11, 40] has shown large language models to be effective zero- or few-shot learners, with high accuracy on many NLP tasks and datasets. These large language models have a number of exciting downstream applications such as client feedback summarization, automatic dialogue generation, semantic search, and code autocompletion [1, 4, 5]. As a result, the number of parameters in state-of-the-art NLP models have grown at an exponential rate (Figure 1). Training such models, however, is challenging for two reasons: (a) it is no longer possible to fit the parameters of these models in the main memory of even the largest GPU (NVIDIA recently released 80GB-A100 cards), and (b) even if we are able to fit the model in a single GPU (e.g., by swapping parameters between host and device memory [38]), the high number of compute operations required can result in unrealistically long training times (e.g., training GPT-3 with 175 billion parameters [11] would require approximately 288 years with a single V100 NVIDIA GPU). This calls for parallelism. Data-parallel scale-out usually works well, but suffers from two limitations: a) beyond a point, the per-GPU batch size becomes too small, reducing GPU utilization and increasing communication cost, and b) the maximum number of devices that can be used is the batch size, limiting the number of accelerators that can be used for training.

⋆Work done as an intern at NVIDIA.

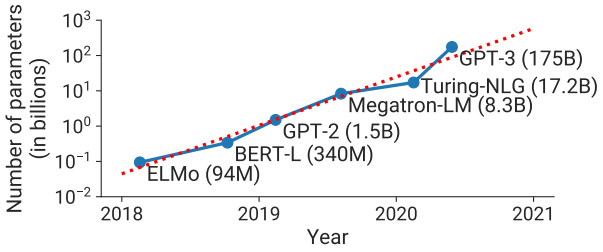


Figure 1: Trend of sizes of state-of-the-art Natural Language Processing (NLP) models with time. The number of floating-point operations to train these models is increasing at an exponential rate.

Various model parallelism techniques have been proposed to address these two challenges. For example, recent work [39, 40] has shown how tensor (intra-layer) model parallelism, where matrix multiplications within each transformer layer are split over multiple GPUs, can be used to overcome these limitations. Although this approach works well for models of sizes up to 20 billion parameters on NVIDIA DGX A100 servers (with 8 80GB-A100 GPUs), it breaks down for larger models. Larger models need to be split across multiple multi-GPU servers, which leads to two problems: (a) the all-reduce communication required for tensor parallelism needs to go through inter-server links, which are slower than the high-bandwidth NVLink [9] available within a multi-GPU server, and (b) a high degree of model parallelism can create small matrix multiplications (GEMMs), potentially decreasing GPU utilization.

Pipeline model parallelism [14, 20, 23, 29, 30, 45] is another technique to support the training of large models, where layers of a model are striped over multiple GPUs. A batch is split into smaller microbatches, and execution is pipelined across these microbatches. Layers can be assigned to workers in various ways, and various schedules for the forward and backward passes of inputs can be used. The layer assignment and scheduling strategy results in different performance tradeoffs. Regardless of schedule, to preserve strict optimizer semantics, optimizer steps need to be synchronized across devices, leading to a *pipeline flush* at the end of every batch, where microbatches are allowed to complete execution (and no new microbatches are injected). As much as 50% of time can be spent flushing the pipeline depending on the number of microbatches injected into the pipeline. The larger the ratio of number of microbatches to the pipeline size, the smaller the time spent in the pipeline flush. Therefore, to achieve high efficiency, a larger batch size is often necessary. In this work, we also introduce a new pipeline schedule that improves efficiency at small batch sizes.

Users can thus train their large models using various techniques, each with different tradeoffs. Moreover, these techniques can be

combined. However, combining these techniques leads to non-trivial interactions, which need to be reasoned through carefully for good performance. In this paper, we address the following question:

> *How should parallelism techniques be combined to maximize the training throughput of large models given a batch size while retaining strict optimizer semantics?*

In particular, we show how to combine pipeline, tensor, and data parallelism, a technique we call *PTD-P*, to train large language models with good computational performance (52% of peak device throughput) on 1000s of GPUs. Our method leverages the combination of pipeline parallelism across multi-GPU servers, tensor parallelism within a multi-GPU server, and data parallelism, to practically train models with a trillion parameters with graceful scaling in an optimized cluster environment with high-bandwidth links between GPUs on the same server and across servers. We can use similar ideas to train larger models as well, given more training resources. In our experiments, we demonstrate close to linear scaling to 3072 A100 GPUs, with an achieved end-to-end training throughput of 163 teraFLOP/s per GPU (including communication, data processing, and optimization), and an aggregate throughput of 502 petaFLOP/s, on a GPT model [11] with a trillion parameters using mixed precision. This throughput facilitates practical training times: we estimate end-to-end training of this model to take ~ 3 months. We believe this is the fastest training throughput achieved for this size of model: past systems [29, 40] cannot train such large models since they do not combine pipeline and tensor parallelism. We also compared to ZeRO [36], and found that our approach outperforms ZeRO-3 by 70% for models with 175 and 530 billion parameters due to less cross-node communication. These models are too large to fit on a multi-GPU server.

Achieving this throughput at scale required innovation and careful engineering along multiple axes: efficient kernel implementations that allowed most of the computation to be compute-bound as opposed to memory-bound, smart partitioning of computation graphs over the devices to reduce the number of bytes sent over network links while also limiting device idle periods, domain-specific communication optimization, and fast hardware (state-of-the-art GPUs and high-bandwidth links between GPUs on the same and different servers). We are hopeful that our open-sourced software (available at https://github.com/nvidia/megatron-lm) will enable other groups to train large NLP models efficiently at scale.

In addition, we studied the interaction between the various components affecting throughput, both empirically and analytically when possible. Based on these studies, we offer the following guiding principles on how to configure distributed training:

- Different forms of parallelism interact in non-trivial ways: the parallelization strategy has an impact on the amount of communication, the compute efficiency with which kernels are executed, as well as the idle time workers spend waiting for computation due to pipeline flushes (pipeline bubbles). For example, in our experiments, we found that sub-optimal combinations of tensor and pipeline model parallelism can lead to up to 2× lower throughput, even with high-bandwidth network links between servers; tensor model parallelism is effective within a multi-GPU server, but pipeline model parallelism must be used for larger models.
- The schedule used for pipeline parallelism has an impact on the amount of communication, the pipeline bubble size, and memory used to store activations. We propose a novel interleaved schedule that can improve throughput by as much as 10% compared to previously-proposed schedules [20, 30] with comparable memory footprint.
- Values of hyperparameters such as microbatch size have an impact on the memory footprint, the arithmetic efficiency of kernels executed on the worker, and the pipeline bubble size. In our experiments, the optimal value of the microbatch size is problem-dependent and can increase throughput by 15%.
- At scale, distributed training is communication-intensive. When training a trillion-parameter model on 3072 GPUs, our implementation used an effective bisection bandwidth of 892 GB/s for pipeline-parallel communication, and 13 TB/s for data-parallel communication. Using slower inter-node interconnects or more communication-intensive partitionings would hinder scaling performance.

We should note that we do not automatically explore the search space of parallelism strategies (such as FlexFlow [22], PipeDream [29], Tarnawski et al. [41], and DAPPLE [14]), but instead suggest heuristics (in §3) that we found work well in practice.

2 MODES OF PARALLELISM

In this section, we discuss the parallelism techniques that facilitate the *efficient* training of large models that do not fit in the memory of a single GPU. In this work, we combine pipeline model parallelism and tensor model parallelism (combination shown in Figure 2) with data parallelism. We call this PTD-P for short.

2.1 Data Parallelism

With data parallelism [25, 43], each worker has a copy of the full model, the input dataset is sharded, and workers aggregate their gradients periodically to ensure that all workers see a consistent version of the weights. For large models which do not fit on a single worker, data parallelism can be used on smaller model shards.

2.2 Pipeline Model Parallelism

With pipeline parallelism, the layers of a model are sharded across multiple devices. When used on models with the same transformer block repeated, each device can be assigned an equal number of transformer layers. We do not consider more asymmetric model architectures, where assignment of layers to pipeline stages is harder; we defer to related work [22, 29, 41] to solve this problem.

A batch is split into smaller microbatches; execution is then pipelined across microbatches. Pipelining schemes need to ensure that inputs see consistent weight versions across forward and backward passes for well-defined synchronous weight update semantics. Specifically, naive pipelining can lead to an input seeing weight updates in the backward pass not seen in the forward pass.

To retain strict optimizer semantics *exactly*, we introduce periodic pipeline flushes so that optimizer steps are synchronized across devices. At the start and end of every batch, devices are idle. We call this idle time the *pipeline bubble*, and want to make it as small as possible. Asynchronous and bounded-staleness approaches such as PipeMare, PipeDream, and PipeDream-2BW [23, 29, 30, 45] do

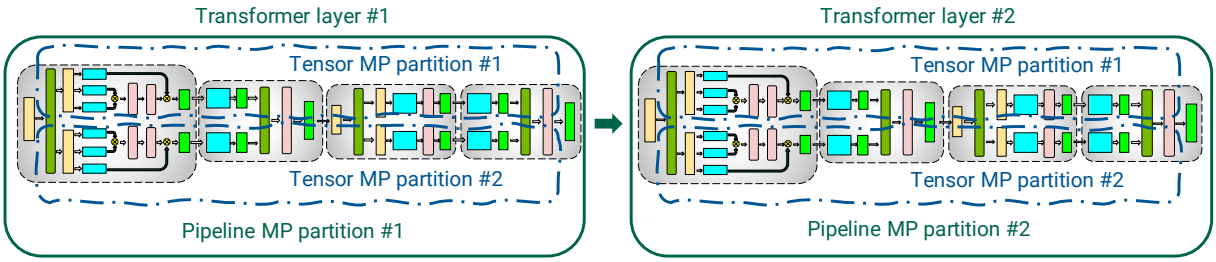


Figure 2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.

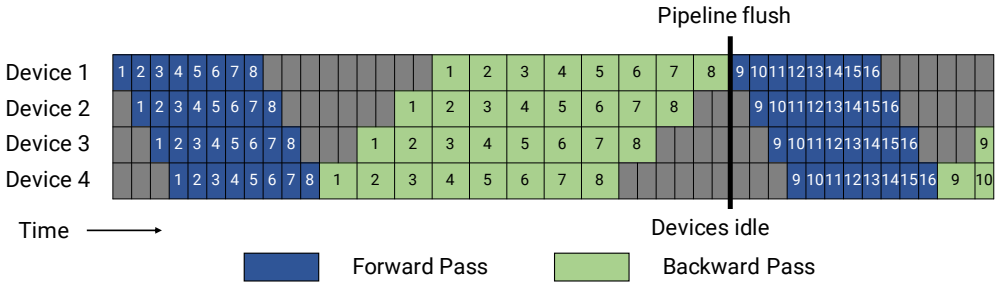


Figure 3: GPipe pipeline schedule with forward passes (blue) for all microbatches (represented by numbers) followed by backward passes (green). The gray area represents the pipeline bubble. For simplicity, we assume that the backward pass takes twice as long as the forward pass. The efficiency of the pipeline schedule does not depend on this factor. Each batch in this example consists of 8 microbatches, and the numbers in each blue or green box are unique identifiers given to the corresponding microbatch (in particular, the first batch consists of microbatches $1-8$, the second batch consists of microbatches $9-16$, and so on). The optimizer is stepped and weight parameters updated at the pipeline flush to ensure strict optimizer semantics, leading to idle devices and a pipeline bubble.

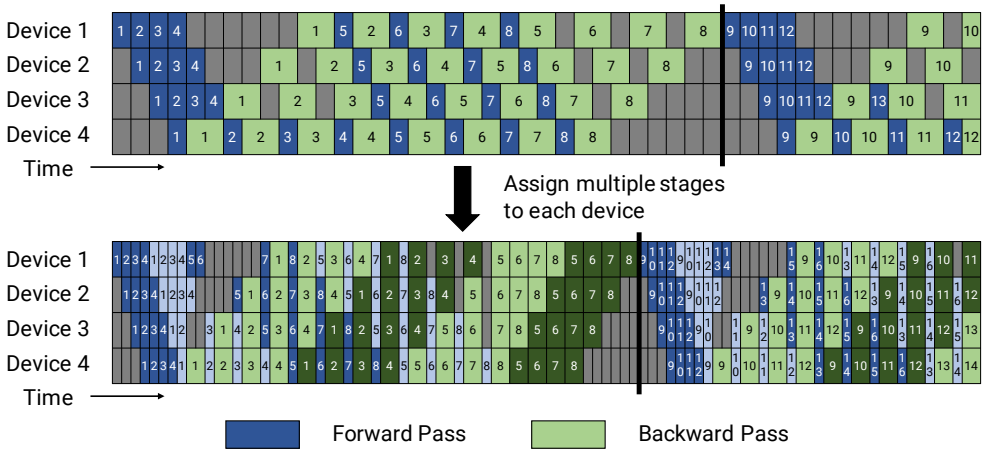
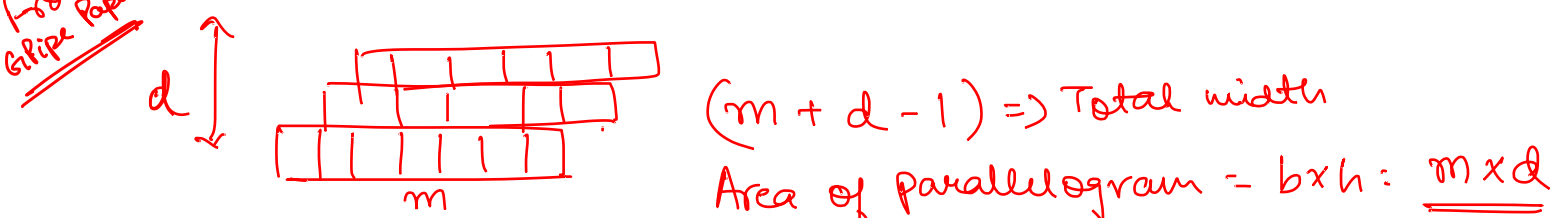


Figure 4: Default and interleaved 1F1B pipeline schedules. The top figure shows the default non-interleaved 1F1B schedule. The bottom figure shows the interleaved 1F1B schedule, where each device is assigned multiple chunks (in this case, 2). Dark colors show the first chunk and light colors show the second chunk. The size of the pipeline bubble is smaller (the pipeline flush happens sooner in the interleaved timeline).

away with flushes completely, but relax weight update semantics. We defer consideration of such schemes to future work.

There are several possible ways of scheduling forward and backward microbatches across devices; each approach offers different tradeoffs between pipeline bubble size, communication, and memory footprint. We discuss two such approaches in this section.

2.2.1 Default Schedule. GPipe [20] proposes a schedule where the forward passes for all microbatches in a batch are first executed, followed by backward passes for all microbatches (shown in Figure 3). We can quantify the size of GPipe's pipeline bubble (t_{pb}). We denote the number of microbatches in a batch as m, the number of pipeline stages (number of devices used for pipeline parallelism) as p, the ideal time per iteration as t_{id} (assuming perfect or ideal scaling), and the time to execute a single microbatch's forward and backward pass as t_f and t_b. In this schedule, the pipeline bubble consists of $p-1$ forward passes at the start of a batch, and $p-1$ backward passes at the end. The total amount of time spent in the



pipeline bubble is then $t_{pb} = (p-1) \cdot (t_f + t_b)$. The ideal processing time for the batch is $t_{id} = m \cdot (t_f + t_b)$. Therefore, the fraction of ideal computation time spent in the pipeline bubble is:

$$\text{Bubble time fraction (pipeline bubble size)} = \frac{t_{pb}}{t_{id}} = \frac{p-1}{m}.$$

For the bubble time fraction to be small, we thus need $m \gg p$. However, for such large m, this approach has a high memory footprint as it requires stashed intermediate activations (or just input activations for each pipeline stage when using activation recomputation) to be kept in memory for all m microbatches through the lifetime of a training iteration.

Instead, we use the PipeDream-Flush schedule [30]. In this schedule, we first enter a warm-up phase where workers perform differing numbers of forward passes as shown in Figure 4 (top). This schedule limits the number of in-flight microbatches (the number of microbatches for which the backward pass is outstanding and activations need to be maintained) to the depth of the pipeline, instead of the number of microbatches in a batch. After the warm-up phase, each worker then enters a steady state, where workers perform one forward pass followed by one backward pass (1F1B for short). Finally, at the end of a batch, we complete backward passes for all remaining in-flight microbatches. The time spent in the bubble is the same for this new schedule, but the number of outstanding forward passes is at most the number of pipeline stages for the PipeDream-Flush schedule. As a result, this schedule requires activations to be stashed for p or fewer microbatches (compared to m microbatches for the GPipe schedule). Consequently, when $m \gg p$, PipeDream-Flush is much more memory-efficient than GPipe.

2.2.2 Schedule with Interleaved Stages. To reduce the size of the pipeline bubble, each device can perform computation for multiple subsets of layers (called a model chunk), instead of a single contiguous set of layers. For example, if each device had 4 layers before (i.e., device 1 had layers $1-4$, device 2 had layers $5-8$, and so on), we could have each device perform computation for two model chunks (each with 2 layers), i.e., device 1 has layers 1, 2, 9, 10; device 2 has layers 3, 4, 11, 12; and so on. With this scheme, each device in the pipeline is assigned multiple pipeline stages (each pipeline stage has less computation compared to before).

As before, we can use an "all-forward, all-backward" version of this schedule, but this has a high memory footprint (proportional to m). Instead, we developed an interleaved schedule that adapts the memory-efficient 1F1B schedule from before. This new schedule is shown in Figure 4, and requires the number of microbatches in a batch to be an integer multiple of the degree of pipeline parallelism (number of devices in the pipeline). For example, with 4 devices, the number of microbatches in a batch must be a multiple of 4.

As shown in Figure 4, the pipeline flush for the same batch size happens sooner in the new schedule. If each device has v stages (or model chunks), then the forward and backward time for a microbatch for each stage or chunk will now be t_f/v and t_b/v. The pipeline bubble time thus reduces to $t_{pb}^{\text{int.}} = \frac{(p-1)\cdot(t_f+t_b)}{v}$ and the bubble time fraction is then:

$$\text{Bubble time fraction (pipeline bubble size)} = \frac{t_{pb}^{\text{int.}}}{t_{id}} = \frac{1}{v} \cdot \frac{p-1}{m}.$$

This means that the new schedule reduces the bubble time by v. This reduced pipeline bubble size, however, does not come for free: this schedule requires extra communication. Quantitatively, the amount of communication also increases by v. In the next section, we discuss how we can utilize the 8 InfiniBand networking cards in a multi-GPU server (e.g., a DGX A100 node) to reduce the impact of this extra communication.

2.3 Tensor Model Parallelism

With tensor model parallelism, individual layers of the model are partitioned over multiple devices. In this paper, we use the particular partitioning strategy used by Megatron [40] for transformer layers, the bedrock of language models. We can apply similar ideas to other types of models, like CNNs, as well. We briefly outline this strategy, illustrated in Figure 5, below.

A transformer layer consists of a self-attention block followed by a two-layer multi-layer perceptron (MLP). Further details of the transformer layer can be found in Vaswani et al [42].

The MLP block consists of two GEMMs and a GeLU non-linearity:

$$Y = \text{GeLU}(XA). \quad Z = \text{Dropout}(YB).$$

We can split A along its columns $A = [A_1, A_2]$. This partitioning allows the GeLU non-linearity to be independently applied to the output of each partitioned GEMM:

$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)].$$

This is advantageous as it removes the need for synchronization (needed if A is split along its rows since GeLU is non-linear).

The rows of the second weight matrix B can then be split along its rows to remove the need for any communication between the GEMMs (shown in Figure 5a), as shown below:

$$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \quad Y = [Y_1, Y_2].$$

The output of the second GEMM is then reduced across the GPUs before the dropout layer.

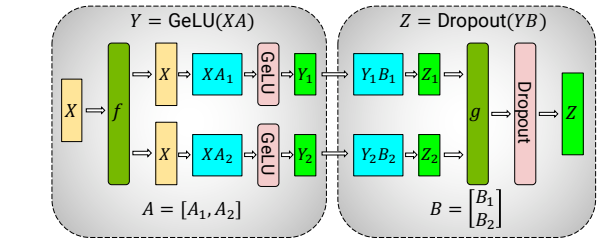
We exploit the inherent parallelism in the multi-head attention operation to partition the self-attention block (shown in Figure 5b). The key (K), query (Q), and value (V) matrices can be partitioned in a column-parallel fashion. The output linear layer can then directly operate on the partitioned output of the attention operation (weight matrix partitioned across rows).

This approach splits GEMMs in the MLP and self-attention blocks across GPUs while requiring only two all-reduce operations in the forward pass (g operator) and two all-reduces in the backward pass (f operator). We implemented f and g in a few lines of code.

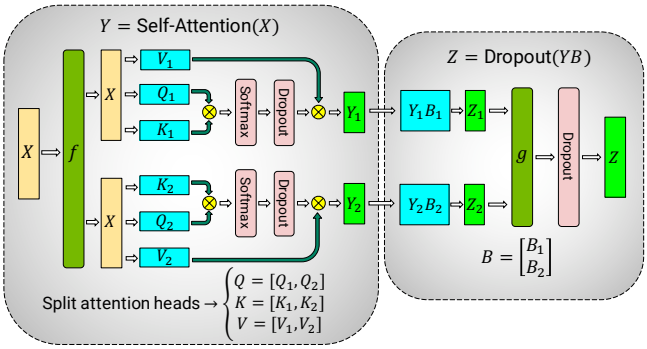
3 PERFORMANCE ANALYSIS OF PARALLELIZATION CONFIGURATIONS

In this section, we consider the performance implications of combining pipeline and tensor model parallelism with data parallelism. Given a fixed budget of GPUs and batch size, one can use different degrees of the parallelism types in PTD-P to train models; each dimension exposes tradeoffs between memory footprint, device utilization, and amount of communication.

We discuss these tradeoffs in the rest of this section, and then show empirical results in §5.4. We present analytical models where



(a) MLP.



(b) Self-Attention.

Figure 5: Blocks of transformer model partitioned with tensor model parallelism (figures borrowed from Megatron [40]). f and g are conjugate. f is the identity operator in the forward pass and all-reduce in the backward pass, while g is the reverse.

relevant for the pipeline bubble size. We qualitatively describe how communication time behaves and present cost models for amount of communication; however, we do not present direct cost models for communication time, which is harder to model for a hierarchical network topology where interconnects between GPUs on the same server have higher bandwidth than interconnects between servers. To the best of our knowledge, this is the first work to analyze the performance *interactions* of these parallelization dimensions.

3.1 Notation

We use the following notation in this section:

- (p, t, d): Parallelization dimensions. p for the pipeline-model-parallel size, t for the tensor-model-parallel size, and d for the data-parallel size.
- n: Number of GPUs. We require $p \cdot t \cdot d = n$.
- B: Global batch size (provided as input).
- b: Microbatch size.
- $m = \frac{1}{b} \cdot \frac{B}{d}$: Number of microbatches in a batch *per pipeline*.

3.2 Tensor and Pipeline Model Parallelism

Tensor and pipeline model parallelism can both be used to partition a model's parameters over multiple GPUs. As stated earlier, using pipeline parallelism with periodic flushes results in a pipeline bubble of size $(p-1)/m$. Let us assume that $d = 1$ (data-parallel size); consequently, $t \cdot p = n$. The pipeline bubble size in terms of t is:

$$\frac{p-1}{m} = \frac{n/t - 1}{m}.$$

As t increases, the pipeline bubble thus decreases for fixed B, b, and d ($m = B/(b \cdot d)$ is fixed as well).

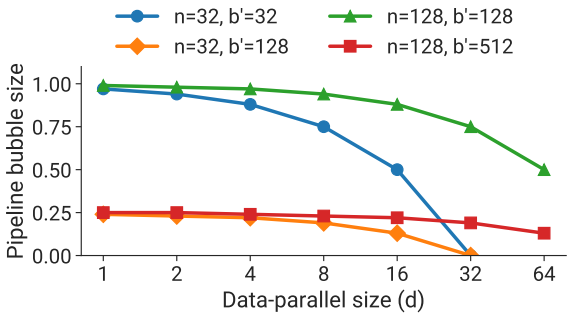


Figure 6: Fraction of time spent idling due to pipeline flush (pipeline bubble size) versus data-parallel size (d), for different numbers of GPUs (n) and ratio of batch size to microbatch size ($b' = B/b$).

The amount of communication performed between different GPUs is also affected by the values of p and t. Pipeline model parallelism features cheaper point-to-point communication. Tensor model parallelism, on the other hand, uses all-reduce communication (two all-reduce operations each in the forward and backward pass, see §2.3). With pipeline parallelism, the total amount of communication that needs to be performed between every pair of consecutive devices (for either the forward or backward pass) for each microbatch is bsh, where s is the sequence length and h is the hidden size. With tensor model parallelism, tensors of total size bsh need to be all-reduced among t model replicas twice each in the forward and backward pass for each layer, leading to a total communication of $8bsh\left(\frac{t-1}{t}\right)$ *per layer* per device for each microbatch. Each device typically has multiple layers; the total amount of tensor-parallel-communication per device for each microbatch is then $l^{\text{stage}} \cdot \left(8bsh\left(\frac{t-1}{t}\right)\right)$, where l^{stage} is the number of layers in a pipeline stage.

Consequently, we see that tensor model parallelism increases the amount of communication between devices. Thus, when t is larger than the number of GPUs in a single node, the overhead of performing tensor model parallelism across slower inter-node links can be impractical. We see these results empirically in §5.4.

Takeaway #1: When considering different forms of model parallelism, tensor model parallelism should generally be used up to degree g when using g-GPU servers, and then pipeline model parallelism can be used to scale up to larger models across servers.

3.3 Data and Model Parallelism

We also want to consider the interaction between data parallelism and the two types of model parallelism. In this section, we consider these interactions independently for simplicity.

3.3.1 Pipeline Model Parallelism. Let $t = 1$ (tensor-model-parallel size). The number of microbatches per pipeline is $m = B/(d \cdot b) = b'/d$, where $b' := B/b$. With total number of GPUs n, the number of pipeline stages is $p = n/(t \cdot d) = n/d$. The pipeline bubble size is:

$$\frac{p-1}{m} = \frac{n/d - 1}{b'/d} = \frac{n-d}{b'}.$$

bsh — split → All reduce
↓ (2) ↓ (2) = 4 bsh

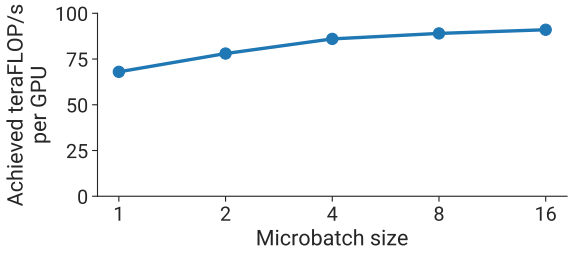


Figure 7: Per-GPU throughput versus microbatch size for a GPT model with a billion parameters (128 attention heads, hidden size of 4096, 4 transformer layers).

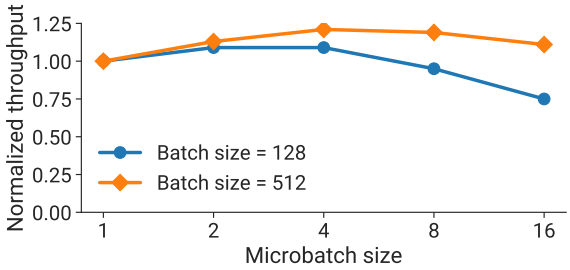


Figure 8: Behavior of normalized estimated throughput (time computed as $t = (b'/b + p - 1) \cdot (t_f(b) + t_b(b))$) with respect to the microbatch size b for the same GPT model from Figure 7.

As d becomes larger, $n - d$ becomes smaller, and thus the pipeline bubble becomes smaller. Figure 6 shows the behavior of the pipeline bubble size for various values of d, n, and b'. It might not be possible to increase d all the way to n for all models, since a model's full training memory footprint might be larger than the memory capacity of a single accelerator.

Overall throughput will thus increase if the all-reduce communication needed for data parallelism does not drastically increase with higher d, which should hold since the communication time for a ring-based implementation scales with $\frac{d-1}{d} = 1 - \frac{1}{d}$.

We can also analyze the impact of increasing the batch size B. For a given parallel configuration, as the batch size B increases, $b' = B/b$ increases, $(n-d)/b'$ decreases, consequently increasing throughput. All-reduce communication required by data parallelism also becomes more infrequent, further increasing throughput.

3.3.2 Data and Tensor Model Parallelism. With tensor model parallelism, all-reduce communication needs to be performed for *every* microbatch. This can be expensive across multi-GPU servers. On the other hand, data parallelism only needs to perform expensive all-reduce communication *once per batch*. Moreover, with tensor model parallelism, each model-parallel rank performs a subset of the computation in each model layer, and thus for insufficiently-large layers, modern GPUs might not perform these sub-matrix computations with peak efficiency.

Takeaway #2: When using data and model parallelism, a total model-parallel size of $M = t \cdot p$ should be used so that the model's parameters and intermediate metadata fit in GPU memory; data parallelism can be used to scale up training to more GPUs.

3.4 Microbatch Size

The choice of the microbatch size b also affects model-training throughput. For example, we see in Figure 7 that per-GPU throughput increases by up to 1.3× with a larger microbatch size on a single GPU. We now want to determine the optimal microbatch size b given a parallel configuration (p, t, d) and batch size B. The amount of data-parallel communication will be the same regardless of the microbatch size. Given functions $t_f(b)$ and $t_b(b)$ that map the microbatch size to the forward and backward computation times for a single microbatch, the total time spent computing a batch, ignoring communication cost, is (as before, define b' as B/d):

$$(b'/b + p - 1) \cdot \left(t_f(b) + t_b(b)\right). \quad (1)$$

The microbatch size thus affects both the arithmetic intensity of operations as well as the pipeline bubble size (by affecting m). Figure 8 shows estimated throughput (equation (1) used to estimate processing time) for a GPT model with a billion parameters and $(p, t) = (8, 8)$. The optimal b for both batch sizes is 4.

Takeaway #3: The optimal microbatch size b depends on the throughput and memory footprint characteristics of the model, as well as the pipeline depth p, data-parallel size d, and batch size B.

3.5 Activation Recomputation

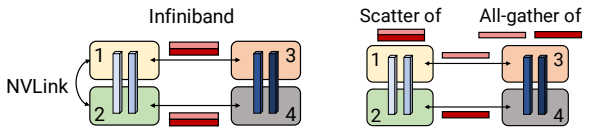
Activation recomputation [12, 18, 20, 21] is an optional technique that trades off an increase in the number of compute operations performed for additional memory footprint, by running the forward pass a second time just before the backward pass (and stashing only the input activations for a given pipeline stage, as opposed to the entire set of intermediate activations, which is much larger). Activation recomputation is required to train reasonably large models with pipeline parallelism to keep memory footprint acceptably low. Previous work like PipeDream-2BW [30] has looked at the performance ramifications of activation recomputation.

The number of activation checkpoints does not impact throughput, but impacts memory footprint. Let A^{input} be the size of the input activations of a layer, and $A^{\text{intermediate}}$ be the size of intermediate activations per layer. If a model stage has l layers, and if c is the number of checkpoints, the total memory footprint is going to be $c \cdot A^{\text{input}} + l/c \cdot A^{\text{intermediate}}$. The minimum value of this function is obtained when $c = \sqrt{l \cdot (A^{\text{intermediate}}/A^{\text{input}})}$. In practice, we measure $A^{\text{intermediate}}$ empirically. For most cases, checkpointing every 1 or 2 transformer layers is optimal.

Other techniques such as activation partitioning [36] can also be used in conjunction with tensor model parallelsim to reduce the memory footprint due to activations further.

4 IMPLEMENTATION

We implemented PTD-P as an extension to the Megatron-LM codebase. Our implementation is built using PyTorch [32]. We use NCCL [7] for communication between devices. To obtain good performance, we implemented optimizations targeting both communication and computation, which we outline below.



(a) W/o scatter/gather optimization. (b) With scatter/gather optimization.

Figure 9: Scatter/gather communication optimization. Light blue blocks are layers in the first pipeline stage, and dark blue blocks are layers in the second pipeline stage. Without the scatter/gather optimization, the same tensor is sent redundantly over inter-node InfiniBand links. Instead, at the sender, we can scatter the tensor into smaller chunks, reducing the sizes of tensors sent over InfiniBand links. The final tensor can then be rematerialized at the receiver using a gather operation.

4.1 Communication Optimizations

When using pipeline parallelism, we want to send and receive tensors in the forward and backward direction in parallel. Each DGX A100 is equipped with 8 InfiniBand (IB) networking cards. Unfortunately, sends and receives are point-to-point, and only happen between a pair of GPUs on two servers, making it hard to leverage all 8 cards for a single communication call within the pipeline.

However, we can leverage the fact that we use both tensor model parallelism and pipeline model parallelism to reduce the overhead of cross-node communication. In particular, we note that the output of each transformer layer is replicated (after g in MLP block, see Figure 5a) across the tensor-parallel ranks. As a result, ranks in two consecutive pipeline stages that are performing tensor model parallelism send and receive the exact same set of tensors (Figure 9a).

For large enough models, we use a tensor-model-parallel size of 8. This means we are sending the same set of tensors 8 times between corresponding GPUs on adjacent multi-GPU servers. To reduce this redundancy, we can instead split the tensor on the send side into equal-sized chunks, and then only send one chunk to the corresponding rank on the next node using the rank's own InfiniBand card (e.g., rank 1 sends to rank 3 and rank 2 sends to rank 4 in Figure 9). With 8 tensor-model-parallel ranks, each chunk would be one-eighth smaller. Then, on the receive side, we can perform an all-gather over NVLink, which is much faster than the InfiniBand interconnect, to re-materialize the full tensor. This is shown in Figure 9b. We call this the *scatter/gather communication optimization*. This optimization helps better leverage the multiple IB cards on the DGX A100 servers, and makes more communication-intensive schedules such as the interleaved one feasible.

Quantitatively, with the scatter-gather communication optimization, the total amount of communication that needs to be performed between every pair of consecutive stages is reduced to $\frac{bsh}{t}$, where t is the tensor-model-parallel size, s is the sequence length, and h is the hidden size ($t = 8$ in our experiments).

4.2 Computation Optimizations

We implemented three model-specific optimizations to the computation graph to attain high performance. First, we changed the data layout in the transformer layer to avoid memory-intensive transpose operations, and to enable the use of strided batched GEMM kernels. Specifically, we changed the data layout from $[b, s, a, h]$ to $[s, b, a, h]$, where b, s, a, and h are batch, sequence, attention-head, and hidden-size dimensions, respectively. Second, we generated fused kernels for a sequence of element-wise operations (bias + GeLU and bias + dropout + add) using PyTorch JIT [10]. Third, we created two custom kernels to enable the fusion of scale, mask, and softmax (reduction) operations: one to support general masking (used in models such as BERT) and another to support implicit causal masking (used in auto-regressive models such as GPT). We quantify the effect of these optimizations in the next section.

5 EVALUATION

In this section, we seek to answer the following questions:

- How well does PTD-P perform? Does it result in realistic end-to-end training times?
- How well does pipeline parallelism scale for a given model and batch size? How much impact does the interleaved schedule have on performance?
- How do different parallelization dimensions interact with each other? What is the impact of hyperparameters such as microbatch size?
- What is the impact of the scatter-gather communication optimization? What types of limits do we put on hardware when running training iterations at scale?

All of our results are run with mixed precision on the Selene supercomputer [8]. Each cluster node has 8 NVIDIA 80-GB A100 GPUs [6], connected to each other by NVLink and NVSwitch [9]. Each node has eight NVIDIA Mellanox 200Gbps HDR Infiniband HCAs for application communication, with an additional two HCAs per node for dedicated storage. The nodes are connected in a three-level (leaf, spine, core) fat-tree topology with 850 switches. This topology allows efficient all-reduce communication (dominant communication pattern in deep learning training). The cluster uses an all-NVME shared parallel filesystem for high-performance data access and storage. The peak device throughput of an A100 GPU with 16-bit precision is 312 teraFLOP/s. For most of our results, we report throughput per GPU. Aggregate throughput can be computed by multiplying with the number of GPUs used.

For our experiments, we use GPT models of appropriate sizes. In particular, for any given microbenchmark, the model needs to fit on the number of model-parallel GPUs used in the experiment. We use standard model architectures such as GPT-3 [11] when appropriate.

5.1 End-to-End Performance

We consider the end-to-end performance of our system on GPT models ranging from a billion to a trillion parameters, using tensor, pipeline, and data parallelism (degrees picked using heuristics described in §3). In particular, we use the interleaved pipeline schedule with the scatter/gather optimization enabled. All models use a vocabulary size (denoted by V) of 51,200 (multiple of 1024) and a sequence length (s) of 2048. We vary hidden size (h), number of attention heads, and number of layers (l). The number of parameters in a model, P, can be computed as:

$$P = 12lh^2 \left(1 + \frac{13}{12h} + \frac{V+s}{12lh}\right). \tag{2}$$

Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7	24	2304	24	1	1	32	512	137	44%	4.4
3.6	32	3072	30	2	1	64	512	138	44%	8.8
7.5	32	4096	36	4	1	128	512	142	46%	18.2
18.4	48	6144	40	8	1	256	1024	135	43%	34.6
39.1	64	8192	48	8	2	512	1536	138	44%	70.8
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0

Table 1: Weak-scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

As the model size increases, we also increase the batch size (B) and the number of GPUs (n). The majority of floating-point operations in the model are performed in the matrix multiplications (GEMMs) in the transformer and logit layers. Considering just these GEMMs, the number of FLOPs per iteration is (more details in the Appendix):

$$F = 96Bslh^2\left(1 + \frac{s}{6h} + \frac{V}{16lh}\right). \quad (3)$$

This is a lower bound for the true FLOP count but should be close to the actual value. We count a FLOP as a floating-point operation regardless of precision. We also note that equation (3) assumes activation recomputation and takes into account the floating-point operations associated with the extra forward pass.

Table 1 shows the model configurations along with the achieved FLOP/s (both per GPU and aggregate over all GPUs). We see super-linear scaling to 3072 A100 GPUs (384 DGX A100 nodes), since GPU utilization improves as the models get larger (larger matrix multiplications) without significant increase in the communication time relative to computation time. Note that throughput is measured for end-to-end training, i.e., includes all operations including data loading, optimizer steps, communication, and logging. We achieve 52% of peak device throughput for the largest model, and 44% of peak device throughput for the smallest model.

Training Time Estimates. Given these throughputs, we can also estimate the total amount of time needed for end-to-end training on T tokens. Training requires $I = T/(B \cdot s)$ iterations. Using the value of F from equation (3) and empirical end-to-end throughputs from Table 1 (denoted by X), we can estimate total training time. We note that for the configurations in Table 1, we have $6h \gg s$, $16lh \gg (V + s)$, and $12lh \gg V$. Combining these observations with equations (2) and (3), we arrive at

$$\text{End-to-end training time} \approx \frac{8TP}{nX}. \quad (4)$$

Let us consider the GPT-3 model with P =175 billion parameters as an example. This model was trained on T = 300 billion tokens. On n = 1024 A100 GPUs using batch size 1536, we achieve X = 140 teraFLOP/s per GPU. As a result, the time required to train this model is 34 days. For the 1 trillion parameter model, we assume that 450 billion tokens are needed for end-to-end training. With 3072 A100 GPUs, we can achieve a per-GPU throughput of 163 teraFLOP/s, and end-to-end training time of 84 days. We believe these training times (using a reasonable number of GPUs) are practical.

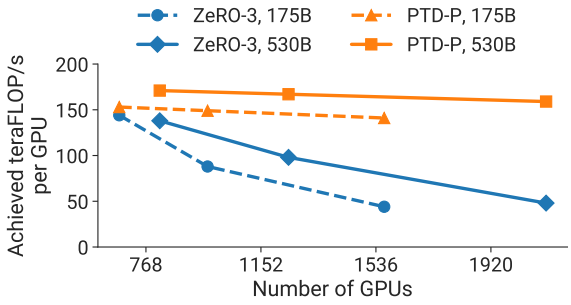


Figure 10: Throughput per GPU of PTD-P and ZeRO-3 for two different GPT models (the 175B GPT-3 model is shown with dotted lines, and the 530B model is shown with solid lines). Global batch sizes are fixed and ZeRO-3 is used without any model parallelism.

5.2 Comparison to ZeRO-3

We compare PTD-P to ZeRO-3 [36, 37] in Table 2 and Figure 10 for the standard GPT-3 model architecture, as well as the 530-billion-parameter model from Table 1. The results provide a point of comparison to a method that does not use model parallelism. We integrated ZeRO into our codebase using the DeepSpeed Python library [3]. We keep the global batch size the same as we increase the number of GPUs. With fewer GPUs and a microbatch size of 4, PTD-P results in 6% and 24% higher throughput for the 175- and 530-billion-parameter models respectively. As we increase the number of GPUs, PTD-P scales more gracefully than ZeRO-3 in isolation (see Figure 10). For example, by doubling the number of GPUs (keeping the batch size the same), PTD-P outperforms ZeRO-3 by 70% for both models due to less cross-node communication. We note that we have only considered ZeRO-3 without tensor parallelism. ZeRO-3 can be combined with model parallelism to potentially improve its scaling behavior.

5.3 Pipeline Parallelism

We now evaluate the weak-scaling performance of pipeline parallelism in isolation, and also compare the performance of the non-interleaved schedule to the interleaved schedule.

5.3.1 Weak Scaling. We evaluate the scaling of the default non-interleaved pipeline-parallel schedule using a weak scaling setup, a GPT model with 128 attention heads and a hidden size of 20480,

Scheme	Number of parameters (billion)	Model-parallel size	Batch size	Number of GPUs	Microbatch size	Achieved teraFlOP/s per GPU	Training time for 300B tokens (days)
ZeRO-3 without Model Parallelism	174.6	1	1536	384	4	144	90
				768	2	88	74
				1536	1	44	74
	529.6	1	2560*	640	4	138	169
			2240	1120	2	98	137
				2240	1	48	140
PTD Parallelism	174.6	96	1536	384	1	153	84
				768	1	149	43
				1536	1	141	23
	529.6	280	2240	560	1	171	156
				1120	1	167	80
				2240	1	159	42

Table 2: Comparison of PTD Parallelism to ZeRO-3 (without model paralllelism). The 530-billion-parameter GPT model did not fit on 560 GPUs when using a microbatch size of 4 with ZeRO-3, so we increased the number of GPUs used to 640 and global batch size to 2560 to provide a throughput estimate (relevant row marked in table with a *).

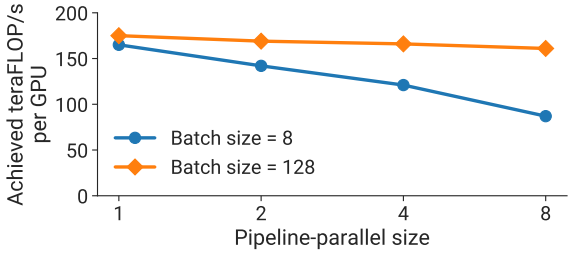


Figure 11: Throughput per GPU of pipeline parallelism using two different batch sizes in a weak-scaling experiment setup (model size increases with the pipeline-parallel size).

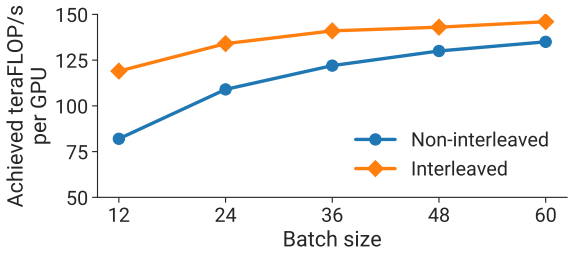


Figure 12: Throughput per GPU of interleaved and non-interleaved schedules for a GPT model (175 billion parameters) on 96 GPUs.

and a microbatch size of 1. As we increase the number of pipeline stages, we also increase the size of the model by proportionally increasing the number of layers in the model, e.g., with a pipeline-parallel size of 1, we use a model with 3 transformer layers and 15 billion parameters, and with a pipeline-parallel size of 8, we use a model with 24 transformer layers and 121 billion parameters. We use a tensor-parallel size of 8 for all configurations, and vary the total number of A100 GPUs used from 8 to 64. Figure 11 shows throughput per GPU for two different batch sizes to illustrate the impact of the pipeline bubble, which behaves as $\frac{p-1}{m}$ (§2.2.1). As expected, the higher batch size scales better since the pipeline bubble is amortized over more microbatches.

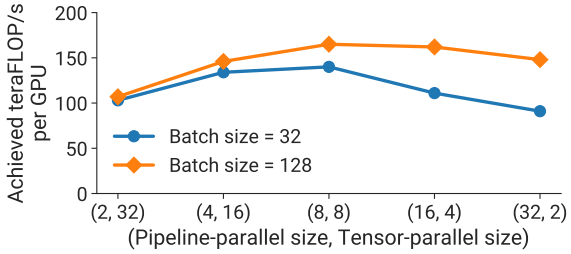


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

5.3.2 Interleaved versus Non-Interleaved Schedule. Figure 12 shows the per-GPU-throughput for interleaved and non-interleaved schedules on the GPT-3 [11] model with 175 billion parameters (96 layers, 96 attention heads, hidden size of 12288). The interleaved schedule with the scatter/gather communication optimization has higher computational performance than the non-interleaved (default) schedule. This gap closes as the batch size increases due to two reasons: (a) as the batch size increases, the bubble size in the default schedule decreases, and (b) the amount of point-to-point communication within the pipeline is proportional to the batch size, and consequently the non-interleaved schedule catches up as the amount of communication increases (the interleaved schedule features more communication per sample). Without the scatter/gather optimization, the default schedule performs better than the interleaved schedule at larger batch sizes (not shown).

5.4 Comparison of Parallel Configurations

In this sub-section, we show the various tradeoffs associated with combining different parallelization dimensions. In particular, we show the performance for parallel configurations using the same number of GPUs for a given model and multiple batch sizes.

5.4.1 Tensor versus Pipeline Parallelism. We evaluate the impact of pipeline and tensor model parallelism on performance for a given model and batch size. The empirical results in Figure 13 show the

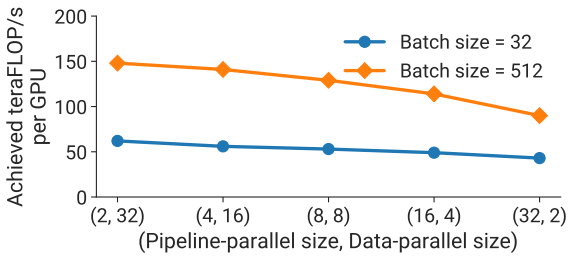


Figure 14: Throughput per GPU of various parallel configurations that combine data and pipeline model parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

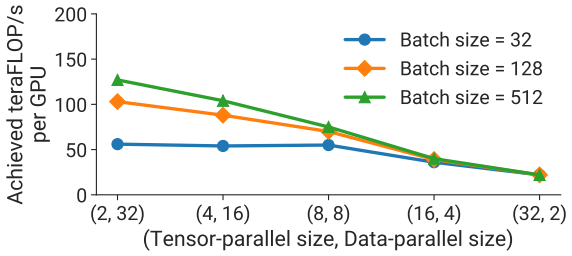


Figure 15: Throughput per GPU of various parallel configurations that combine data and tensor model parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

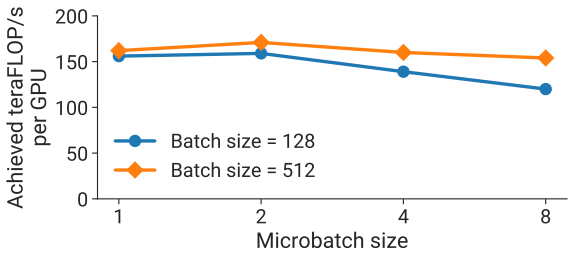


Figure 16: Throughput per GPU of a $(t, p) = (8, 8)$ parallel configuration for different microbatch sizes on a GPT model with 91 billion parameters, for two different batch sizes using 64 A100 GPUs.

importance of using both tensor and pipeline model parallelism in conjunction to train a 161-billion-parameter GPT model (32 transformer layers to support pipeline-parallel size of 32, 128 attention heads, hidden size of 20480) with low communication overhead and high compute resource utilization. We observe that tensor model parallelism is best within a node (DGX A100 server) due to its expensive all-reduce communication. Pipeline model parallelism, on the other hand, uses much cheaper point-to-point communication that can be performed across nodes without bottlenecking the entire computation. However, with pipeline parallelism, significant time can be spent in the pipeline bubble: the total number of pipeline stages should thus be limited so that the number of microbatches in the pipeline is a reasonable multiple of the number of pipeline stages. Consequently, we see peak performance when the tensor-parallel size is equal to the number of GPUs in a single node (8 with DGX A100 nodes). This result indicates that neither tensor model parallelism (used by Megatron [40]) nor pipeline model parallelism (used by PipeDream [30] and others) in isolation can match the performance of using both techniques in conjunction.

5.4.2 Pipeline versus Data Parallelism. We evaluate the impact of data and pipeline model parallelism on performance for a GPT model with 5.9 billion parameters (32 transformer layers, 32 attention heads, hidden size of 3840) in Figure 14. We use a smaller model than before since we want to show performance for models that fit when the model-parallel size is only 2. For simplicity, we keep the microbatch size equal to 1 in these experiments. We see that for each batch size, the throughput decreases as the pipeline-parallel size increases, matching our analytical model from §3.3. Pipeline model parallelism should be used primarily to support the training of large models that do not fit on a single worker, and data parallelism should be used to scale up training.

5.4.3 Tensor versus Data Parallelism. We also evaluate the impact of data and tensor model parallelism on performance for the same GPT model with 5.9 billion parameters in Figure 15 (smaller model used for same reason as above). As before, we keep the microbatch size equal to 1 initially. With larger batch sizes and a microbatch size of 1, data-parallel communication is infrequent; the all-to-all communication required in tensor model parallelism needs to be performed for *every* microbatch in a batch. This all-to-all communication with tensor model parallelism dominates end-to-end training time, especially when communication needs to be performed across multi-GPU nodes. Additionally, as the tensor-model-parallel size increases, we perform smaller matrix multiplications on every GPU, decreasing utilization on each GPU.

We should note that although data parallelism can lead to efficient scaling, we cannot use data parallelism in isolation for very large models with a limited training batch size because of a) insufficient memory capacity, and b) scaling limitations of data parallelism (e.g., GPT-3 was trained to convergence with a batch size of 1536. Data parallelism thus supports parallelization to only 1536 GPUs; however, roughly 10,000 GPUs were used to train this model in a reasonable amount of time).

5.5 Microbatch Size

We evaluate the impact of the microbatch size on the performance of parallel configurations that combine pipeline and tensor model parallelism in Figure 16 for a model with 91 billion parameters ($(t, p) = (8, 8)$). We see that the best microbatch size is 2 for this model; the optimal microbatch size is different for other models (not shown in Figure) and *model-dependent*. For a given batch size, increasing the microbatch size decreases the number of microbatches in the pipeline (m), leading to a larger pipeline bubble; however, increasing the microbatch size can also improve GPU utilization by increasing the arithmetic intensity of executed kernels. These two factors are at odds with each other, which makes the choice of optimal microbatch size challenging. Our analytical model from §3.3 reasonably approximates true performance, and can be used as a proxy to determine how to pick this hyperparameter value for various training configurations and models.

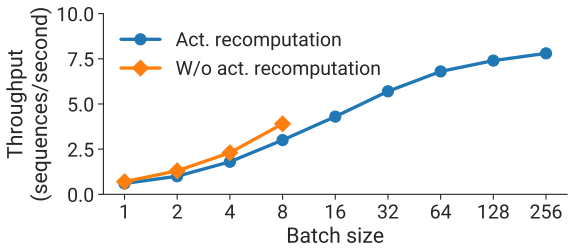


Figure 17: Throughput (in sequences per second) with and without activation recomputation for a GPT model with 145 billion parameters using 128 A100 GPUs ($(t, p) = (8, 16)$).

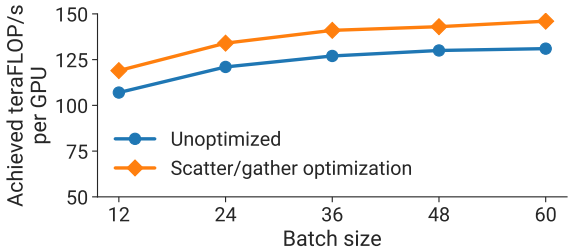


Figure 18: Throughput per GPU with and without the scatter/gather optimization for a GPT model with 175 billion parameters using 96 A100 GPUs and the interleaved schedule.

5.6 Activation Recomputation

Figure 17 shows throughput with and without activation recomputation for a GPT model with 145 billion parameters (80 transformer layers, 96 attention heads, hidden size of 12288) using 128 A100 GPUs, $(t, p) = (8, 16)$, and a range of batch sizes. For small batch sizes, activation recomputation leads to up to 33% lower throughput (in sequences per second) due to the extra forward pass that needs to be executed during the backward pass. However, activation recomputation is needed to support larger batch sizes. Throughput at large batch sizes with activation recomputation is up to 2× higher than the best throughput achieved without activation recomputation (for a smaller batch size) due to a smaller pipeline bubble.

5.7 Scatter-Gather Optimization

Figure 18 shows per-GPU-throughput with and without (unoptimized) the scatter/gather communication optimization for the GPT-3 model with 175 billion parameters. We see an improvement of up to 11% in throughput for communication-intensive schedules (large batch size with interleaving) by reducing the amount of communication over cross-node links.

5.8 Fused Operators

We also evaluate the performance impact of operator fusion described in §4.2. For the GPT-3 model (175 billion parameters), throughput increased by 19% with fusion (113 teraFLOP/s per GPU to 135 teraFLOP/s per GPU). For the larger GPT model with 530 billion parameters (model configuration in Figure 1), throughput increased by 11% (133 teraFLOP/s per GPU to 148 teraFLOP/s per GPU).

5.9 Inter-Node Communication Bandwidth

Our strong results are a byproduct of using an optimized software and hardware stack *together*. In particular, we take advantage of the high-bandwidth communication links between GPUs on the same server and across servers. On the trillion-parameter model with 3072 GPUs, we observed that the effective bisection bandwidth of point-to-point communication among pipeline stages is 892 GB/s, while the effective bisection bandwidth of all-reduce operations among data-parallel replicas is 12.9 TB/s. A less-optimized partitioning of operators across devices would lead to more inter-node communication, hampering scaling performance.

5.10 Checkpoint Loading and Saving

An important practical consideration for the training of large models is loading and saving model checkpoints, which are especially large for the models considered in this paper. For example, the trillion-parameter model has a checkpoint of size 13.8 terabytes. The initial load of checkpoints for the trillion-parameter model by all 384 nodes (3072 GPUs) reaches a peak read bandwidth of 1TB/s, the maximum read throughput possible from the parallel filesystem. Checkpoint saves reach 40% of peak write bandwidth (273 GB/s).

6 RELATED WORK

In this section, we discuss other techniques to train models at scale.

Parallelism for Large Models. Pipeline model parallelism is a common technique used to train large models. Pipeline parallelism comes in a few flavors: the mode discussed in this paper uses flushes to ensure *strict* optimizer semantics. TeraPipe [26] exposes fine-grained pipeline parallelism across tokens in a single training sequence for auto-regressive models like GPT. PipeTransformer [19] elastically adjusts the degree of pipelining and data parallelism by freezing layers with "stable" weights, and instead dedicates resources to train the remaining "active" layers. HetPipe [31] uses a combination of pipeline and data parallelism on a set of heterogeneous accelerators. Pipeline parallelism can also be implemented with relaxed semantics: PipeDream-2BW [30] maintains two weight versions and guarantees 1-stale weight updates without expensive flushes, while PipeMare [45] and Kosson et al. [23] use asynchoronous pipeline parallelism. These techniques have improved throughput compared to the techniques with pipeline flushes considered in this paper, but potentially at the cost of convergence rate or final accuracy. Moreover, pipeline parallelism in isolation can still only scale to a number of devices equal to the number of layers in the model, which is limiting for certain model architectures.

PipeDream [29] combined pipeline parallelism and data parallelism in a principled way to reduce cross-device communication. DeepSpeed [2] combined pipeline parallelism with tensor and data parallelism to train models with up to a trillion parameters, but with lower throughput than what was shown in this paper (52% vs. 36% of peak) for a few reasons: operator fusion to keep most of the operator graph compute-bound, a more-efficient pipeline parallelism schedule to minimize the pipeline bubble size, fast hardware (A100 vs. V100 GPUs and high-bandwidth links between GPUs on the same and different servers), and scaling to more GPUs. We want to emphasize that this higher throughput makes estimated

training times much more practical (about 3 months); an aggregate throughput of 37.6 petaFLOP/s would take about 40 months to train an equivalently-sized model. We can scale to larger models as well, but would need more GPUs to keep training time practical.

Mesh-TensorFlow [39] proposes a language for easily specifying parallelization strategies that combine data and model parallelism. Switch Transformers [15] used Mesh-Tensorflow to train a sparsely activated expert-based model with 1.6 trillion parameters, with improved pre-training speed over the T5-11B model [35].

Sharded Data Parallelism. As part of performance optimizations for MLPerf 0.6 [28], sharded data parallelism [24, 44], where optimizer state is sharded over data-parallel workers, was introduced. This method has two advantages: (a) it does not introduce extra communication over vanilla data parallelism, and (b) it divides the optimizer's computation and memory cost across the data-parallel partitions. ZeRO [36, 37] extends this idea: weight parameters and gradients are sharded across data-parallel workers as well, and workers fetch relevant state from their "owning" workers before performing computations. This adds additional communication, which can be partially hidden by carefully overlapping computation and communication. However, this can become harder if tensor parallelism is not used or the batch size is not large enough to hide the extra communication overhead (Figure 10). ZeRO-Infinity [37] uses NVMe to efficiently swap parameters, enabling the training of very large models on a small number of GPUs. We note that using a small number of GPUs for training a very large model results in unrealistic training times (e.g., thousands of years to converge).

Automatic Partitioning. FlexFlow [22], PipeDream [29], DAPPLE [14], and Tarnawski et al. [41] all auto-partition model training graphs over multiple devices with the help of cost models. However, each of these do not consider *all* the parallelism dimensions considered in this paper: pipeline and tensor model parallelism, data parallelism, microbatch size, and the effect of memory-savings optimizations like activation recomputation on the training of models larger than the memory capacity of an accelerator. These added dimensions increase the search space that needs to be explored. Gholami et al. [16] show how communication costs for combinations of data and model parallelism can be modeled.

HPC for Model Training. Goyal et al. [17] and You et al. [47] both demonstrate the use of High Performance Computing techniques to train highly-accurate ImageNet models in minutes. However, the image classification models considered fit comfortably on a single accelerator, rendering model parallelism unnecessary, support very large batch sizes (> 32k) that allow scaling data parallelism to large worker counts with infrequent communication, and are composed of compact convolutional layers that are inherently amenable to data-parallel communication.

7 DISCUSSION AND CONCLUSION

In this paper, we have shown how PTD-P (inter-node pipeline parallelism, intra-node tensor parallelism, and data parallelism) can be composed to achieve high aggregate throughput (502 petaFLOP/s) while training large models with a trillion parameters. This facilitates end-to-end training in reasonable times (estimated time of around 3 months for a trillion-parameter model). We discussed the various tradeoffs associated with each of these types of parallelism, and how the interactions between them need to be considered carefully when combined.

Even though the implementation and evaluation in this paper is GPU-centric, many of these ideas translate to other types of accelerators as well. Concretely, the following are ideas that are accelerator-agnostic: a) the idea of smartly partitioning the model training graph to minimize the amount of communication while still keeping devices active, b) minimizing the number of memory-bound kernels with operator fusion and careful data layout, c) other domain-specific optimizations (e.g., scatter-gather optimization).

ACKNOWLEDGEMENTS

We thank the anonymous reviewers, Seonmyeong Bak, Keshav Santhanam, Trevor Gale, Dimitrios Vytiniotis, and Siddharth Karamcheti for their help and feedback that improved this work. This research was supported in part by NSF Graduate Research Fellowship grant DGE-1656518 and NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors alone.

APPENDIX: FLOATING-POINT OPERATIONS

In this section, we describe how we calculate the number of floating-point operations (FLOPs) in a model. We consider a language model with l transformer layers, hidden size h, sequence length s, vocabulary size V, and training batch size B.

A $A_{m\times k} \times X_{k\times n}$ matrix multiplication requires $2m \times k \times n$ FLOPs (factor of 2 needed to account for multiplies and adds).

A transformer layer consists of an attention block followed by a 2-layer feed-forward network. For the attention block, the main FLOP contributors are the key, query, and value transformation ($6Bsh^2$ operations), attention matrix computation ($2Bs^2h$ operations), attention over values ($2Bs^2h$ operations), and post-attention linear projection ($2Bsh^2$ operations). The feed-forward network increases the hidden size to $4h$ and then reduces it back to h; this requires $16Bsh^2$ FLOPs. Summing these together, each transformer layer results in $24Bsh^2 + 4Bs^2h$ FLOPs for the forward pass. The backward pass requires double the number of FLOPs since we need to calculate the gradients with respect to both input and weight tensors. In addition, we are using activation recomputation, which requires an additional forward pass before the backward pass. As a result, the total number of FLOPs per transformer layer is $4 \times \left(24Bsh^2 + 4Bs^2h\right) = 96Bsh^2\left(1 + \frac{s}{6h}\right)$.

The other main contributor to the FLOP count is the logit layer in the language model head, which transforms features of dimension h to the vocabulary dimension V. The required FLOPs for this operation is $2BshV$ in the forward pass and $4BshV$ in the backward pass, resulting in $6BshV$ FLOPs in total.

Thus, for a transformer model with l transformer layers, the total number of floating-point operations is:

$$96Bslh^2\left(1 + \frac{s}{6h} + \frac{V}{16lh}\right).$$

REFERENCES

[1] Applications of GPT-3. https://openai.com/blog/gpt-3-apps/.

[2] DeepSpeed: Extreme-Scale Model Training for Everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[3] DeepSpeed Repository. https://www.deepspeed.ai/.

[4] GitHub Copilot. https://copilot.github.com/.

[5] Microsoft Translates Spoken Text to Code. https://techcrunch.com/2021/05/25/microsoft-uses-gpt-3-to-let-you-code-in-natural-language/.

[6] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[7] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[8] NVIDIA Selene Supercomputer. https://www.top500.org/system/179842/.

[9] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[10] PyTorch JIT. https://pytorch.org/docs/stable/jit.html.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, and et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, 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*, pages 431–445, 2021.

[15] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[16] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

[17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[18] Andreas Griewank and Andrea Walther. 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.

[19] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. PipeTransformer: Automated Elastic Pipelining for Distributed Training of Transformers. *arXiv preprint arXiv:2102.03161*, 2021.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497–511. 2020.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*, 2018.

[23] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined Backpropagation at Scale: Training Large Models without Batches. *Proceedings of Machine Learning and Systems*, 2021.

[24] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale MLPerf-0.6 Models on Google TPU-v3 Pods. *arXiv preprint arXiv:1909.09756*, 2019.

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv preprint arXiv:2006.15704*, 2020.

[26] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. *arXiv preprint arXiv:2102.07988*, 2021.

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.

[28] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. MLPerf Training Benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[31] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[33] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training, 2018.

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(8):9, 2019.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683*, 2019.

[36] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

[37] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv preprint arXiv:2104.07857*, 2021.

[38] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv preprint arXiv:2101.06840*, 2021.

[39] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*, 2018.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models using GPU Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[41] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient Algorithms for Device Placement of DNN Graph Operators. In *Advances in Neural Information Processing Systems*, pages 15451–15463, 2020.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *arXiv preprint arXiv:1706.03762*, 2017.

[43] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[44] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic Cross-Replica Sharding of Weight Updates in Data-Parallel Training. *arXiv preprint arXiv:2004.13336*, 2020.

[45] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. *Proceedings of Machine Learning and Systems*, 2021.

[46] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR*, abs/1906.08237, 2019.

[47] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.