BREADTH-FIRST PIPELINE PARALLELISM

Joel Lamy-Poirier [1]

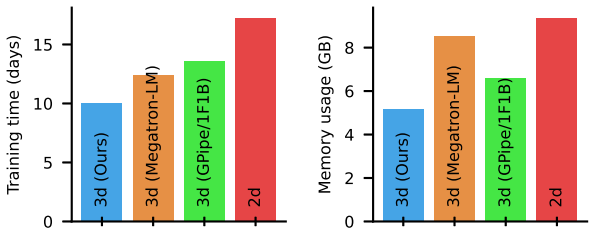
ABSTRACT

We introduce Breadth-First Pipeline Parallelism, a novel training schedule which optimizes the combination of pipeline and data parallelism. Breadth-First Pipeline Parallelism lowers training time, cost and memory usage by combining a high GPU utilization with a small batch size per GPU, and by making use of fully sharded data parallelism. Experimentally, we observed an increase of up to 43% in training throughput for a 52 billion-parameter model using a small batch size per GPU compared to Megatron-LM, which would reduce the training time and cost by the same amount on a large GPU cluster.

1 INTRODUCTION

Large language models (Vaswani et al., 2017; Brown et al., 2020) are quickly becoming an essential tool for natural language processing. However, a challenging aspect of developing such models is their long and expensive training process. A single training may require tens, or even hundreds of thousands of GPU-days worth of computation (Brown et al., 2020; Narayanan et al., 2021; Hoffmann et al., 2022). This results in price tags that can reach several million dollars and a large environmental footprint. Significant efforts have been made towards reducing the training duration and cost, for example by improving the model (Fedus et al., 2021), the training scheme (Hoffmann et al., 2022; Kaplan et al., 2020) or the hardware utilization (Narayanan et al., 2021; Chowdhery et al., 2022; Rajbhandari et al., 2019; Shoeybi et al., 2019; Korthikanti et al., 2022; Dao et al., 2022). However, the training time and cost can only be jointly optimized up to a certain point, as there is an inherent trade-off between them. This trade-off is largely invisible for small models but becomes a limiting factor for large models with tens or hundreds of billions of parameters, that need to be trained on large GPU clusters.

On the one hand, reducing training time requires an increased number of GPUs (N_{GPU}), which in turn needs a larger batch size (B). These extra GPUs will typically be added through data parallelism, so they need to process *different* samples. In general, distributed training requires a *minimum batch size per GPU* (β_{min}), which is typically equal or slightly smaller than one. In practice, most models are trained with a batch size per GPU much higher than this bare minimum, to allow for a higher GPU utilization.



(a) Training time (b) Memory

Figure 1: Predicted training time (a) and memory usage (b) for a 52 billion parameter model on a cluster of 4096 Nvidia V100 GPUs, using our method (Breadth-First Pipeline Parallelism), compared to 3d and 2d baselines.

On the other hand, increasing the batch size hurts the effectiveness of stochastic gradient descent (SGD). The efficiency is maximal when the batch size is well-below an empirical value known as the *critical batch size* B_{crit} (McCandlish et al., 2018), $B \ll B_{\text{crit}}$. However, this *small batch size* regime is unattainable on larger clusters since $B \geq \beta_{\text{min}} N_{\text{GPU}}$. A large body of work (McCandlish et al., 2018; Kaplan et al., 2020; Shallue et al., 2018; Goyal et al., 2017; Smith et al., 2018) has demonstrated that larger batches are able to train machine learning models given a careful adjustment of the training hyperparameters, but they slow down training by requiring extra training samples to reach the same validation loss. That is, they add an overhead which increases the training cost (and time).

Thus, the trade-off can be summarized as follows: reducing the training time requires a larger batch, but a large batch increases the cost and has diminishing returns beyond a certain point. We stress that this concerns the entire training process rather than the batch time or GPU utilization which,

[1]ServiceNow Research, Montreal, Qu'ebec, Canada. Correspondence to: Joel Lamy-Poirier <joel.lamy-poirier@servicenow.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

while important, do not tell the full story. Although this trade-off is difficult (if not impossible) to avoid, we can mitigate it by reducing the batch size per GPU as much as possible, ideally to β_{min}. However, there is a major obstacle to doing so: existing parallelization methods are inefficient in this *small batch size per GPU* regime. Indeed, the state-of-the-art methods such as 2d (Chowdhery et al., 2022; Zhang et al., 2022) and 3d (Narayanan et al., 2021; Korthikanti et al., 2022) parallelism are able to achieve a high GPU utilization (i.e., to use a high fraction of the available flop/s) for a wide range of model sizes, but require a batch size per GPU significantly higher than β_{min} to do so.

Therefore, to train large language models more efficiently, we should look for a training method that not only achieves a high GPU utilization, but that does so with a *low batch-size per GPU*. We propose a novel method, Breadth-First Pipeline Parallelism, that achieves precisely that by using a *looping* placement of the network layers, together with a *breadth-first* micro-batch schedule. Looping pipelines provide a way to reduce the pipeline-parallel overhead from the *pipeline bubble*, as opposed to the more common mitigation method of increasing the batch size. They were first introduced in (Narayanan et al., 2021), where they allowed for an increased computational efficiency. However, we show that the *depth-first* schedule used in that paper is sub-optimal for two main reasons. First, it increases the network overhead, which negates much of the benefit from looping. Second, looping pipelines are more efficient for *smaller* pipelines, which for larger models are prevented by memory constraints. The breadth-first schedule avoids both of these limitations: it allows lowering the memory usage to a minimum with *fully sharded data parallelism* (Rajbhandari et al., 2019) and prevents a network bottleneck by maximizing the *overlap* between network communication and computation. Experimentally, we observed an increase in throughput of up to 43% near β_{min} for a 52 billion parameter model, which translates into a similar (though slightly lower) reduction in training cost and time reduction on large clusters (Figure 1).

This paper is organized as follow. In section 2, we clarify our main claim and its assumptions. In section 3, we introduce the required background on distributed training. In section 4, we introduce our main method, Breadth-First Pipeline Parallelism, and summarize its theoretical justification. In section 5, we demonstrate our claims experimentally.

2 Extended introduction

Our contribution is summarized as follows: Breadth-First Pipeline Parallelism reduces the time and/or cost of training large language models on large GPU clusters, when compared to state-of-the-art methods. Before continuing, we clarify the meaning of this claim and its assumptions.

Time and cost We assume the time and cost to be important for obvious reasons. There may be some flexibility on the training time, but we assume that a *reasonable* training time does not exceed a few months. The price tag depends on many factors, but we approximate it by the total hardware usage in GPU-days. We assume that the type of hardware used is outside of our control, so the training cost and time are determined by the total compute requirement, number of GPUs used and the *GPU utilization*, defined as the fraction of the available computing power that is effectively used:

$$\text{Cost} \propto \frac{\text{Total compute}}{\text{Utilization}}, \quad \text{Time} \propto \frac{\text{Cost}}{\text{Num GPUs}}. \tag{1}$$

In this paper we are particularly interested in the impact of the number of GPUs, which both the total compute and utilization may depend on.

Large GPU cluster Our method is aimed at clusters with hundreds or thousands of GPUs, for which the batch size is a limiting factor due to its effect on SGD (Section 3.5). We assume a cluster of modern NVIDIA GPUs such as V100s or A100. Such a cluster normally consists of several *nodes* (servers) with several GPUs (typically 8) connected with a very fast NVLink connection. The nodes are connected via a slower InfiniBand network. Other types of clusters, such as TPU pods, should also benefit from our method but may affect certain aspect of our analysis, especially when the network structure is different.

Training This refers to pre-training. Fine-tuning may also benefit from our method, but it typically runs on small clusters for which the batch size per GPU is not as important.

Large language model Large language models, i.e. models with a transformer architecture (Vaswani et al., 2017) and more than a few billion parameters, are the main use case for our method, largely because of their size and computational requirement. Smaller models should also benefit from our method but may invalidate certain aspects of our analysis. Other architectures are also possible under certain assumptions, most importantly they should admit a breakdown into similarly sized layers for efficient pipeline parallelism.

State-of-the-art By state-of-the-art, we principally refer to methods that were successfully used to train large language models. These methods all consists of a combination of (up to) three basic methods: *data parallelism* (DP), *pipeline parallelism* (PP) and *tensor parallelism* (DP). We describe each of this method and their variations in the next section. The state-of-the-art are *2d and 3d parallelism*, as described in 3.4. We exclude the methods introduced in (Korthikanti et al., 2022) (*sequence parallelism* and *selective activation recomputation*), as the paper was published

after our codebase was completed. However, these methods are largely orthogonal to ours so should work well in combination. In fact, the lower memory usage of Breadth-First Pipeline Parallelism should make it easier to avoid recomputing activations.

3 DISTRIBUTED TRAINING

In this section, we review the three basic methods (DP, PP and TP), both in isolation and in combination with others. We place emphasis on the memory usage from the training state (weights, optimizer momenta), which for large models largely exceeds the memory available on a single GPU, and on the *batch size per GPU* ($\beta = B/N_{\text{GPU}}$). We also take a special look at the interaction between data and pipeline parallelism, which is the focus of the present paper. Finally, we review how the batch size impacts the training process, which effectively sets a limit on distributed scaling. We remain qualitative and refer to the appendix for more detailed results and examples.

All distributed methods involve network operations, and for efficient training these operations should have a minimal overhead. To achieve this, the operation may be *overlapped* (run in parallel) with computation.[1] An overlapped network operation has a negligible overhead provided that its duration (T_{net}) is less than that of the overlapped computation (T_{overlap}). If overlap is impossible, the network operation should instead be short compared to the total computation time (T_{comp}). In short, efficiency requires either

$$T_{\text{net}} \leq T_{\text{overlap}} \quad \text{or} \quad T_{\text{net}} \ll T_{\text{comp}}. \tag{2}$$

3.1 Data parallelism

Data parallelism (DP) divides the batch between the N_{DP} devices. Each device calculates the loss and gradients for its input, then shares its results through *gradient reduction* and updates the weights.

The input consists of N_{DP} parallel and N_{mb} sequential micro-batches of size S_{mb}, for a batch size $B = N_{\text{DP}} N_{\text{mb}} S_{\text{mb}}$. With pure DP, the batch size per GPU $\beta = N_{\text{mb}} S_{\text{mb}}$ is at least one, i.e., there is a *minimum batch size per GPU* $\beta_{\text{min}} = 1$. However, training at β_{min} may be inefficient. First, a higher micro-batch size leads to more efficient computational kernels due to increased *thread-level parallelism* and reduced *memory IO overhead*, though this is mainly relevant for smaller models, and larger ones generally allow for efficient kernels at any micro-batch size. Second, the gradient reduction is generally a bottleneck at β_{min}. With data parallelism alone, the gradient reduction time is fixed, so additional computation is needed to satisfy Eq. (2), i.e., a higher batch size. The computation time is approximately proportional to the amount of computation, hence the batch size. However, only one of the sequential micro-batches can be overlapped, so the overlapped time is proportional to the micro-batch size. Summing up, we can rewrite Eq. (2) as[2]

$$\beta \geq N_{\text{mb}} \beta_{\text{net}} \quad \text{or} \quad \beta \gg \beta_{\text{net}} \tag{3}$$

for some constant β_{net}. Intuitively, β_{net} represents the lowest value of β for which Eq. (2) can be satisfied. Its exact value depends on the hardware, model and implementation, but is almost always larger than one. As an example, OPT-175B (Zhang et al., 2022) was trained with a micro-batch size of 8, which suggests $\beta_{\text{net}} \lessapprox 8$ for that setup. Note that in the overlapped case, β_{net} is effectively a strict threshold because there is a sharp decline in training efficiency below this value (Figure 2a).

In the original form of data parallelism (DP_0), the computed gradients are all-reduced (summed) between the devices, after which the weights are updated redundantly on each of them. However, DP_0 is inefficient from a memory perspective as it requires a duplication of the whole training state on every device. This duplication can be avoided with *partially sharded data parallelism* (DP_{PS}) (Rajbhandari et al., 2019), where each device instead optimizes a fraction (*shard*) of the weights. The weights are reduce-scattered on the appropriate devices, then updated and *reconstructed* (all-gathered) back on all devices. Due to the efficiency of the network operations, the communication volume remains the same as with DP_0. Given enough data parallelism, DP_{PS} divides the memory usage from the training state by up to 8 times (see Appendix A.2.1). However, this reduction may still not be sufficient for very large models.

The memory usage can be reduced further with *fully sharded data parallelism* (DP_{FS}),[3] where the layers are not kept on device and are instead reconstructed prior to every use. Each layer is reconstructed in both the forward and backward passes, increasing the network usage by at least 50%. The network operations are also repeated for every micro-batch,[4] so the usage is also multiplied by N_{mb}. In short, DP_{FS} shrinks the memory usage from the layer weights and gradients to a minimum (typically that of two layers), but increases the network usage, especially with gradient accumulation.

[1] For simplicity, we assume that the overlap is perfect, i.e. that it does not add any overhead on either operation. In practice, there may be a small overhead; for example, on a Nvidia A100 GPU, the InfiniBand network transfers uses 2 of the 108 execution units (*SM*), slowing the computation by approximately 2%.

[2] This can be derived by substituting $T_{\text{net}} \propto 1$, $T_{\text{comp}} \propto N_{\text{mb}} S_{\text{mb}} = \beta$ and $T_{\text{overlap}} \propto S_{\text{mb}} = \frac{\beta}{N_{\text{mb}}}$, and by absorbing the proportionality factor into a constant β_{net}.

[3] In the language of (Rajbhandari et al., 2019), DP_{PS} corresponds to stage two, while DP_{FS} below corresponds to stage three.

[4] This can be avoided with the breadth-first schedule introduced in Appendix C.

Data parallelism alone can be used to train large models, with DP_{FS}. However, it requires a high batch size per GPU, which makes it less efficient on large clusters (see Section 3.5). Scaling can be improved by combining with model parallelism (pipeline or tensor), to which we now turn.

3.2 Pipeline parallelism

Pipeline parallelism (PP) is a form of *model parallelism*, dividing the model along its *depth* (Huang et al., 2018). Each of the N_{PP} pipeline-parallel devices hosts a single contiguous set of layers, or a *stage* (Figure 3a). In particular, it only stores a fraction of the training state memory. The stages should be identical or near identical in size, so that they take about the same time (and memory) to process a micro-batch.

Parallel computation is achieved with multiple (N_{mb}) sequential micro-batches, with $N_{\text{mb}} \geq N_{\text{PP}}$ ($\beta_{\text{min}} = 1$) so that all devices may perform computation at the same time. However, the data takes time to traverse the pipeline, which causes the devices to be idle (input-starved) much of the time. This phenomenon, known as the *pipeline bubble*, adds an overhead equivalent to $N_{\text{PP}} - 1$ micro-batches, or

$$\text{Bubble} = \frac{N_{\text{PP}} - 1}{N_{\text{mb}}}. \tag{4}$$

Therefore, $N_{\text{mb}} \gg N_{\text{PP}}$ is required for computational efficiency. Although this is a worse requirement than for DP, the method *does* allow for training with a lower batch size, at a reduced efficiency (Figure 2a, non-looped). The bubble is maximal at β_{min}, with an overhead of up to 100%.

When compared to the other methods PP requires the lowest amount of network communication, which is mostly negligible for the large models and fast networks considered in this paper. This communication can also be overlapped with computation, which requires $N_{\text{mb}} \geq N_{\text{PP}} + 1$ since a micro-batch cannot take part in computation while being transferred.

There are two common schedules for pipeline parallelism: with GPipe (PP_{gpipe}) (Huang et al., 2018), the entire forward pass is run first, followed by the backward pass (Figure 4a), while with 1F1B (PP_{1f1b}) (Harlap et al., 2018), the forward and backward steps are alternated so that earlier micro-batches finish as soon as possible. The two schedules have the same computational efficiency, but PP_{1f1b} uses less activation memory.

Pipeline parallelism alone can in theory train moderately large models but is impractical as its scaling is limited by the depth of the model. Instead, PP is most relevant when combined with DP because it may lower the gradient reduction overhead for a low batch size per GPU. For a fixed batch size, it divides all of T_{net}, T_{overlap} and T_{comp}, so in terms of β the efficiency condition becomes

$$\beta \geq \frac{N_{\text{mb}}\beta_{\text{net}}}{N_{\text{PP}}} \quad \text{or} \quad \beta \gg \frac{\beta_{\text{net}}}{N_{\text{PP}}}. \tag{5}$$

While both equations appeared to be improved when compared to Eq. (3), the overlapped equation is in general *worse* due to the high number of sequential micro-batches required for PP. On the other hand, the non-overlapped condition is less constraining, and with a high enough N_{PP} the overhead may be minimal even at β_{min}.

An important caveat when combining DP and PP is that it excludes DP_{FS}. PP requires gradient accumulation, so combining with DP_{FS} would require a repetition of the network operations, making the data-parallel network usage even worse than with DP alone. Instead, DP_0 or DP_{PS} should be used, and a high N_{PP} may be needed to limit the training state memory usage.

There has been recent progress in reducing the size of the pipeline bubble. *Chimera* (Li & Hoefler, 2021) achieves it with a hybrid of data and pipeline parallelism where each device stores multiple pipeline stages, so that it is only idle when *all* the stages are input-starved. However, Chimera requires additional memory and data-parallel network communication, which complicates its use for larger models. An alternative method, *looping pipelines*, introduced in (Narayanan et al., 2021), shrinks the bubble by storing multiple smaller, non-consecutive stages per device (Figure 3b). Looping pipeline avoid the memory and data-parallel network overhead of Chimera, though they require extra pipeline-parallel communication. They are the discussed in more details in Section 4.

3.3 Tensor parallelism

Tensor parallelism (TP) is another form of model parallelism, dividing the model along its *width* (Shazeer et al., 2018; Shoeybi et al., 2019). By extension, it also divides the training state and reduces its memory usage. Each of the N_{TP} tensor-parallel devices processes a subset of the channels for the *same* samples, and shares intermediate activations as needed. In particular, it has no requirement on the batch size, so $\beta_{\text{min}} = N_{\text{TP}}^{-1}$. However, the high network usage of TP (which increases with N_{TP}) requires an extremely fast network such as NVLink, generally restricting TP to a single node.

Although tensor parallelism scales poorly in isolation, it can be used in combination with other methods to reduce the memory usage and train with a lower batch size per GPU. Following the same reasoning as for Eq. (5), we find

$$\beta_{\text{min}} = \frac{1}{N_{\text{TP}}}; \quad \beta \geq \frac{N_{\text{mb}}\beta_{\text{net}}}{N_{\text{PP}}N_{\text{TP}}} \quad \text{or} \quad \beta \gg \frac{\beta_{\text{net}}}{N_{\text{PP}}N_{\text{TP}}}, \tag{6}$$

i.e., TP divides both β_{min} and the minimum β needed for DP network efficiency.

3.4 State-of-the-art

As DP_{FS} and PP are mutually exclusive, there are two options for training large language models.

The combination of all three base methods (DP, PP and TP), **3d parallelism**, was the first to successfully train large language models. This method scales well to large clusters, but generally has a lower GPU utilization due to the pipeline bubble and poor data-parallel network overlap. 3d parallelism was for example used to train GPT-3 (175 B parameters) (Brown et al., 2020) and Megatron-Turing NLG (530 B, with DP_{PS}) (Smith et al., 2022). Although we expect the looped, depth-first pipelines of (Narayanan et al., 2021) to be the most efficient version of 3d parallelism, we also treat non-looped pipelines as state-of-the-art, as they are still widely used and may be more efficient in certain cases (as demonstrated in Section 5).

Alternatively, the combination of DP_{FS} and TP, (a form of) **2d parallelism**, generally allows for a higher GPU utilization, but does not scale as well due to the strict requirement on the batch size per GPU. 2d parallelism has been successfully used to train OPT (175 B) (Zhang et al., 2022) and PaLM (540 B parameters) (Chowdhery et al., 2022), which also used the advantageous network structure of the TPU pod to lower β.

Breadth-First Pipeline Parallelism, introduced in the next section, offers a third option which can efficiently mix DP_{FS} with PP. It also combines the low batch size per GPU of 3d parallelism with the computational efficiency of 2d parallelism.

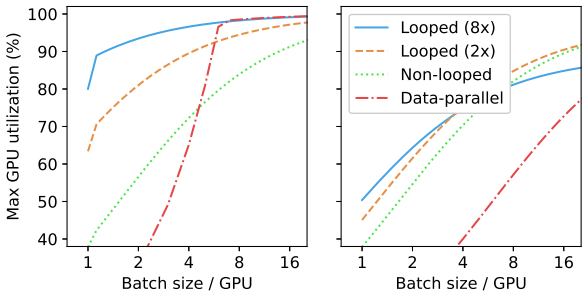
3.5 Effect of the batch size

In stochastic gradient descent, a (mini-)batch is used to approximate the true gradients of the weights with respect to the loss. Increasing the batch size B generally improves this approximation, leading to more efficient steps. For small batches, this is computationally efficient, with larger batches allowing to train for proportionally fewer steps, for a near-constant computing power. However, for large batches the approximation is already accurate and additional samples provide a negligible improvement, leading to a waste of computing power.

Empirically, the number of samples needed to reach a given validation loss has been shown to follow the curve (McCandlish et al., 2018)

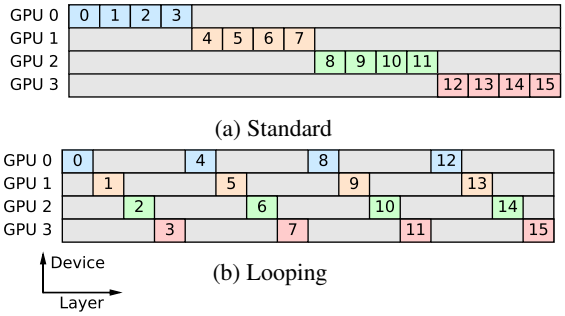
$$\text{Samples} \propto 1 + \frac{B}{B_{\text{crit}}}, \tag{7}$$

where the *critical batch size* B_{crit} depends on the model, training scheme and target validation loss, and it can be estimated by measuring the gradient statistics (see Appendix B). In short, the relative overhead is equal to the ratio B/B_{crit}.



(a) Theoretical efficiency (b) Without network overlap

Figure 2: (a) Comparison of the theoretical efficiency as a function of the batch size per GPU for looped and non-looped pipelines, and for pure data parallelism, for and example with $\beta_{\text{net}} = 6$, N_{TP} =1. Note the jump near $\beta_{\text{min}} = 1$ related to the pipeline-parallel network overlap. (b) The theoretical efficiency for the same configurations without data and pipeline network overlap, shown to emphasize the renewed importance of overlap for looped pipelines.



(a) Standard

(b) Looping

Figure 3: Comparison of the standard and looping layer placements for a 16-layer model. The numbers (and x axis) show the index of the layers.

For example, GPT-3 was trained with a batch size of 3 million tokens, with a critical batch size estimated to 10 million tokens (Kaplan et al., 2020), for an overhead of about 30%.

For both state-of-the-art methods, the number of GPUs N_{GPU} can be scaled with minimal impact on the GPU utilization, provided the batch size per GPU β is kept constant. Therefore, Eq. (1) can be rewritten as a trade-off with respect to N_{GPU}, assuming the cost is proportional to the number of samples processed (7)

$$\text{Cost} \propto 1 + \beta \frac{N_{\text{GPU}}}{B_{\text{crit}}}, \qquad \text{Time} \propto \frac{\text{Cost}}{N_{\text{GPU}}}. \tag{8}$$

4 Breadth-First Pipeline

In this section, we introduce our method, Breadth-first Pipeline Parallelism. We begin by introducing the two main components, looping pipelines and the breadth-first schedule, then present their benefits to large language model train-

ing from a theoretical perspective. We also briefly survey other use cases for our method.

4.1 Looping pipeline and breadth-first schedule

As described in section 3.2, pipeline parallelism typically splits the layers into a single stage per device (Figure 3a). This linear topology minimizes network communication but suffers heavily from the pipeline bubble. In a looping pipeline, first introduced in (Narayanan et al., 2021), we instead divide the network into a large number of (identical or near-identical) stages (N_{stage}), wrapping them around by connecting the first and last device to form a ring (or more precisely a coil), looping $N_{\text{loop}} = \frac{N_{\text{stage}}}{N_{\text{PP}}}$ times (Figure 3b). With this method, data reaches the last device after traversing a fraction of the layers, so the bubble overhead is reduced to

$$\text{Bubble} = \frac{N_{\text{PP}} - 1}{N_{\text{mb}} N_{\text{loop}}}. \tag{9}$$

In a looping pipeline, a given device can only process a single stage at once, even if there is queued input on multiple of them. The schedule may either prioritize earlier micro-batches (*depth-first*), running micro-batches in "sequences" of N_{PP}, or earlier stages (*breadth-first*), running all micro-batches at once. These two options pair naturally with the backward-first approach of PP_{1f1b} and the forward-first approach of PP_{gpipe}, respectively. We call the resulting methods *depth-first pipeline* (PP_{DF}) and *breadth-first pipeline* (PP_{BF}). The former, suggested in (Narayanan et al., 2021), allows lowering the activation memory but only for a large number of micro-batches, i.e., in the scenario we are trying to avoid. It also constrains N_{mb} to a multiple of N_{PP}. Instead, we argue that latter is preferable, allowing to train more efficiently with a low batch size per GPU.

4.2 Analysis

According to Equation (9), computational efficiency now requires $N_{\text{mb}} N_{\text{loop}} \gg N_{\text{PP}}$, so no longer strictly needs a high batch size. Instead, it can be achieved by maximizing N_{loop}, which is however far from trivial. By its definition, a high N_{loop} requires a high N_{stage} and a small N_{PP}.

Increasing N_{stage} is straightforward but adds extra pipeline-parallel network communication. The network usage remains small from a bandwidth perspective, but in practice this communication has a major impact on performance due to the small but numerous latency and synchronization overheads (see Section 5.2). This overhead can be largely avoided by overlapping the transfers with computation. The depth-first schedule as introduced in (Narayanan et al., 2021) does not allow such overlap, since the transfers introduce delays in the pipeline which prevent the micro-batches from looping around when expected, causing the first device to be input-starved. (We believe (but did not verify) this can be addressed by running with sequences of more than N_{PP} micro-batches, essentially forming a hybrid between the two schedules.) The breadth-first schedule, on the other hand, allows for such overlap when $N_{\text{mb}} > N_{\text{PP}}$, because the $N_{\text{mb}} - N_{\text{PP}}$ extra micro-batches can absorb the delay. The increase in batch size is unavoidable because micro-batches cannot take part in computation while being transferred, though a single extra micro-batch is generally sufficient.

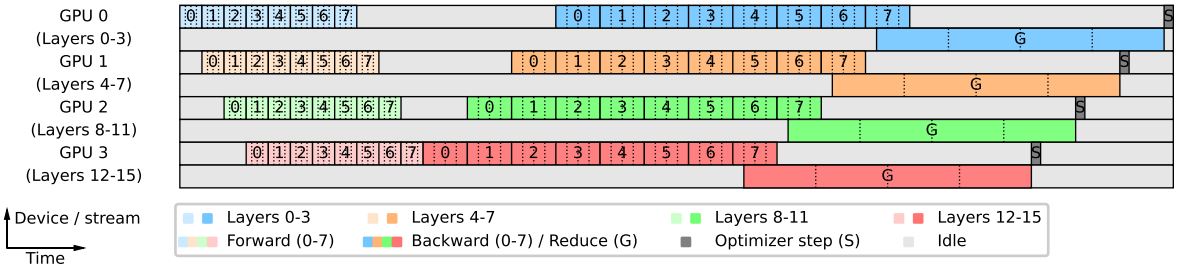
As N_{stage} is limited to the number of layers in the model, increasing it alone may not be enough for a high N_{loop}. For example, in (Narayanan et al., 2021) a 128-layer, trillion-parameter model was trained with $N_{\text{PP}} = 64$, constraining to $N_{\text{loop}} \leq 2$. Further progress can be made by reducing N_{PP}, however such small pipelines go against the recommendations of Section 3.2, which suggested a large N_{PP} (1) to limit the memory usage of the training state and (2) to reduce the data-parallel network overhead.

First, in a non-looping pipeline, a large N_{PP} is needed to limit the training state memory usage for large models. DP_{FS} is inefficient as it involves a repetition of the weight reconstruction and gradient reduction for every micro-batch. A breadth-first schedule avoids any such repetition as it aggregates the steps by layer, so each layer is reconstructed for a single pair of contiguous intervals. The depth-first schedule also improves on non-looped pipelines, but requires a repetition for every micro-batch sequence, and has twice as many reconstructed layers when alternating between the forward and backward passes.

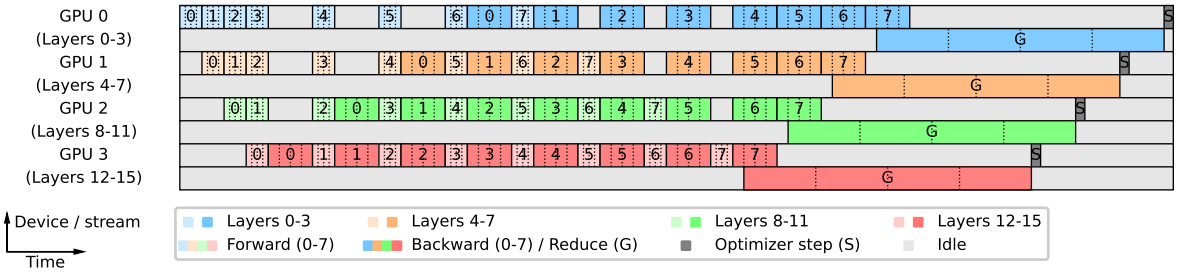
Second, according to Eq. (6), a large N_{PP} may be needed to minimize overhead from the data-parallel network operations, especially because these operations are poorly overlapped with computation. This requirement is often ignored in the literature because it's already avoided through a high batch size per GPU and/or large model parallelism. For example, (Narayanan et al., 2021) selects N_{PP} according to memory usage only, which is justified as the factor $\beta N_{\text{PP}} N_{\text{TP}}$ ranges from 48 to 512, so is always well above $\beta_{\text{net}} \approx 4$ (see Appendix A.3.1). However, this is not necessarily the case for a low batch size per GPU, as Eq. (6) reduces to $N_{\text{PP}} \gg \beta_{\text{net}}$ at β_{min}. With a looping pipeline, the overlap is greatly improved: instead of a single micro-batch, PP_{DF} overlaps with a sequence of N_{PP} micro-batches, while PP_{BF} overlaps with the entire batch. Thus, Breadth-First Pipeline Parallelism has the best network overlap, with a milder efficiency condition

$$\beta \geq \frac{\beta_{\text{net}}}{N_{\text{PP}} N_{\text{TP}}}, \tag{10}$$

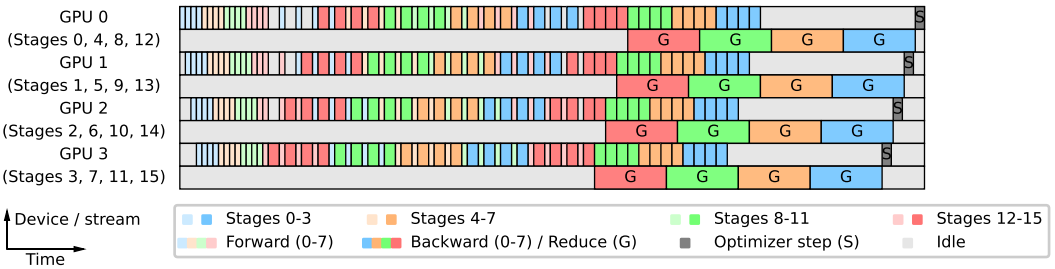
which makes it more efficient for a low N_{PP} and a low β. This condition also sets a lower bound on N_{PP} for efficient training at β_{min} (and an upper bound on N_{loop}), $N_{\text{PP}} \geq \frac{\beta_{\text{net}}}{\beta_{\text{min}}}$.



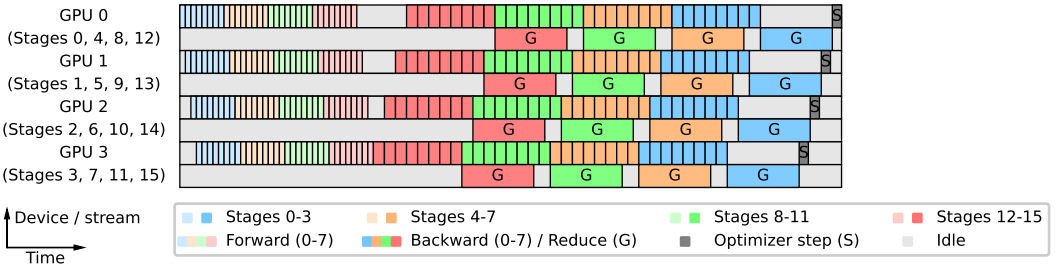
(a) Non-looped pipeline, GPipe schedule (PP_{gpipe}): large bubble, poor overlap



(b) Non-looped pipeline, 1F1B schedule (PP_{1f1b}): large bubble, poor overlap



(c) Looped pipeline, depth-first schedule (PP_{DF}): small bubble, moderate overlap



(d) Looped pipeline, breadth-first schedule (PP_{BF}): small bubble, best overlap

Figure 4: Comparison of the four pipeline schedules considered in this paper (times to scale), for a 16-layer model on 4 pipeline devices, with 8 sequential micro-batches (numbered 0-7), in the presence of data parallelism. We show both the computation (even rows) and the data-parallel communication (odd rows), assumed to run in parallel CUDA streams. (We omit the pipeline-parallel communication for simplicity.) Looped schedules run significantly faster than their non-looped counterparts, with PP_{BF} being the fastest.

Table 4.1: Relative performance of distributed training methods on a large cluster ($N_{\text{DP}} \gg 1$). For large models with a small batch size per GPU, the important quantities are the pipeline bubble, state memory and DP network overhead. Only Breadth-First Pipeline Parallelism achieves good performance on all three, with the depth-first being a close second when DP_{FS} is not required.

Method	Pipeline bubble	State memory[1]	Activation memory[2]	DP network	DP overlap	PP Network	Easy PP overlap[3]	Flexible N_{mb}
No pipeline	0 👍👍	N_{layers} 👎👎	S_{mb} 👍	2 👎👎	$(1 - 1/N_{\text{layers}})/N_{\text{mb}}$ 👍👍[4]	0 👍👍	N.A.	👍👎[4]
No pipeline (DP_{FS})	0 👍👍	2 👍👍	S_{mb} 👍	$3N_{\text{mb}}$ 👎👎👎[4]	$(1 - 1/N_{\text{layers}})/N_{\text{mb}}$? 👍👍[4]	0 👍👍	N.A.	👍👎[4]
GPipe	1 👎👎	$N_{\text{layers}}/N_{\text{PP}}$ 👍	$S_{\text{mb}} N_{\text{mb}}/N_{\text{PP}}$ 👍👎	$2/N_{\text{PP}}$ 👍👍	$(1 - N_{\text{PP}}/N_{\text{layers}})/N_{\text{mb}}$ 👎👎	1 👍	👍	👍
1F1B	1 👎👎	$N_{\text{layers}}/N_{\text{PP}}$ 👍	$\lessapprox 2S_{\text{mb}}$ 👍	$2/N_{\text{PP}}$ 👍👍	$(1 - N_{\text{PP}}/N_{\text{layers}})/N_{\text{mb}}$ 👎👎	1 👍	👎	👍
1F1B (DP_{FS})	1 👎👎	2 👍👍	$\lessapprox 2S_{\text{mb}}$ 👍	$3N_{\text{mb}}$ 👎👎👎👎👎	$1 - N_{\text{PP}}/N_{\text{layers}}$ 👍	1 👍	👎	👍
Chimera[5]	$1/N_{\text{Ch}}$ 👍	$N_{\text{Ch}} N_{\text{layers}}/N_{\text{PP}}$ 👎	$\leq 2S_{\text{mb}}$ 👍	$2N_{\text{Ch}}/N_{\text{PP}}$ 👎	$\approx 1 - 1/N_{\text{Ch}}$ 👍👎	1 👍	👍👎	👎
Depth-first	$1/N_{\text{loop}}$ 👍	$N_{\text{layers}}/N_{\text{PP}}$ 👍	$\lessapprox S_{\text{mb}} + S_{\text{mb}}/N_{\text{loop}}$ 👍	$2/N_{\text{PP}}$ 👍👍	$(1 - N_{\text{PP}}/N_{\text{layers}}) N_{\text{PP}}/N_{\text{mb}}$ 👎	N_{loop} 👎	👎	👎
Breadth-first	$1/N_{\text{loop}}$ 👍	$N_{\text{layers}}/N_{\text{PP}}$ 👍	$S_{\text{mb}} N_{\text{mb}}/N_{\text{PP}}$ 👍👎	$2/N_{\text{PP}}$ 👍👍	$1 - N_{\text{PP}}/N_{\text{layers}}$ 👍	N_{loop} 👎	👍	👍
Breadth-first (DP_{FS})	$1/N_{\text{loop}}$ 👍	2 👍👍	$S_{\text{mb}} N_{\text{mb}}/N_{\text{PP}}$ 👍👎	$3/N_{\text{PP}}$ 👍	$1 - N_{\text{PP}}/N_{\text{layers}}$ 👍	N_{loop} 👎	👍	👍

[1] Assuming DP_{PS} (or DP_{FS}), otherwise the values are multiplied by 3 or more. (Appendix A.2.1)
[2] These values are somewhat misleading, as the non-pipelined methods typically have a higher micro-batch size so generally need *more* activation memory. At β_{net}, all values are equal. With activation checkpointing, this represents the checkpoint memory, and the layer activations and gradients have the same memory usage for all methods.
[3] We believe PP overlap is feasible to some extent for all schedules, but 1F1B and Depth-first add significant complications and need to be modified to support it, while Chimera has only been shown to support it for $N_{\text{mb}} \geq 2N_{\text{PP}}$.
[4] The qualitative performance of the non-pipeline approach assumes $N_{\text{mb}} = 1$, otherwise it is much worse. In Appendix C, we present a breadth-first gradient accumulation method which allows for a good data-parallel performance with $N_{\text{mb}} > 1$ at the cost of extra activation memory.
[5] Chimera with N_{Ch} pipelines, where N_{Ch} is an even integer (typically 2). We assume 'forward doubling' and 'backward halving' ((Li & Hoefler, 2021), Section 3.5), which reduces the pipeline bubble and allows for PP overlap for $N_{\text{mb}} \geq 2N_{\text{PP}}$ at the cost of extra activation memory.

In summary, a breadth-first schedule trains more efficiently with a high N_{loop} and thus for a low batch size per GPU, because it allows for better overlap of the data and pipeline-parallel network communication (Figure 2), and combines better with DP_{FS}. Additional details can be found in Table 4.1, as well as a comparison with alternative methods.

One caveat of our analysis is that the two schedules are relatively similar at the minimum batch size per GPU β_{min}, which is precisely the value we would like to use. At β_{min}, both fully overlap the data-parallel operations, and the depth-first schedule is only slightly less effective with DP_{FS}. For a slightly higher batch size, both should allow for pipeline-parallel network overlap (though we only verified this for PP_{BF}). However, this similarity disappears when considering realistic, non-ideal scenarios. While a looping pipeline significantly reduces the impact of the batch size per GPU, it does not eliminate it. There are still benefits to training with a larger batch size, for example because N_{loop} can only be increased up to a certain point (Figure 2). The exact batch should be selected such that it minimizes the training cost and time when taking into account both the GPU utilization and the batch size overhead (7). For larger clusters, the optimal batch size is near β_{min}, but smaller ones offer more flexibility.

4.3 Additional use cases

Although Breadth-First Pipeline Parallelism is aimed at training large language models as described in section 2, it is also useful in other scenarios. Most importantly, the improved network overlap makes the method well suited for slower networks, for example on GPU clusters without InfiniBand support, that are instead only connected through a slow Ethernet network. This is the for example the case for many cloud platforms and older clusters. In that case, it is more difficult to minimize the data-parallel network overhead, and Breadth-First Pipeline Parallelism is expected to perform more efficiently and at a lower batch size per GPU due to its advantageous network efficiency condition (10). The justification for this is identical to the analysis of Section 4.2, with a high β_{net} instead of a low N_{PP}. Note that this overlap is achieved with looping which affects the condition (10) (smaller N_{PP}), so training is still not optimal and is expected to require a batch size above β_{min}.

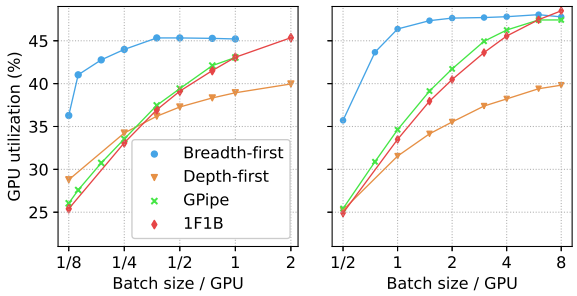
Table 5.1: Details of the models

Model	Num layers	Attention heads	Head size	Hidden size	Seq length
52 B	64	64	128	8192	1024
6.6 B	32	32	128	4096	1024

5 EVALUATION

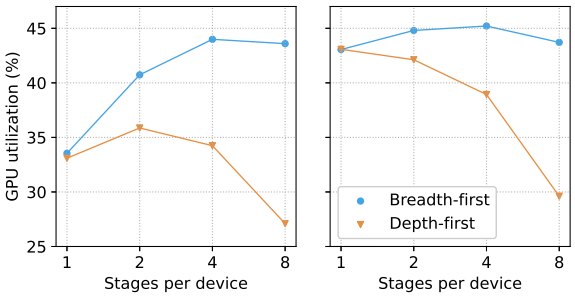
We ran a series of experiments to verify our main claim, namely that breadth-first pipeline parallelism allows for a faster and/or cheaper training of large language models, under the assumptions described in section 2.

We ran our experiments on a cluster of eight DGX-1 nodes, for a total of 64 V100-SXM2-32GB GPUs, connected through an InfiniBand network. All our experiments were run on the same hardware, except for one node which was temporarily replaced due to a hardware failure. Although



(a) 52 B model (b) 6.6 B model

Figure 5: GPU utilization for the looped ($N_{\text{loop}} = 4$) and non-looped schedules as a function of the number of sequential micro-batches ($S_{\text{mb}} = 1$) for (a) the 52 B model ($N_{\text{PP}} = N_{\text{TP}} = 8$, $N_{\text{DP}} = 1$) and (b) the 6.6 B model ($N_{\text{PP}} = 4$, $N_{\text{TP}} = 2$, $N_{\text{DP}} = 8$).



(a) $B = 16$ (b) $B = 64$

Figure 6: GPU utilization for the breadth-first (ours) depth-first (Megatron-LM) schedules as a function of the number of stages per device N_{loop} for the 52 B model for two different batch sizes ($N_{\text{PP}} = N_{\text{TP}} = 8$, $N_{\text{DP}} = 1$, $S_{\text{mb}} = 1$).

we tried our best to also use the same software, our implementation (described in Appendix D) does not support the 1F1B and depth-first pipeline schedules, for which we used Megatron-LM (Narayanan et al., 2021) instead. As Megatron-LM does not support (data and pipeline-parallel) network overlap or DP_{PS}, our results may somewhat underestimate the performance for these schedules.[5] We tested two different models (with a BERT architecture): a moderately large, 52 billion-parameter model, and a smaller, 6.6 billion-parameter model (Table 5.1).

5.1 Fixed configurations

We first compared the four pipeline schedules for matching configurations. In figure 5, we show the GPU utilization as a function of the batch size per GPU (with a fixed micro-batch size) for both models, with fixed distributed configurations. The results are qualitatively similar to the theoretical predictions of Figure 2, where the depth-first and 1F1B schedules are not overlapped (Figure 2b).[6] For smaller batches, the breadth-first schedule is by far the most efficient, minimizing both the bubble and network overheads. The depth-first schedule also reduces the pipeline bubble, but its high network overhead makes the performance *worse* than than the non-looped configurations in most cases. For larger batches, the pipeline bubble is small in all cases, and 1F1B is the fastest because of its lower pipeline-parallel network overhead and memory usage. Note that while Figure 5 supports our claims, it does not reflect the full potential of each method. A more unbiased comparison is performed in section 5.3.

5.2 Bubble and network overheads

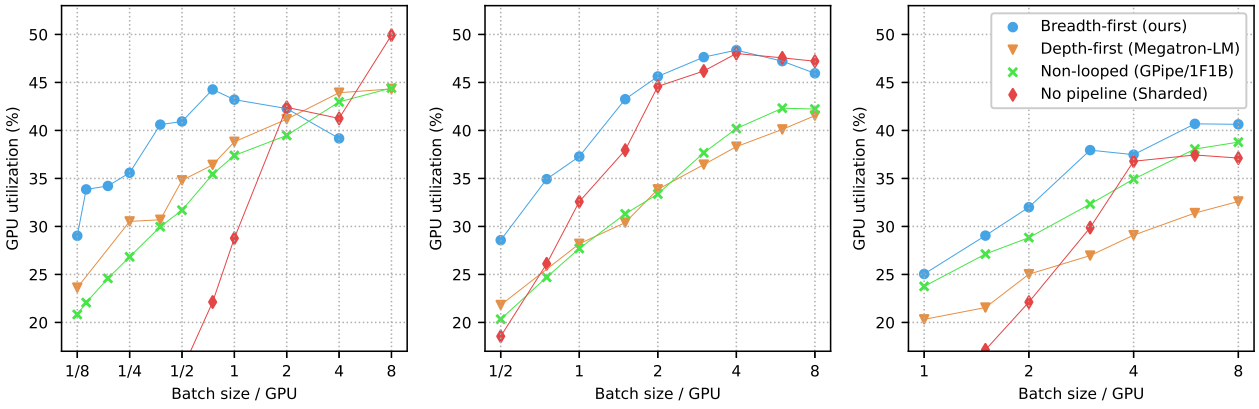
To quantify the relative impact of the bubble and pipeline-parallel network overhead, we compared the breadth-first and depth-first schedules as a function of N_{loop} (including $N_{\text{loop}} = 1$, which corresponds to GPipe and 1F1B, respectively). The results are shown in Figure 6 for the 52 B model with two different batch sizes. Both schedules benefit from the bubble reduction, especially at a smaller batch size (Figure 6a), but the network overhead is far more important for the depth-first schedule, which only benefits from looping for small batch sizes and small N_{loop} (as already observed in Figure 5). From Figure 6b, we estimate that the network overhead is at least 40% for $N_{\text{loop}} = 8$ (30% utilization vs 43% for $N_{\text{loop}} = 1$). This is far higher than the value of 1.6% predicted from the bandwidth usage (Appendix A.3.2), which suggests the overhead is mainly due to latency and synchronization, as claimed in Section 4.2. The breadth-first schedule avoids most but not all of this overhead with network overlap, resulting in $N_{\text{loop}} = 4$ being the optimal value for the present case.

5.3 Optimal configurations

While the results of Section 5.1 agree with our theoretical prediction, they use sub-optimal configurations and thus underestimate the true performance of the methods. For example, the breadth-first schedule stands to benefit from DP_{FS} and reduced model parallelism at higher batch sizes, while Megatron-LM is more efficient with a higher micro-batch size and fewer stages per device. The fixed configuration also prevented a comparison with a non-pipelined

[5] Megatron-LM added support for PSDP ("distributed optimizer") in a later version, published alongside (Korthikanti et al., 2022), but it could not be included in our experiments which were already underway.

[6] We recall that GPipe and 1F1B should have exactly the same performance, other than GPipe running out of memory for larger batch sizes. Thus, the observed difference can be entirely attributed to implementation differences, in particular the lack of network overlap in Megatron-LM (1F1B).



(a) 52 B model (b) 6.6 B model (c) 6.6 B model, Ethernet

Figure 7: Highest GPU utilization observed on a cluster of 64 V100 GPUs for the selected methods, as a function of the batch size.

approach which evidently does not admit a configuration with $N_{\text{PP}} > 1$.

To achieve a fair comparison, we performed an extensive search over the configuration spaces and selected the most efficient one for each method and for a selection of batch sizes. We compared Breadth-First Pipeline Parallelism against the three state-of-the-art methods introduced in Section 3.4: a depth-first pipeline as in (Narayanan et al., 2021), a non-looped pipeline (GPipe or 1F1B), and no pipeline at all. In all cases, we also allowed for data and tensor parallelism. We performed the experiment for both the 52 B and 6.6 B models. To test the effect of our method for slow networks, we repeated the experiment for the smaller model a second time, where we disabled InfiniBand and instead trained using an Ethernet network. The results are shown in Figure 7.[7] See Appendix E for additional details, including a description of each optimal configuration and its memory usage.

For the larger, 52 B model (Figure 7a), the results roughly match the theory (Figure 2), and breadth-first approach is the fastest at all but the largest batch size. Our method outperforms all other methods near $\beta_{\text{min}} = \frac{1}{8}$ (with one extra sample to allow for pipeline-parallel network overlap), running 53% and 43% faster than the non-looped and depth-first baselines, respectively. However, our method does benefit from larger batches, in large part by lowering the amount of tensor model parallelism (see Appendix E), which has a high overhead even for this model size. The non-pipelined approach does achieves higher utilization than our method, but for an excessively high batch size per GPU $\beta = 8$ (even though the actual batch size has a perfectly reasonable value of 512, as will be demonstrated in section 5.4). In this case, efficiency (nearly) plateaus at $\beta = 2$, $N_{\text{TP}} = 2$, suggesting $\beta_{\text{net}} \approx 4$.

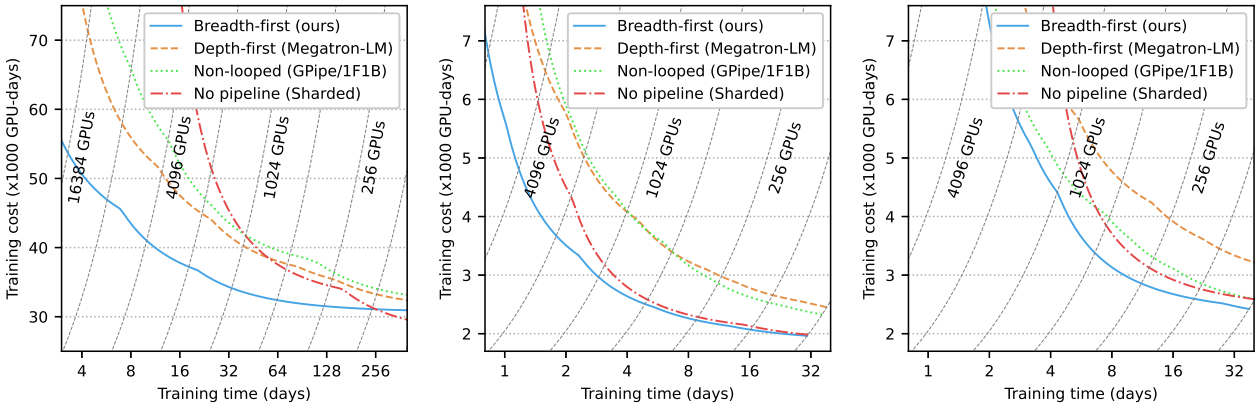
For the smaller, 6.6 B model (Figure 7b), our approach is again the most efficient, but by a smaller margin as the non-pipelined approach performs nearly as well (also with $\beta_{\text{net}} \approx 4$). This is largely because the model is at the lower end of "large" models, and our large model assumptions do not hold as well. For example, the micro-batch size has a noticeable influence on thread-level parallelism, and there is a high model-parallel overhead. None of the approaches is efficient at a low β.

The 6.6 B model can also be trained with an Ethernet network (Figure 7b), though at a reduced efficiency. In that case, our method shows improvements for all β, and the non-pipelined approach is not as efficient even for a high β ($\beta_{\text{net}} \approx 32$). The poor performance of Megatron-LM is largely attributed to the lack of network overlap.

5.4 Trade-off

Evaluating the training time and cost required some extrapolation as we did not have access to a sufficiently large cluster, and it was not possible to run training to completion. We extrapolated the results of Section 5.3 to a range of cluster sizes by scaling data parallelism with a constant batch size per GPU, assuming a constant GPU utilization. This is justified because it has almost no effect on the compute and network usages per GPU. We evaluated the training time for each extrapolation for a base training length of 50,000 times the critical batch size (347 and 176 billion tokens for the 52

[7] The curves of Figure 7 are not particularly smooth, which is a side effect of the optimization over the discrete configuration space. In other words, they result from a combination of many smooth curves such as those of Figure 5, most of which do not cover the entire range of batch sizes, and jumps may occur at the edge of a curve.



(a) 52 B model ($B_{\text{crit}} \approx 6780$) (b) 6.6 B model ($B_{\text{crit}} \approx 3430$) (c) 6.6 B model, Ethernet

Figure 8: Predicted trade-offs between the training cost and time, extrapolated from the results of Figure 7.

B and 6.6 B model, respectively), scaled according to the high batch size overhead predicted from Eq. 7. We used the results of (Kaplan et al., 2020) to estimate the critical batch size. [8][9]

For both models and each method, we selected the best extrapolation as a function of the cluster size, and plotted the associated times and costs (Figure 8). For the 52 B model, the breadth-first pipeline shows significant cost and time improvements at nearly all scales. The largest efficiency seen for the 2d approach is only meaningful for unreasonable training times of above six months. For the smaller model, our method shows improvements for all cluster sizes, though only significantly for larger clusters and at some extra cost.

6 Conclusion

We demonstrated that Breadth-First Pipeline Parallelism reduces the training time and cost of training large language models by combining a high efficiency with a low batch size per GPU. It minimizes both the network overhead and pipeline bubble, while its compatibility with DP_{FS} allows for more efficient configurations which would otherwise not be possible from a memory perspective. As a bonus, DP_{FS} should enable the training of much larger models than previously possible with pipeline-parallelism, with tens if not hundreds of trillions of parameters. This was already possible with 2D methods such as ZeRO-infinity (Rajbhandari et al., 2021), but with a prohibitively high training time. Although our method improves on that level, models of these sizes remain fundamentally limited in terms of *both* training time and cost.

In the next step, we would like to evaluate our method on bigger models and with more modern hardware such as NVIDIA A100 or the upcoming H100. For example, Megatron-LM achieves its highest GPU utilization (57%) for a 530 billion-parameter model on 280 A100s. We would also like to combine our method with other recent progress such as *FlashAttention* (Dao et al., 2022), *sequence parallelism* and *selective activation recomputation* (Korthikanti et al., 2022). These methods are orthogonal to ours, so could be used to further improve training efficiency.

[8] These results were estimated by extrapolating the results for smaller models under a variety of assumptions, so come with a high uncertainty. Because of this (as well as other uncertainties, for example from the extrapolation and the batch size overhead 7 being approximate), our projected trade-offs in Figure 8 should not be considered as quantitatively exact. Note that the vast majority of the uncertainty only affects the scaling of the training time (x-axis).

[9] While the estimated critical batch sizes are higher than typical batch sizes, we remind that 7 still predicts a significant overhead. For example, a batch size of 1024 leads to an overhead of 15% (52 B) or 30% (6.6 B).

Acknowledgements

The author is thankful to Harm de Vries for providing extensive support in writing the paper, and to Deepak Narayanan, Stefania Raimondo, Adam Salvail and Chris Tyler for reviewing and providing feedback.

References

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

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, 2022. URL https://arxiv.org/abs/2204.02311.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. URL https://arxiv.org/abs/1810.04805.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021. URL https://arxiv.org/abs/2101.03961.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *ArXiv*, abs/1706.02677, 2017.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training, 2018. URL https://arxiv.org/abs/1806.03377.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., Driessche, G. v. d., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., and Sifre, L. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018. URL https://arxiv.org/abs/1811.06965.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models, 2022. URL https://arxiv.org/abs/2205.05198.

Li, S. and Hoefler, T. Chimera. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, nov 2021. doi: 10.1145/3458817.3476145. URL https://doi.org/10.1145%2F3458817.3476145.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. An empirical model of large-batch training, 2018. URL https://arxiv.org/abs/1812.06162.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021. URL https://arxiv.org/abs/2104.04473.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners, 2019.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models, 2019. URL https://arxiv.org/abs/1910.02054.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021. URL https://arxiv.org/abs/2104.07857.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training, 2018. URL https://arxiv.org/abs/1811.03600.

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, 2019. URL https://arxiv.org/abs/1909.08053.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022. URL https://arxiv.org/abs/2201.11990.

Smith, S. L., Kindermans, P.-J., and Le, Q. V. Don't decay the learning rate, increase the batch size. *ArXiv*, abs/1711.00489, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., 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, 2022. URL https://arxiv.org/abs/2205.01068.

A ANALYSIS

We now turn to the theoretical analysis of large-scale distributed training, to provide a theoretical justification for the various assertions and claims made in earlier sections, and to demonstrate the benefits of Breadth-First Pipeline Parallelism. As already stated, distributed training is largely guided by memory, network, computational efficiency and batch size. We analyze each of these topics in isolation, then combine them to obtain general prescriptions.

A.1 Setup

We begin by introducing the setup for our analysis. For clarity, we repeat some earlier definitions.

We consider a language model such as Bert (Devlin et al., 2018) or GPT (Radford et al., 2019), with N_{layers} identical transformer (encoder) layers (Vaswani et al., 2017) with hidden size S_{hidden}. Such layers consist of a multi-head self-attention layer with N_{heads} heads of size S_{head}, followed by a two-layer MLP with hidden size S_{mlp}. We assume the common choice $N_{\text{heads}} \times S_{\text{head}} = S_{\text{hidden}}$ and $S_{\text{mlp}} = 4S_{\text{hidden}}$. The model has a total of $N_{\text{params}} \approx 12 N_{\text{layers}} S_{\text{hidden}}^2$ parameters. We assume mixed precision training, the Adam optimizer and activation checkpoints. Our analysis generalizes straightforwardly to other models and setups but may require extra considerations if the layers are not all identical.

The GPU *cluster* consists of N_{Node} *nodes* (servers) of *size* S_{Node} (typically 8), for a total of $N_{\text{GPU}} = N_{\text{Node}} \times S_{\text{Node}}$ *devices* (GPUs or similar machines such as TPUs). In modern Nvidia GPU clusters, the nodes are connected via an InfiniBand network, while the GPUs themselves are connected with a faster NVLink network. When combining distributed methods, the cluster forms a (up to) three-dimensional grid $N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}}$. The devices are parameterized by their *ranks* (location on the grid), and the devices of constant pipeline and tensor rank form a self-contained data-parallel *group* of *size* N_{DP}, and so on. We designate the absence of a method with a group of size one.

With pipeline parallelism, the model is split into N_{stage} stages, looping $N_{\text{loop}} = \frac{N_{\text{stage}}}{N_{\text{PP}}}$ times (one for non-looping pipelines, not necessarily an integer). For a language model as considered here, each stage consists of a fixed number of transformer layers, while the input and output layers may form separate stages or be attached to others, whichever is preferable for a given scenario.

The input batch is split into N_{DP} parallel and N_{mb} sequential micro-batches of size S_{mb}, for a total batch size $B = N_{\text{DP}} \times N_{\text{mb}} \times S_{\text{mb}}$. The batch size per GPU β is minimized with $N_{\text{mb}} = N_{\text{PP}}$ and $S_{\text{mb}} = 1$, i.e., $\beta \geq N_{\text{TP}}^{-1}$. As $N_{\text{TP}} \leq S_{\text{Node}}$, this implies $\beta_{\text{min}} = S_{\text{Node}}^{-1}$. The bulk of the computation consists of matrix multiplications. These come from the linear layers, which require approximately 8 flop of computation per parameter and token, and from self-attention, for a per-GPU total of

$$C_{\text{GPU}} \approx \frac{96 N_{\text{mb}} S_{\text{mb}} N_{\text{layers}} S_{\text{hidden}}}{N_{\text{PP}} N_{\text{TP}}} \times \left(S_{\text{hidden}} + \frac{S_{\text{seq}}}{6} + \frac{S_{\text{voc}}}{16 N_{\text{layers}}} \right). \tag{11}$$

$$\approx \frac{96 N_{\text{mb}} S_{\text{mb}} N_{\text{layers}} S_{\text{hidden}}^2}{N_{\text{PP}} N_{\text{TP}}} \tag{12}$$

To support our analysis, we consider two examples: GPT-3 ($S_{\text{hidden}} = 12288$, $N_{\text{heads}} = N_{\text{layers}} = 96$) and a trillion-parameter model 1T ($S_{\text{hidden}} = 12288$, $N_{\text{heads}} = 160$, $N_{\text{layers}} = 128$), both trained with $S_{\text{seq}} = 2048$, $N_{\text{TP}} = 8$. Unless otherwise specified, we also select a small pipeline with $N_{\text{PP}} = 4$

A.2 Memory

The bulk of the memory usage falls in two main categories. First, the state memory consists of the training state and related variables such as half-precision buffers and parameter gradients, and scales proportionally with the model size. Second, the activation memory consists of the layer activations and their gradients, as well as the activation checkpoints. The activation memory scales principally with the input size, though it also scales with the model size.

Table A.1: Table of symbols.

Symbol	Description	Formula
N_{GPU}	GPUs in the cluster	$N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}}$
N_{DP}	Data-parallel (DP) group size. Can be non-sharded (DP_0), partially sharded (DP_{PS}) or fully sharded (DP_{FS})	
N_{TP}	Tensor-parallel (TP) group size	
N_{PP}	Pipeline-parallel (PP) group size. The schedule can be GPipe (PP_{gpipe}), 1F1B (PP_{1f1b}), breadth-first(PP_{BF}), depth-first (PP_{DF}) or Chimera	
B	Batch size	$N_{\text{DP}} \times S_{\text{mb}} \times N_{\text{mb}}$
S_{mb}	Micro-batch size	
N_{mb}	Sequential micro-batches	
B_{crit}	Critical batch size	
β	Batch size per GPU	B/N_{GPU}
β_{min}	Minimum batch size per GPU	N_{TP}^{-1}
β_{net}	Batch size per GPU for which $T_{\text{comp}} = T_{\text{net}}$	$\approx \beta T_{\text{net}}/T_{\text{comp}}$
N_{layers}	Number of layers	
N_{stage}	Number of pipeline stages.	
N_{loop}	Number of pipeline loops	$N_{\text{stage}}/N_{\text{PP}}$
N_{Ch}	Number of pipelines in the Chimera schedule (even integer)	
T_{net}	Duration of a network operation (ex.: gradient reduction)	
T_{overlap}	Duration of an overlapped computation (ex.: backward pass for the last micro-batch)	
T_{comp}	Total computation time (ex.: full backward pass)	

A.2.1 State memory

The state memory usage depends on the type of data parallelism used, and is approximately:

$$M_0 = \frac{(12 \text{ to } 20) N_{\text{params}}}{N_{\text{PP}} N_{\text{TP}}}, \tag{13}$$

$$M_{\text{PS}} = \frac{(2 \text{ or } 4) N_{\text{params}}}{N_{\text{PP}} N_{\text{TP}}}, \tag{14}$$

$$M_{\text{FS}} = \frac{8 N_{\text{params}}}{N_{\text{layers}} N_{\text{TP}}}. \tag{15}$$

These formulas are justified as follows. For DP_0, the bulk of the memory usage is from the training state itself, i.e., the full-precision weights and the optimizer momenta, which takes 12 bytes per parameter. The (full-precision) gradients and half-precision weight and gradient buffers may add up to 8 bytes per parameters, depending on the setup and implementation. With DP_{PS}, the training state has a minimal memory usage given enough data parallelism, leaving the half-precision buffers as the main contributors to the state memory. With PP_{BF} or $N_{\text{mb}} = 1$, the gradients can be reduced immediately, dividing the memory usage by half. With DP_{FS}, the buffers are only required for the reconstructed layers. In general, two reconstructed layers are sufficient, which allows overlapping computation on a layer with reconstruction on another one.

For example, GPT-3 can be trained on 80 GB GPUs with $N_{\text{TP}} = 8$ and $N_{\text{PP}} = 4$ using DP_{PS} (10 or 20 GB), while 1T requires DP_{FS} (7 GB). Both models can also be trained with DP_0 but need larger pipelines with $N_{\text{PP}} \geq 8$ and $N_{\text{PP}} \geq 64$, respectively.

A.2.2 Activation memory

With activation checkpointing, the full activations and their gradients are only stored for one layer and micro-batch at the time. Their memory usage is approximated by (Korthikanti et al., 2022)

$$M_{\text{act}} = S_{\text{seq}} S_{\text{mb}} S_{\text{hidden}} \left(10 + \frac{24}{N_{\text{TP}}} + \frac{5 S_{\text{seq}} N_{\text{heads}}}{S_{\text{hidden}} N_{\text{TP}}} \right). \tag{16}$$

This memory usage is minimal for a small micro-batch size and scales mildly with the model size. For example, GPT-3 uses 552 MB per sample, while 1T uses 1050 MB per sample. Note that due to memory fragmentation, the memory footprint may be significantly higher (see for example (Rajbhandari et al., 2019)).

For PP_{gpipe} or PP_{BF}, the activation checkpoints have a memory usage of

$$M_{\text{ckpt}} = \frac{N_{\text{mb}} N_{\text{layers}}}{N_{\text{PP}}} \times \frac{2 S_{\text{seq}} S_{\text{mb}} S_{\text{hidden}}}{N_{\text{TP}}}. \tag{17}$$

For PP_{1f1b} and PP_{DF}, the number of checkpoints (first ratio) is capped to $(2N_{\text{PP}} - 1)\frac{N_{\text{layers}}}{N_{\text{PP}}}$ and $N_{\text{layers}} + N_{\text{PP}} - 1$, respectively. When training at β_{min}, the memory usage is 576 MB for GPT-3 and 1600 MB for 1T.

A.3 Network

As described in section 3, the efficiency of network operations mainly depends on the ratio $T_{\text{comp}}/T_{\text{net}}$ of the compute and network times, and on the possibility of overlapping the two operations. This ratio can be estimated by comparing the *arithmetic intensity* I_{op} of the operations, defined as the ratio of computation performed with the amount of data transferred, with the ratio I_{used} of compute and network that is actually performed per unit of time:

$$\frac{T_{\text{comp}}}{T_{\text{net}}} = \frac{I_{\text{op}}}{I_{\text{used}}} \tag{18}$$

Although I_{used} is difficult to determine, it can be approximated by the known ratio I_{hw} of available compute and network for the device, which we call the *hardware intensity*:

$$\frac{T_{\text{comp}}}{T_{\text{net}}} \approx \frac{I_{\text{op}}}{I_{\text{hw}}} \tag{19}$$

For example, a NVidia A100 has 312 Tflop/s of available half-precision tensor core compute, and a network capacity of 46.6 GB/s with InfiniBand and 559 GB/s with NVLink,[10] resulting in intensities of $I_{\text{NVLink}} = 520$ flop/byte and $I_{\text{IB}} = 6240$ flop/byte.

A.3.1 Data-parallel

For DP_0 and DP_{PS}, the network operations (reduction and reconstruction) transfer approximately 8 bytes per parameter per batch, which when compared to the computation (12), give an intensity of

$$I_0 \approx I_{\text{PS}} = N_{\text{mb}} S_{\text{mb}} S_{\text{seq}}. \tag{20}$$

The intensity at β_{min} is numerically equal to the sequence length. For example, when training on a A100 with $S_{\text{seq}} = 2048$, β_{net} has the theoretical value $\lceil I_{\text{op}}/I_{\text{IB}} \rceil = 4$. Note that the model size makes no difference.

With $N_{\text{mb}} > 1$, only the breadth-first schedule allows overlapping with the whole batch. The non-looped schedules can overlap with a single micro-batch, while depth-first is limited to N_{PP} of them. This implies the following requirements for computational efficiency:

$$\text{Non-looped} : S_{\text{mb}} S_{\text{seq}} \geq I_{\text{hw}} \text{ or } N_{\text{mb}} S_{\text{mb}} S_{\text{seq}} \gg I_{\text{hw}}, \tag{21}$$

$$\text{Depth-first} : N_{\text{PP}} S_{\text{seq}} \geq I_{\text{hw}} \text{ or } N_{\text{mb}} S_{\text{mb}} S_{\text{seq}} \gg I_{\text{hw}}, \tag{22}$$

$$\text{Breadth-first} : N_{\text{mb}} S_{\text{mb}} S_{\text{seq}} \geq I_{\text{hw}}, \tag{23}$$

with the breadth-first case being far less constraining and potentially satisfied at β_{net}.

With DP_{FS}, the repeated network operations reduce the intensity, depending on the schedule. With a non-looped pipeline or a non-pipelined schedule with standard gradient accumulation, the intensity becomes

$$I_{\text{FS}} = \frac{2}{3} S_{\text{mb}} S_{\text{seq}}, \tag{24}$$

in particular it is no longer affected by the micro-batch count. Depth-first and breadth-first schedules improve this to

$$I_{\text{FS-DF}} = \frac{2}{3} N_{\text{PP}} S_{\text{mb}} S_{\text{seq}}, \tag{25}$$

$$I_{\text{FS-BF}} = \frac{2}{3} N_{\text{mb}} S_{\text{mb}} S_{\text{seq}}. \tag{26}$$

The efficiency conditions become

$$\text{Non-looped} : \frac{2}{3} S_{\text{mb}} S_{\text{seq}} \geq I_{\text{hw}}, \tag{27}$$

$$\text{Depth-first} : \frac{2}{3} N_{\text{PP}} S_{\text{seq}} \geq I_{\text{hw}}, \tag{28}$$

$$\text{Breadth-first} : \frac{2}{3} N_{\text{mb}} S_{\text{mb}} S_{\text{seq}} \geq I_{\text{hw}}, \tag{29}$$

i.e., a large micro-batch count no longer compensates for the poor overlap.

A.3.2 Pipeline-parallel

Pipeline parallelism requires about $4 \frac{S_{\text{hidden}}}{N_{\text{TP}} N_{\text{layers}}}$ bytes of network per token every $\frac{N_{\text{layers}}}{N_{\text{PP}} N_{\text{loop}}}$ layers, for an intensity

$$I_{\text{PP}} = 24 S_{\text{hidden}} \frac{N_{\text{layers}}}{N_{\text{PP}} N_{\text{loop}}}. \tag{30}$$

For $N_{\text{PP}} = 4$, this results in an intensity of 7.1 M for GPT-3 and 19.7 M for 1T when non-looped, or 294 K for GPT-3 and 614 K for 1T when maximally looped. All these values are far higher than the hardware intensities, but in practice the data transfers are much longer than predicted from Eq. (30), and so is the overhead in the absence of overlap.

A.3.3 Tensor-parallel

In a transformer layer, tensor parallelism (Shoeybi et al., 2019) requires approximately $96 \frac{S_{\text{hidden}}^2}{N_{\text{TP}}}$ flop of computation and $48 S_{\text{hidden}}$ bytes of network for each token, 2/3 of which cannot be overlapped,[11] for an arithmetic intensity

$$I_{\text{TP}} = 2 \frac{S_{\text{hidden}}}{N_{\text{TP}}}. \tag{31}$$

[10] These values differ slightly from the advertized values of 25 GB and 600 GB because of the conversion between base 10 and base 2, and because we consider the total bandwidth only (Input+Output).

[11] The forward pass involves two non-overlapped all-reduce operations, each requiring 8 bytes of network for each hidden parameter and token. The backward pass adds an equivalent amount from the activation recomputation and two overlapped extra all-reduce from the gradient computation.

As claimed, this restricts TP to the largest models and small-scale fast intra-node networks. For example, with $N_{\text{TP}} = 8$, the intensity is 3072 for GPT-3 and 6400 for 1T, with expected overheads of about 11% and 5%, respectively.

B CRITICAL BATCH SIZE

We provide a summary of the theoretical justification for Eq. (7)describing the overhead from the batch size. More details can be found in the original paper (McCandlish et al., 2018).

In stochastic gradient descent, we attempt to minimize a loss function $L(\theta)$ having only access to a batch of noisy estimates G_i of its gradient, $0 \leq i < B$. By the central limit theorem, the average G_{est} over the samples approximates to a multivariate normal distribution $\mathcal{N}(G(\theta), \Sigma(\theta))$, where $G(\theta)$ is the true gradient and the covariance matrix $\Sigma(\theta)$ is the noise. To the second order approximation, a step of size $-\epsilon G_{\text{est}}$ modifies the loss by

$$\Delta L \approx -\epsilon G_{\text{est}}^T G + \frac{1}{2}\epsilon^2 G_{\text{est}}^T H G_{\text{est}}, \tag{32}$$

where H is the Hessian, with expected value

$$\mathbb{E}\left[\Delta L\right] \approx -\epsilon |G|^2 + \frac{1}{2}\epsilon^2 \left(G^T H G + \text{tr}(H\Sigma)\right). \tag{33}$$

This value is minimized with

$$\epsilon = \frac{|G|^2}{G^T H G + \text{tr}(H\Sigma)}, \quad \mathbb{E}\left[\Delta L\right] \approx \frac{\frac{1}{2}|G|^4}{G^T H G + \text{tr}(H\Sigma)}. \tag{34}$$

This result depends on the batch through the covariance matrix (from the central limit theorem). We extract that dependence by defining

$$\Sigma = \frac{\Sigma_0}{B}, \quad B_{\text{noise}} = \frac{\text{tr}(H\Sigma_0)}{G^T H G} \approx \frac{\text{tr}(\Sigma_0)}{|G|^2}. \tag{35}$$

The latter approximation assumes H is close to the identity, which is not expected to hold in practice, but has been empirically shown to provide a fair estimate when the Hessian is unavailable. Using these definitions, we rewrite Eq. (34) as

$$\mathbb{E}\left[\Delta L\right] \approx \frac{\Delta L_0}{1 + B_{\text{noise}}/B}, \tag{36}$$

where ΔL_0 is an unimportant proportionality factor. If neither of these quantities varies significantly, the same amount of progress is made each step, and a target loss is attained after a number of steps

$$\text{Steps} \propto 1 + \frac{B_{\text{noise}}}{B}, \tag{37}$$

In terms of samples seen, this rewrites as

$$\text{Samples} \propto 1 + \frac{B}{B_{\text{noise}}}, \tag{38}$$

Despite the variety of assumptions and approximations used to obtain this results (second order, central limit theorem, optimal learning rate, consistent step, etc.), most of which are not expected to hold in practice, in (McCandlish et al., 2018) Eq. (38) was shown to hold experimentally when replacing B_{noise} by an empirical parameter B_{crit}, generating Eq. (7)

$$\text{Samples} \propto 1 + \frac{B}{B_{\text{crit}}}. \tag{39}$$

In most cases, B_{noise} is a good approximation of the critical batch size, $B_{\text{crit}} \approx B_{\text{noise}}$

C BREADTH-FIRST GRADIENT ACCUMULATION

We consider a data-parallel scenario with multiple sequential micro-batches. This may happen when a high batch size is needed to mitigate the gradient reduction network overhead, and the micro-batch size is limited by activation memory constraints. In that case, we typically use a *depth-first* schedule, where a given micro-batch goes through the entire forward and backward passes before the next one begins. This schedule achieves the goal of limiting the memory usage, as all intermediate activations can be dropped between micro-batches. However, the gradient reduction cannot begin until the last micro-batch, leading to poor overlap with computation (Figure 9c). Therefore, the network overhead is mitigated, but not eliminated. With DP_{FS} there is no mitigation at all since the network operations (reconstruction and reduction) need to be repeated for each micro-batch (Figure 9b).

A breadth-first schedule solves these problem, allowing to overlap the gradient reduction with most of the backward pass, and with DP_{FS} avoiding duplicating the operations (Figures 9c and 9d). The breadth-first schedule comes at the cost of memory, since more activations need to be stored at once. However, when using activation checkpoints, this memory increase remains small, and for larger models the memory savings from DP_{FS} are far more important.

However, when the stage outputs coincide with activation checkpoints, this only increases the checkpoint memory, which remains smaller than the layer activation unless there are lots of sequential micro-batches. Furthermore, for large models, the state memory is the bottleneck, so the memory usage may be *lower* for the breadth-first schedule when combined with DP_{FS}.

D IMPLEMENTATION

We evaluated our method using a custom library, with a model and training scheme identical to the Megatron-LM implementation of Bert (Shoeybi et al., 2019; Narayanan et al., 2021). As a reference, we used the source code

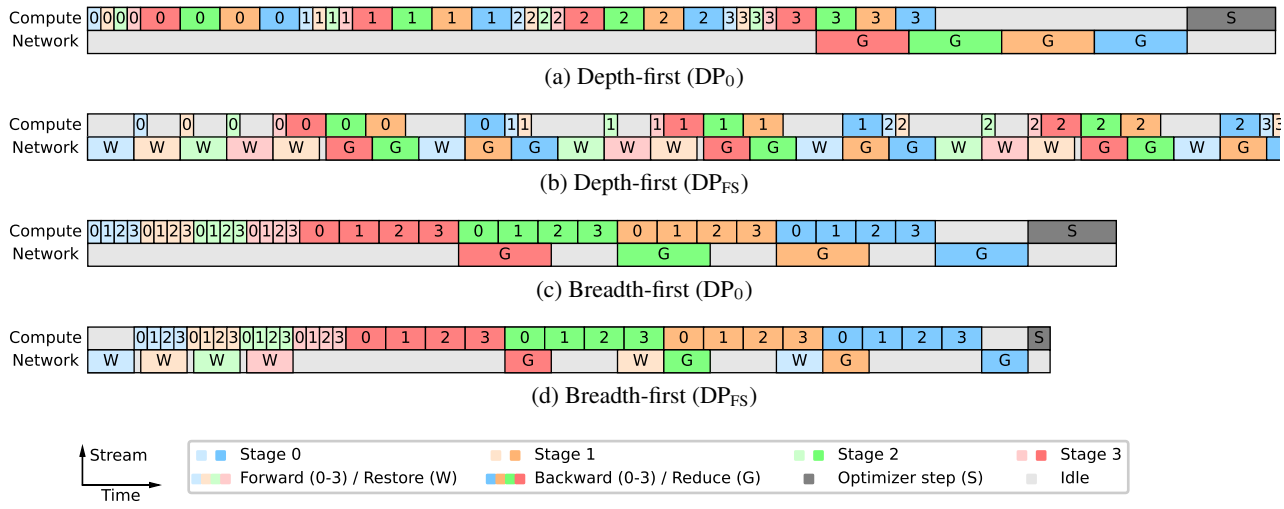


Figure 9: Example depth-first and breadth-first schedules for gradient accumulation with DP_0 and DP_{FS}. The depth-first approach achieves poor network overlap, and with DP_{FS} involves a costly repetition of the network operations. Both issues are solved with the breadth-first schedule, resulting in a faster training.

for Megatron-LM as it was just before the publication of (Korthikanti et al., 2022) (commit e156d2f). We verified that the model forward and backward passes are identical, generating the same kernels on the GPU, except for the fused GeLU (we used a slightly faster compiled implementation). However, our implementation differ for mixed precision (we overlap the casting), the optimization step (we removed some unnecessary synchronizations) and data and pipeline parallelism (as described below).

D.1 Distributed training

We used a custom implementation of data parallelism, with built-in support for mixed precision and sharded data parallelism. It relies on the breakdown into stages to optimize the data conversion and transfer, where the stage size may be set to any number of transformer layers. For that purpose, the embedding and final layers are either treated equivalently as transformer layers, or merged with the adjacent layer, depending on what is most efficient for a given configuration. We use a double-buffered approach to achieve network overlap at a minimal memory cost. For example, the computation for a given stage may be done in parallel with the weights for the next stage being restored on the other buffer. The network operations are performed in-place, avoiding the memory and kernel time overhead of network buffers.

We implemented breadth-first pipeline parallelism, with support for network overlap as described in Section 4. It reduces to standard, non-looped pipeline parallelism when the stages are sufficiently large.

D.2 Memory efficiency

Large models tend to suffer heavily from memory fragmentation, where the GPU has enough memory to allocate a given tensor but not in a contiguous chunk, which leads to unnecessary out-of-memory errors. To reduce the fragmentation, we pre-allocate tensor whenever possible, including for the training state (fp32 weights and gradients, optimizer momenta), fp16 weight and gradient buffers, the activation checkpoints, the pipeline receive buffers. Apart from a few small tensors, this leaves the intermediate layer activations and their gradients as the only actively allocated tensors; these still suffer heavily from memory fragmentation but are difficult to pre-allocate within Pytorch.

We also observed a significant memory overhead and in some cases important slowdowns due to how the Pytorch caching allocator is implemented[12]. The tensor-parallel network operations are run in a separate CUDA stream set up by NCCL, and while that stream is immediately synchronized with the default stream, this limits the ability to free the tensors involved in the operation. As a result, the tensor memory is blocked from the CPU perspective until the operation is completed on GPU, which increases the memory usage when there are many queued kernels[13]. This may pre-

[12]This problem is not unique to our implementation. It was observed with *Pytorch Fully Sharded Data Parallel*, and we were able to reproduce it on Megatron-LM without pipeline parallelism. (The Megatron-LM implementation of pipeline parallelism includes frequent CUDA synchronizations which, prevent the issue from happening, although inefficiently.)

[13]In that scenario, some of the tensors involved may have been deleted on CPU, which means the underlying memory block will be available when the queued kernels complete. The CUDA

vent the caching allocator from finding enough memory for future kernels, at which point it synchronizes the GPU, then flushes the cached allocations[14]. The flushing operation is relatively slow, causing some idle time on the GPU side. The overhead is multiplied when there are many parallel devices, as the slowdowns happen at different times, and each is enough to block the whole group. In some cases, we observed a combined overhead of more than 100%. We fixed this by explicitly preventing the kernel queue from growing two big, by adding frequent CUDA synchronizations (on earlier events, so the synchronization is non-blocking.)

E EXPERIMENTATION DETAILS AND RESULTS

For each model, method and batch size, we ran a grid search over the following parameters, as applicable:

- The implementation we used. This variable was fixed for all methods except non-looped, which was supported by both ours (GPipe) and Megatron-LM (1F1B).
- The pipeline-parallel group size N_{PP}.
- The tensor-parallel group size N_{TP}.
- The micro-batch size S_{mb}.
- The number of sequential micro-batches N_{mb}.
- The number of transformer stages per device N_{loop}. This excludes the embedding and output layers which may add extra stages in our implementation.
- Whether we enabled DP_{FS} or DP_{PS}. Note that we only tried DP_{FS} (not DP_{PS}) for breadth-first and non-pipelined configurations and DP_{PS} for non-looped configurations. We also did not try DP_{PS} with Megatron-LM, which did not support it at the time. This may have led to an underestimation of the performance in some cases.

We excluded configurations that were obviously inferior, such as those with excessive model parallelism and those inefficiently combining DP_{FS} and gradient accumulation. We also excluded configurations that were certain or highly likely to run out of memory.

We ran each configuration for 50 batches and measured the the average batch time, excluding the first 10 batches. We then calculated the throughput using Eq. (11). We also measured the GPU memory usage, taken as the peak memory allocated on the GPU with global rank zero. This value is generally representative of the entire cluster, though the actual memory usage may vary slightly depending on the pipeline-parallel rank, and it does not take memory fragmentation into account.

Note that the measured memory usage does not reflect the true memory efficiency of each method because we optimized for throughput only, and because we used a relatively small cluster which limits the benefits of sharded data parallelism. To account for DP_{FS} and DP_{PS}, we predicted the minimum memory usage achievable with with sharded data parallelism, i.e., on an arbitrarily large cluster. For our implementation, the difference is exactly 16 bits per parameter, while for Megatron-LM it is approximately 12 bits per parameter.[15]

The most efficient configurations for each model, method and batch size are presented in tables E.1, E.2 and E.3, as well as the number of configurations tried in each case.[16]

caching allocator provides a way to reuse that memory before the kernels are run, by predicting the future memory usage. However, it can only do so efficiently in a single-stream setting.

[14]The flushing is designed for a different scenario, where the memory is available but there is no cached block of the correct size. In the present case, the flush is generally unnecessary as the synchronization frees up many blocks but is performed either way.

[15]The difference is due to the full-precision gradients. They are pre-allocated in our implementation, which reduces memory fragmentation and mitigates the issues described in Appendix D.2, but at the cost of extra memory. In Megatron-LM, they are instead allocated on the fly and do not typically contribute to the peak memory usage, which occurs around the beginning of the backward pass. Nevertheless, way be slightly overestimating the memory usage for smaller batch sizes where the activation memory is small, and the gradients potentially contribute to the peak memory.

[16]The number of configurations reflects the size of the configuration space and memory usage, and is not representative of the effort spent for each method. In particular, the larger number of configurations tried for the breadth-first schedule is due to its reduced memory usage (with DP_{FS}), which allows for many more distributed configurations.

Table E.1: Selected optimal configurations for the 52 B model.

Method	Batch size	Implementation	N_{PP}	N_{TP}	S_{mb}	N_{mb}	N_{loop}	Sharded	Throughput (Tflop/s/GPU)	Memory (GB)	Memory min (GB)	Configs
Breadth-first	8	Ours	8	8	1	8	4	✗	36.28	15.96	3.22	10
Breadth-first	9	Ours	8	8	1	9	8	✗	42.33	14.74	2.25	3
Breadth-first	12	Ours	8	8	1	12	4	✗	42.77	16.66	3.92	10
Breadth-first	16	Ours	4	8	1	8	8	✓	44.49	16.60	3.60	30
Breadth-first	24	Ours	4	8	2	6	8	✓	50.76	17.96	4.96	29
Breadth-first	32	Ours	8	4	1	16	4	✓	51.17	19.12	5.13	59
Breadth-first	48	Ours	8	2	1	12	8	✓	55.34	19.73	5.80	59
Breadth-first	64	Ours	4	2	1	8	16	✓	54.01	20.23	6.30	89
Breadth-first	128	Ours	4	2	2	8	8	✓	52.85	24.65	11.66	51
Breadth-first	256	Ours	2	2	1	16	32	✓	48.97	26.32	12.38	5
Depth-first	8	Megatron-LM	8	8	1	8	2	✗	29.53	15.78	6.42	3
Depth-first	16	Megatron-LM	8	8	2	8	4	✗	38.16	15.94	6.57	8
Depth-first	24	Megatron-LM	8	8	1	24	2	✗	38.37	15.78	6.42	3
Depth-first	32	Megatron-LM	8	8	4	8	4	✗	43.50	17.77	8.41	13
Depth-first	48	Megatron-LM	8	8	2	24	2	✗	45.52	16.27	6.91	8
Depth-first	64	Megatron-LM	8	8	4	16	4	✗	48.52	19.18	9.81	15
Depth-first	128	Megatron-LM	8	8	4	32	4	✗	51.46	19.18	9.81	18
Depth-first	256	Megatron-LM	16	4	4	64	2	✗	54.91	21.35	11.62	19
Depth-first	512	Megatron-LM	8	8	4	128	2	✗	55.41	19.87	10.50	8
Non-looped	8	Ours	8	8	1	8	1	✗	26.04	16.87	4.38	3
Non-looped	9	Ours	8	8	1	9	1	✗	27.59	16.99	4.50	1
Non-looped	12	Ours	8	8	1	12	1	✗	30.74	17.38	4.89	2
Non-looped	16	Ours	8	8	1	16	1	✗	33.54	17.89	5.40	9
Non-looped	24	Ours	8	8	1	24	1	✗	37.46	18.91	6.42	7
Non-looped	32	Ours	8	8	2	16	1	✗	39.62	20.12	7.63	16
Non-looped	48	Ours	8	4	1	24	1	✓	44.30	22.71	9.74	14
Non-looped	64	Ours	8	4	1	32	1	✓	46.74	23.75	10.78	19
Non-looped	128	Megatron-LM	8	8	2	64	1	✗	49.35	15.75	6.38	10
Non-looped	256	Megatron-LM	16	4	2	128	1	✗	53.72	16.33	6.61	8
Non-looped	512	Megatron-LM	8	8	4	128	1	✗	55.52	17.68	8.31	4
No pipeline	8	Ours	1	8	1	1	1	✓	4.73	14.23	1.98	1
No pipeline	16	Ours	1	8	2	1	1	✓	9.43	15.44	3.19	3
No pipeline	24	Ours	1	8	3	1	1	✓	14.07	16.64	4.39	1
No pipeline	32	Ours	1	8	4	1	1	✓	18.79	17.85	5.60	6
No pipeline	48	Ours	1	8	6	1	1	✓	27.66	20.29	8.04	3
No pipeline	64	Ours	1	8	8	1	1	✓	35.97	22.73	10.48	10
No pipeline	128	Ours	1	2	4	1	1	✓	53.01	21.43	9.18	12
No pipeline	256	Ours	1	2	4	2	1	✓	51.57	21.43	9.18	12
No pipeline	512	Ours	1	2	4	4	1	✓	62.40	21.44	9.19	7

Table E.2: Selected optimal configurations for the 6.6 B model.

Method	Batch size	Implementation	N_{PP}	N_{TP}	S_{mb}	N_{mb}	N_{loop}	Sharded	Throughput (Tflop/s/GPU)	Memory (GB)	Memory min (GB)	Configs
Breadth-first	32	Ours	4	2	1	4	4	✗	35.72	15.56	2.34	15
Breadth-first	48	Ours	4	2	1	6	8	✗	43.66	14.61	1.64	15
Breadth-first	64	Ours	2	2	1	4	4	✓	46.60	5.67	4.02	35
Breadth-first	96	Ours	2	1	1	3	8	✓	54.07	5.95	4.30	35
Breadth-first	128	Ours	2	1	1	4	8	✓	57.03	6.10	4.45	55
Breadth-first	192	Ours	2	1	2	3	8	✓	59.55	6.72	5.06	55
Breadth-first	256	Ours	2	1	2	4	8	✓	60.45	7.02	5.36	71
Breadth-first	384	Ours	2	1	4	3	8	✓	59.02	8.25	6.59	71
Breadth-first	512	Ours	2	1	4	4	16	✓	57.44	8.95	5.52	80
Depth-first	32	Megatron-LM	4	2	1	4	2	✗	27.27	16.27	6.54	3
Depth-first	64	Megatron-LM	4	2	2	4	4	✗	35.24	16.35	6.62	8
Depth-first	96	Megatron-LM	4	2	1	12	2	✗	38.00	16.27	6.54	3
Depth-first	128	Megatron-LM	4	2	4	4	4	✗	42.33	16.44	6.72	13
Depth-first	192	Megatron-LM	4	2	2	12	2	✗	45.55	16.29	6.56	8
Depth-first	256	Megatron-LM	4	2	4	8	4	✗	47.89	16.44	6.72	18
Depth-first	384	Megatron-LM	4	2	4	12	2	✗	50.14	16.32	6.59	13
Depth-first	512	Megatron-LM	4	2	4	16	2	✗	51.92	16.32	6.59	20
Non-looped	32	Ours	4	2	1	4	1	✗	25.42	16.73	3.76	4
Non-looped	48	Ours	4	2	1	6	1	✗	30.88	16.86	3.89	2
Non-looped	64	Ours	4	2	1	8	1	✗	34.63	17.00	4.03	10
Non-looped	96	Ours	4	2	1	12	1	✗	39.13	17.27	4.30	7
Non-looped	128	Ours	4	2	1	16	1	✗	41.72	17.54	4.57	16
Non-looped	192	Ours	4	1	1	12	1	✓	47.09	11.21	7.78	12
Non-looped	256	Ours	4	1	1	16	1	✓	50.25	11.49	8.06	21
Non-looped	384	Ours	4	1	1	24	1	✓	52.90	12.06	8.63	19
Non-looped	512	Ours	4	1	2	16	1	✓	52.78	12.94	9.51	24
No pipeline	32	Ours	1	8	4	1	1	✗	23.19	14.04	1.72	6
No pipeline	48	Ours	1	8	6	1	1	✗	32.64	14.74	2.42	3
No pipeline	64	Ours	1	8	8	1	1	✗	40.73	15.45	3.13	10
No pipeline	96	Ours	1	8	12	1	1	✗	47.44	16.89	4.57	6
No pipeline	128	Ours	1	2	4	1	1	✓	55.73	4.40	2.82	13
No pipeline	192	Ours	1	2	6	1	1	✓	57.74	5.30	3.72	10
No pipeline	256	Ours	1	1	4	1	1	✓	60.02	6.01	4.43	15
No pipeline	384	Ours	1	1	6	1	1	✓	59.45	7.19	5.62	13
No pipeline	512	Ours	1	1	8	1	1	✓	59.01	8.38	6.80	16

Table E.3: Selected optimal configurations for the 6.6 B model (Ethernet).

Method	Batch size	Implementation	N_{PP}	N_{TP}	S_{mb}	N_{mb}	N_{loop}	Sharded	Throughput (Tflop/s/GPU)	Memory (GB)	Memory min (GB)	Configs
Breadth-first	64	Ours	4	4	2	8	4	✗	31.31	8.70	2.21	88
Breadth-first	96	Ours	4	4	4	6	4	✗	36.31	9.47	2.98	88
Breadth-first	128	Ours	2	4	4	4	8	✗	40.00	16.40	3.79	113
Breadth-first	192	Ours	2	4	8	3	8	✗	47.44	18.04	5.43	113
Breadth-first	256	Ours	2	4	4	8	8	✗	46.85	18.83	6.21	121
Breadth-first	384	Ours	2	4	16	3	4	✗	50.86	23.35	10.73	130
Breadth-first	512	Ours	2	4	16	4	8	✗	50.80	25.02	12.41	106
Depth-first	64	Megatron-LM	8	2	2	8	2	✗	25.40	8.78	3.56	25
Depth-first	96	Megatron-LM	8	2	1	24	2	✗	26.94	8.77	3.54	16
Depth-first	128	Megatron-LM	8	1	1	16	2	✗	31.28	17.43	6.98	28
Depth-first	192	Megatron-LM	8	1	1	24	2	✗	33.70	17.43	6.98	25
Depth-first	256	Megatron-LM	8	1	2	16	2	✗	36.37	17.45	7.00	28
Depth-first	384	Megatron-LM	8	1	2	24	2	✗	39.24	17.45	7.00	28
Depth-first	512	Megatron-LM	8	1	2	32	2	✗	40.75	17.45	7.00	28
Non-looped	64	Ours	8	2	1	16	1	✗	29.70	9.52	2.55	40
Non-looped	96	Ours	8	2	1	24	1	✗	33.91	9.81	2.84	35
Non-looped	128	Ours	8	2	1	32	1	✗	36.05	10.10	3.13	52
Non-looped	192	Ours	8	1	1	24	1	✗	40.42	18.78	4.85	50
Non-looped	256	Ours	8	1	1	32	1	✗	43.66	19.10	5.17	56
Non-looped	384	Ours	8	1	1	48	1	✗	47.58	19.74	5.81	60
Non-looped	512	Ours	8	1	1	64	1	✗	48.48	20.38	6.45	49
No pipeline	64	Ours	1	8	8	1	1	✗	15.37	15.45	3.13	4
No pipeline	96	Ours	1	8	12	1	1	✗	21.43	16.89	4.57	3
No pipeline	128	Ours	1	8	16	1	1	✗	27.65	18.33	6.02	5
No pipeline	192	Ours	1	8	24	1	1	✗	37.35	21.22	8.90	4
No pipeline	256	Ours	1	8	32	1	1	✗	45.99	24.10	11.78	5
No pipeline	384	Ours	1	8	48	1	1	✓	46.81	19.09	17.51	5
No pipeline	512	Ours	1	8	32	2	1	✗	46.40	24.13	11.81	5