低秩压缩流水线后向激活，保留上个microbatch的压缩误差加在下一个microbatch梯度，再压缩;可以只压缩后面几个microbatch
将embedding的all-reduce+synchronization换成all-reduce
前面的stage最后dp，可以压缩前面的stage的dp

Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression

Jaeyong Song
jaeyong.song@yonsei.ac.kr
Yonsei University
Seoul, South Korea

Jinkyu Yim*
skyson00@snu.ac.kr
Seoul National University
Seoul, South Korea

Jaewon Jung
jungjaewon@yonsei.ac.kr
Yonsei University
Seoul, South Korea

Hongsun Jang
hongsun.jang@snu.ac.kr
Seoul National University
Seoul, South Korea

Hyung-Jin Kim
hj.windy.kim@samsung.com
Samsung Electronics
Hwaseong, South Korea

Youngsok Kim
youngsok@yonsei.ac.kr
Yonsei University
Seoul, South Korea

Jinho Lee
leejinho@snu.ac.kr
Seoul National University
Seoul, South Korea

ABSTRACT

In training of modern large natural language processing (NLP) models, it has become a common practice to split models using 3D parallelism to multiple GPUs. Such technique, however, suffers from a high overhead of inter-node communication. Compressing the communication is one way to mitigate the overhead by reducing the inter-node traffic volume; however, the existing compression techniques have critical limitations to be applied for NLP models with 3D parallelism in that 1) only the data parallelism traffic is targeted, and 2) the existing compression schemes already harm the model quality too much.

In this paper, we present Optimus-CC, a fast and scalable distributed training framework for large NLP models with aggressive communication compression. Optimus-CC differs from existing communication compression frameworks in the following ways: First, we compress pipeline parallel (inter-stage) traffic. In specific, we compress the inter-stage backpropagation and the embedding synchronization in addition to the existing data-parallel traffic compression methods. Second, we propose techniques to avoid the model quality drop that comes from the compression. We further provide mathematical and empirical analyses to show that our techniques can successfully suppress the compression error. Lastly, we analyze the pipeline and opt to selectively compress those traffic lying on the critical path. This further helps reduce the compression error. We demonstrate our solution on a GPU cluster, and achieve superior speedup from the baseline state-of-the-art solutions for distributed training without sacrificing the model quality.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; **Neural networks**.

KEYWORDS

Distributed Systems, Systems for Machine Learning, Large-scale NLP Training, Pipeline Parallelism, 3D Parallelism, Gradient Compression, Communication Optimization

1 INTRODUCTION

In the era of deep learning (DL), the size of models has been growing at an exponential rate [29, 50]. Now, utilizing tens to hundreds of GPU-equipped nodes in a distributed manner to rapidly train a single model has become a common practice [25, 59, 65]. To achieve high speedups with multiple GPUs, the early work on distributed training employs *data parallelism* [27, 39, 42]. With data parallelism, a DL model gets duplicated to multiple nodes with identical weight parameters. Then, a batch of input data is split into the nodes and each node performs backpropagation on its copy of the DL model. After every node completes backpropagation on their input data, inter-node communication is necessary for sharing the parameter gradients so that each node updates its copy of the DL model to have exactly identical states. The volume of the communication is proportional to the size of the weights, which becomes a burden as the model size becomes larger. Although the method has a problem with limited scalability of model size, it has been employed in many environments [2, 25, 48, 84] because it is easy to implement.

To minimize the increase in the communication overhead, prior studies proposed to compress the parameter gradients of a DL model caused by data parallelism [11, 44, 81]. The parameter gradients are known to be especially robust to some errors, so the DL model can tolerate a certain amount of misdirection. By reducing the bitwidth of [81], taking the top-k of [12, 41, 44], or performing low-rank approximation on [75] the gradients, the parameter gradient compression techniques successfully reduce the inter-node traffic, making it feasible to perform distributed training on top of low-end ethernet [44], and improve the speedup with high-speed inter-node networks [12, 75].

This is an author preprint version of a paper which will appear in the proceedings of ASPLOS'23.
* Both authors contributed equally to this research.
† Corresponding author.

However, with the rapid growth in the DL model sizes, especially for large natural language processing (NLP) models [8, 19, 47, 57], a single GPU can no longer store a complete DL model even with the smallest batch size (i.e., one). As a workaround, recent distributed training frameworks [59, 65] suggest splitting the models into multiple pieces with *pipeline* parallelism and *tensor* parallelism. Pipeline parallelism distributes the layers of a DL model to the nodes, whereas tensor parallelism partitions a layer into multiple sub-layers and distributes the sub-layers to the GPUs. The three types (i.e., data, pipeline, tensor) of parallelism are collectively known as the *3D parallelism* of distributed training.

Unfortunately, we found that existing compression methods targeting only the data-parallel traffic are inefficient for distributed training of a large DL model, especially for those using 3D parallelism [59, 65]. First, data-parallel traffic is no longer the sole source of inter-node communication. Pipeline parallelism requires point-to-point communication for passing forward activations and backward gradients between layers, and tensor parallelism incurs several all-reduce communications during forward and backward passes. Second, we find that naïvely applying the existing compression techniques on recent larger NLP models causes a significant drop in the model quality (i.e., downstream task accuracy). Moreover, applying the compression on the newly-introduced pipeline parallel traffic yields even more quality drops as illustrated in Section 3. This states the need for new techniques that can suppress the compression error, or reduce the communication volume without loss. Last, existing compression methods [12, 41, 75] overlook the opportunity coming from the pipelined schedule. When a training process is pipelined, much of the communication latency is hidden by the computation (i.e., forward and backward pass) of the following micro-batches. Thus, blindly compressing all communication traffic only yields more compression errors without any throughput gain.

In such circumstances, we propose *Optimus-CC* (Compressed Communication), a fast and scalable distributed training framework for large NLP models. Our goal is to achieve throughput gain by exploiting characteristics of inter-node communication in 3D parallelism without sacrificing the model quality. Optimus-CC employs the following three key ideas: First, *compressed backpropagation* targets the inter-stage communication of pipeline parallelism. It compresses the inter-stage backward traffic which contains activation gradients. By focusing on the pipeline epilogue and propagating compression errors in a lazy manner within an iteration, compressed backpropagation can increase speedup without compromising the model quality. Second, *fused embedding synchronization* fuses the two all-reduce communications from the embedding layers into a single all-reduce communication. The embedding synchronization occurs due to an embedding layer being shared at the beginning and the end of the network. We find that fusing the synchronization reduces the communication volume without changing the mathematical outcome. Third, *selective stage compression* targets data parallel communication, but restricts the compression target to only a few stages. The data parallel communication begins as soon as the backward pass of the corresponding stage finishes. Therefore, the earlier stages are likely to place the data parallel traffic on the critical path, so selective stage compression chooses not to compress some stages for less error.

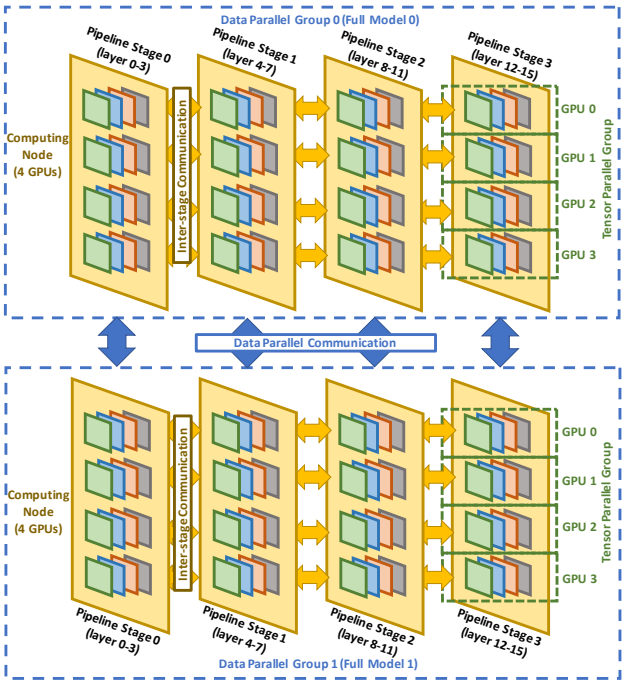


Figure 1: Example distributed training configuration with 3D parallelism, with 2 data-parallel groups, 4 tensor-parallel groups, and 4 pipeline stages.

We tested our approach on a cluster of 128 Nvidia A100 GPUs with 200Gb/s Infiniband HDR interconnect to achieve up to 15.09% speedup on multi-billion NLP models without compromising application performance and up to 44.91% speedup with comparable model quality.

Overall, our contributions can be summarized as follows:

- We propose Optimus-CC, a fast and scalable distributed training framework which aggressively compresses inter-node communication while sustaining the model quality. To the best of our knowledge, Optimus-CC is the first work to accelerate large-scale NLP model training with inter-node communication compression.
- We propose three techniques tailored for reducing the communication volumes of 3D parallelism: compressed backpropagation, fused embedding synchronization, and selective stage compression. They significantly improve training throughput without suffering from the model quality drop.
- We demonstrate the high effectiveness of Optimus-CC by training two versions of GPT2 [57] models which have 8.3- and 2.5-billion-parameter respectively. In our large-scale GPU cluster setting equipped with a high-end interconnect, we obtain significant speedup on training.

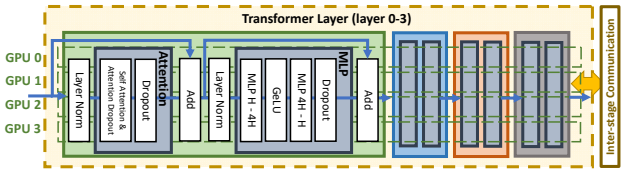


Figure 2: Layer structure of Megatron-LM. The weight parameters of each layer are split into multiple GPUs.

2 BACKGROUND

2.1 3D Parallelism

3D parallelism is a strategy of utilizing data parallelism, tensor parallelism, and pipeline parallelism commonly adopted to large NLP model training [59, 65] as depicted in Fig. 1. *Data parallelism* [2, 17, 84] duplicates the same model weights on several groups. Then, mini-batches of a dataset are equally split into each data-parallel group. After a forward and backward pass, each group d has different parameter gradients $G^{(d)} = \nabla f^{(d)}(W)$ (where f is a loss function and W is the weight parameters) of the dedicated mini-batch for them. To maintain identical model weights in all the data-parallel groups, averaged parameter gradients ($\frac{1}{D}\sum_d^D G^{(d)}$) from all the D groups are updated on weights. (i.e., $W \leftarrow W - \alpha\frac{1}{D}\sum_d^D G^{(d)}$). It can reduce the training time because the forward and backward passes are parallelly executed in multiple GPUs. The communication overhead of the averaged parameter gradient is the main cost issue for using data parallelism.

Pipeline parallelism and *tensor parallelism* [10, 13, 36] let us handle large models that do not even fit in device memory by distributing a part of the model into multiple GPUs. During forward and backward pass, these parallelisms should communicate activations and gradients between GPUs. This mechanism accompanies huge communication volume, so the model should be carefully split to minimize the overhead.

Pipeline parallelism [29, 49, 83] places a set of layers (i.e., a stage) to a GPU as depicted in Fig. 1 and overlaps its executions in a pipelined manner as illustrated in Fig. 4a. A number of prior work [29, 49, 83] carefully schedule the executions such that the pipeline bubbles are minimized. Between the stages, activations and activation gradients $\nabla f(Y)$ (where Y is the intermediate activations) have to be communicated in a point-to-point manner for forward and backward passes. Although the latency of many point-to-point communications are hidden by overlapping with computations, some communications are still not hidden, which become the target of this work.

Tensor parallelism splits a layer into multiple GPUs. By duplicating the activations to the GPUs in the same tensor-parallel group, each GPU applies different weight parameters to produce partial results. Hence, all-reduce communications are required during forward and backward passes for the activations and activation gradients. [65]

2.2 Large-Scale NLP Model

In the NLP task, a large model [8, 19, 58] based on transformer [74] is preferred due to its representational capability. Megatron-LM [50, 65] is a framework for training extreme-scale NLP models using 3D parallelism. Fig. 1 and Fig. 2 show the details of Megatron-LM. It applies *tensor parallelism* to split model layers as depicted in Fig. 2. To reduce the communication time, each tensor parallel group is placed within a server node such that its communications can utilize high-bandwidth intra-server interconnects (i.e., NVLink).

Multiple server nodes are utilized for data parallelism and pipeline parallelism. Megatron-LM also uses interleaved scheduling and 1F1B scheduling to reduce the pipeline bubbles and peak memory usage. Because the communications from data parallelism and pipeline parallelism take place in an inter-node network, those two communications often become bottlenecks in the 3D parallelism of Megatron-LM. This work targets reducing the volume of such communications, thereby maximizing the throughput of large NLP model training.

2.3 Gradient Compression

Gradient compression is commonly used for mitigating the huge parameter gradient communication cost in data parallelism. Top-k, quantization, and low-rank approximation are three popular approaches for gradient compression.

Top-k [44, 64] based approaches take top-k elements of the gradients per layer to compress. In this case, top-k selection entails sorting overhead which is critical in fast training. Some work [11, 12] use quasi-sorting to reduce this overhead. One problem with the top-k method is that as the number of GPUs in data parallelism increase, the total number of elements to communicate linearly increase, because each GPU will independently choose its own k elements. Furthermore, top-k methods require an additional gather operation for the chosen indices which induces more overhead than the actual parameter gradient sharing in a multi-worker environment. ScaleCom [12] resolves this gradient build-up problem because of the gather operation by using top-k index similarity between gradients in each worker. Ok-Topk [41] also successfully addresses these problems using its efficient top-k threshold estimation algorithm.

选择相似的top-k?

Quantization-based approaches quantize gradients to reduce communications. TernGrad [81] uses ternary (-1, 0, 1) values to aggressively reduce communications. AdaComp [11] additionally combines residual addition and quantization to minimize the error from lossy compression. SignSGD [5] uses the sign of the momentum to significantly quantize gradients. 1-bit Adam [70] reduces the communication of the Adam optimizer by quantization using the stability of optimizer variables after warm-up iterations.

On the other hand, the low-rank approximation uses matrix factorization to reduce the total communication cost of gradients. Factorization cost is the main bottleneck of matrix factorization. PowerSGD [75] diminishes this cost by iterating power-iteration, which is required for classical SVD, only once. It reuses the factorized matrix from the previous gradient compression stage to minimize the error of compression.

One common drawback of these methods is that all the top-k, quantization, and low-rank approximation methods are inherently lossy, and can degrade the accuracy of a model. Many approaches try to mitigate this problem by considering the momentum [44], providing feedback on errors [11, 75], or estimating the gradient

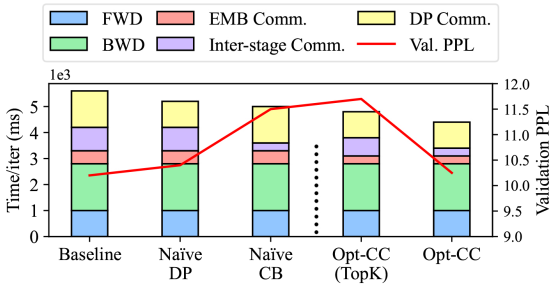


Figure 3: Motivational experiments. The breakdown demonstrates the overhead of communications, and naive compression yields severe increases in the validation perplexity.

movements [35, 83]. Optimus-CC is no exception, as it adopts a low-rank approximation to compress the communications. We propose several methods to maintain the model quality, while compressing the communications for the 3D parallelism.

3 MOTIVATIONAL STUDY

Fig. 3 illustrates the breakdown of 125K iterations training in a large NLP model (GPT-2.5B) using the popular Megatron-LM [65] framework. Because each GPU runs different stages with heterogeneous scheduling, we follow an approach similar to CPI stack [21]. In other words, we turn off each communication/computation and observe the execution time difference. The experiments have been done with 128 GPUs over 16 computing nodes connected with 200Gbps Infiniband HDR interconnect. Please refer to Section 9.1 for a detailed setup.

Even with such high-speed interconnect, a significant portion is spent on the inter-node communication part ('DP Comm.' for data-parallel communication, 'Inter-stage Comm.' for communication between pipeline stages, and 'EMB Comm.' for embedding synchronization). One exception is tensor-parallel all-reduce that happens intra-node, which we included in the FWD, and BWD bars. In Megatron-LM, the tensor parallel GPUs are always placed in a single computing node, and connected via relatively fast NVLink with 600GBps bandwidth per GPU, which result in an almost negligible communication time.

Perplexity (PPL) is a representative validation metric in NLP, which measures how confidently the model predicts the next word after given sentences (the lower is the better). The 'naive DP' bar depicts the training time and the validation PPL when a low-rank gradient compression method [75] is applied to the data-parallel communication to reduce its time. It shortens the training time by reducing the volume of communication. However, contrary to the common wisdom on smaller models, even the modest amount of compression rate worsens the model quality as indicated by the increase in validation perplexity. Considering that model quality is a key metric for DNN models, this much deterioration is not acceptable. A similar observation can be made from the 'naive CB' bar representing the compression of inter-stage communication. We naively applied the same low-rank compression ratio to see the potential, even though no attempt has been made on the compression of inter-stage communication. Similar to the observation in the data-parallel gradient compression, the compression yields an unacceptable rise in perplexity, and the phenomenon worsens when both communication types are compressed.

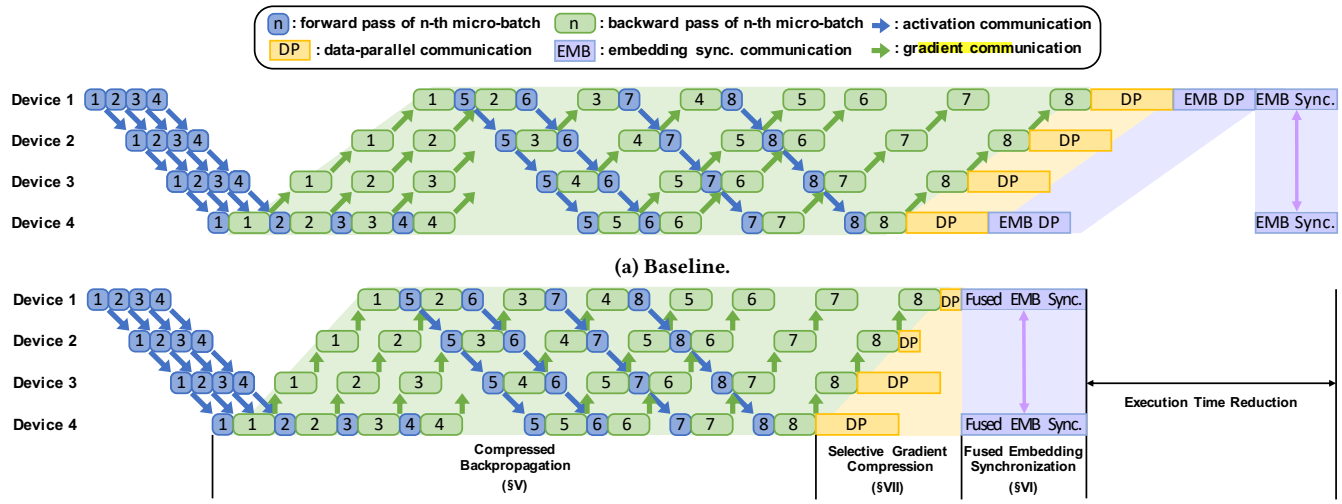
Optimus-CC targets applying compression to those inter-node communications, without sacrificing the quality of the model, as shown in the 'Opt-CC' bar. Optimus-CC compresses the communications until they only consume a negligible amount of execution time to achieve its speedup, and most importantly, maintains the perplexity and zero-shot task quality of the baseline method. As a result, Optimus-CC reduces the training time taken for 125K iterations from 8.00 days into 6.97 days, while maintaining the perplexity equal to that of the baseline. Note that Opt-CC uses low-rank approximation for compression. We also depicted the result of top-k-based compression on the inter-stage communications in the 'Opt-CC (TopK)' bar, but it brings worse perplexity because it is unsuitable for compressing point-to-point communications.

4 OVERVIEW OF OPTIMUS-CC

In this section, we describe three main components of Optimus-CC. Fig. 4a illustrates the baseline 1F1B scheduling [49] of each pipeline stage, considering the communications. The blue boxes represent the forward passes, and the blue arrows represent the inter-stage forward communication. The green boxes represent backward passes where each pass takes approximately twice that of the forward passes, with the green arrows representing the inter-stage backward communications. After the backward pass is complete, data-parallel communication takes place (*DP*). The communication is made with the GPUs of the same stage in other data-parallel ranks (yellow boxes in Fig. 4). Note that this also includes the communication for the parameters of embedding layers (*EMB DP*), but it is depicted separately. Then, the embedding synchronization (*EMB Sync*) happens, between the first stage and the last stage of the pipeline to sync weights because the first stage and the last stage share the same embedding weights.

The main goal of our proposed framework is to reduce the latest finish time or the whole execution, especially that of the first stage, because the next iteration starts from the forward pass of the first stage. Our method pursues the goal by reducing the volume of communication in three sections shaded with green, yellow, and purple depicted in Fig. 4b.

In the first technique, *compressed backpropagation* (Section 5), we compress the inter-stage communication. In the timing diagram, this technique has the effect of shaping the parallelogram-shaped green area closer to a rectangle, contributing to the shorter execution time. We provide two enabler techniques, *lazy error propagation* and *epilogue-only compression*, which help avoid the model quality drop. The second technique, *fused embedding synchronization* (Section 6) targets shared embedding layers, which generate two all-reduce communications. We fuse these two all-reduce communications into a single all-reduce communication for traffic volume reduction. Lastly, *selective stage compression* (Section 7) targets data-parallel traffic, but only compresses those lying on the critical path. This provides a better trade-off between model quality and execution speed than traditional data-parallel traffic compression.



(a) Baseline.

(b) Proposed Optimus-CC.

Figure 4: Timing diagram of baseline and Optimus-CC, including the communication time between stages. Each stage and communication are not drawn as the exact scale for better visibility.

5 COMPRESSED BACKPROPAGATION

- **Compression target**: Pipeline parallelism — inter-stage backpropagation.
- **Method**: Compress the inter-stage activation gradients, with the help of lazy error propagation and epilogue-only compression.

With compressed backpropagation, we target the inter-stage backpropagation traffic (activation gradients, green arrows in Fig. 4a and 4b) using low-rank approximation. Because the communication appears between computations of each stage, its performance impact becomes large, especially when there are many pipeline stages. In principle, compressing the forward traffic could provide a similar speedup, but it would severely break the convergence of the model. While naively compressing the backward traffic also breaks the convergence as demonstrated in Fig. 3, we provide two techniques for preserving the convergence, lazy error propagation and epilogue-only compression.

5.1 Lazy Error Propagation

Lazy error propagation is a novel technique that enables compressed backpropagation without a model quality drop. As shown in Section 3, naively applying compression to inter-stage backpropagation communication causes a severe model quality drop, due to errors from the lossy nature of the compression algorithm.

Fig. 5 illustrates the inter-stage backpropagation with lazy error propagation. When lazy error propagation is used, after compressed data are sent to the earlier stage for backpropagation, the error is preserved in the memory. This preserved error is added to the backward traffic in the next micro-batch. For example, ① in the third micro-batch, the device 2 generates $\nabla f^{16-23}(Y_1)$: gradients from samples 16-23. ② the gradient is compressed and sent to device

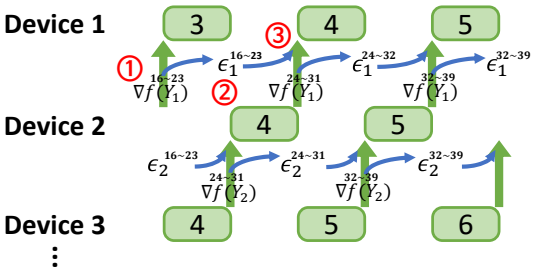


Figure 5: Lazy error propagation, with micro-batch size of 8.

1, while the compression error ϵ_1^{16-23} is preserved in memory. ③ in the next micro-batch, when $\nabla f^{24-31}(Y_1)$ is generated, the ϵ_1^{16-23} is added. The sum is compressed before being sent, and the new error ϵ_1^{24-31} is preserved.

Even though the error is slightly delayed to the next micro-batch, the impact on the model quality is almost negligible because the model update only happens after all micro-batches are processed. In other words, lazy error propagation does not suffer from the weight staleness effect because all micro-batches are still executed based on the same version of the weights. (See Section 9.3 for the detailed model quality results.)

Lazy error propagation delays the error from a specific micro-batch to later micro-batches' errors. For lazy error propagation to work, the resulting average gradient should correctly approximate the average that would have resulted when compression is not used. An intuition is that the gradients are being accumulated over the entire mini-batch, unlike the activations that directly affect the result of the model. Because of this, we posit that the errors propagated in later micro-batch does not greatly affect the total

sum. We found that this is true when the distribution of the errors are independent to that of the activations. In the following, we provide a mathematical analysis of this condition.

Consider a K-layer MLP[1] as a toy example:

$$E = f(W_K \cdot W_{K-1} \cdots W_1 \cdot X^{(i)}), \tag{1}$$

where E is the error from some error function $f(\cdot)$, W_k is the weights for layer k and $X^{(i)}$ is the i^{th} sample from a mini-batch. In addition, we define $Y_k^{(i)}$ as the intermediate activation values at the output of layer k from the i^{th} sample of the mini-batch as below.

$$Y_k^{(i)} = W_k \cdot Y_{k-1}^{(i)}, \qquad Y_0^{(i)} = X^{(i)}, \tag{2}$$

Applying the common backpropagation,

$$\nabla f^{(i)}(Y_k) = W_{k+1} \cdot \nabla f^{(i)}(Y_{k+1}), \tag{3}$$

$$\nabla f^{(i)}(W_k) = \nabla f^{(i)}(Y_k) \cdot Y_{k-1}^{(i)}, \tag{4}$$

Taking $k = K-1$ (the penultimate layer) for example, the parameter gradient for update becomes:

$$\nabla f^{(i)}(W_{K-1}) = W_K \cdot \nabla f^{(i)}(Y_K) \cdot Y_{K-2}^{(i)}, \tag{5}$$

Since the update occurs after the completion of the entire mini-batch, assuming there are N samples in a mini-batch, the update with SGD becomes:

$$W_k \leftarrow W_k - \eta \cdot G_k, \tag{6}$$

$$G_k = \frac{1}{N}\sum_i^N \nabla f^{(i)}(W_k), \tag{7}$$

where η is a learning rate and G represents the average gradient. Now, consider each W_k becomes an individual pipeline stage. Then, the communicated backpropagation values between stage k and $k+1$ are $\nabla f^{(i)}(Y_k)$. When we apply compression, we essentially add an error vector ϵ_i^k to it (i.e., $\nabla f^{(i)}(Y_k)$ becomes $\nabla f^{(i)}(Y_k) + \epsilon_k^{(i)}$). With lazy error propagation, the errors are kept to be subtracted from the next micro-batch that has n samples:

从k+1到k

不是加？

$$\nabla f^{(i)}(Y_k) = W_{k+1} \cdot ((\nabla f^{(i)}(Y_{k+1}) - \epsilon_{k+1}^{(i-n)}) + \epsilon_{k+1}^{(i)}), \tag{8}$$

$$\nabla f^{*(i)}(W_k) = ((\nabla f^{(i)}(Y_k) - \epsilon_k^{(i-n)}) + \epsilon_k^{(i)}) \cdot Y_{k-1}^{(i)}, \tag{9}$$

i-n是lazy，i是压缩

where $\nabla f^{*(i)}(W_k)$ is the approximate parameter gradient with compressed backpropagation.

Setting $k = K-1$ again,

$$\begin{aligned}\nabla f^{*(i)}(W_{K-1}) &= (\nabla f^{(i)}(Y_{K-1}) - \epsilon_{K-1}^{(i-n)} + \epsilon_{K-1}^{(i)}) \cdot Y_{K-2}^{(i)}, \\ &= (W_K \cdot (\nabla f^{(i)}(Y_K) - \epsilon_K^{(i-n)} + \epsilon_K^{(i)}) - \epsilon_{K-1}^{(i-n)} + \epsilon_{K-1}^{(i)}) \cdot Y_{K-2}^{(i)}.\end{aligned}$$

Aggregating them over N samples to obtain G_{K-1}^* yields,

$$\begin{aligned}G_{K-1}^* &= \frac{1}{N}\sum_i^N \nabla f^{*(i)}(W_{K-1}), \\ &= \frac{1}{N}\sum_i^N (W_K \cdot (\nabla f^{(i)}(Y_K) + \epsilon_K^{(i)} - \epsilon_K^{(i-n)}) + \epsilon_{K-1}^{(i)} - \epsilon_{K-1}^{(i-n)}) \cdot Y_{K-2}^{(i)}\end{aligned} \tag{10}$$

Substituting Eq. (5) results in the below equation.

$$= G_{K-1} + \frac{1}{N}\sum_i^N (W_K \cdot (\epsilon_K^{(i)} - \epsilon_K^{(i-n)}) + \epsilon_{K-1}^{(i)} - \epsilon_{K-1}^{(i-n)}) \cdot Y_{K-2}^{(i)}. \tag{11}$$

[1]For brevity, we omit the bias terms and activation functions

Restructuring the sums,

$$\begin{aligned}&= G_{K-1} + \frac{1}{N}\sum_i^{N-n} (W_K \cdot \epsilon_K^{(i)} + \epsilon_{K-1}^{(i)}) \cdot (Y_{K-2}^{(i)} - Y_{K-2}^{(i+n)}) \\ &+ \frac{1}{N}\sum_i^{n-1} \Big\{ W_K (\epsilon_K^{(N-i)} \cdot Y_{K-2}^{(N-i)} - \epsilon_K^{(i-n)} \cdot Y_{K-2}^{(i)}) \\ &+ \epsilon_{K-1}^{(N-i)} \cdot Y_{K-2}^{(N-i)} - \epsilon_{K-1}^{(i-n)} \cdot Y_{K-2}^{(i)} \Big\}\end{aligned}$$

When $N \gg n$, the last term amortizes to yield

$$\approx G_{K-1} + \frac{1}{N}\sum_i^{N-n} (W_K \cdot \epsilon_K^{(i)} + \epsilon_{K-1}^{(i)}) \cdot (Y_{K-2}^{(i)} - Y_{K-2}^{(i+n)}). \tag{12}$$

For G_{K-1}^* to correctly approximate G_{K-1}, the second term should be close to zero. It is possible when one of the following two conditions suffices. First,

$$\epsilon_k^{(i)} \approx \epsilon_k^{(i-n)} \qquad \forall k, i, \tag{13}$$

is derived from Eq. (11), which indicates the compression errors are always similar. Second,

$$\begin{aligned}\epsilon_k^{(i)} \perp\!\!\!\perp (Y_k^{(i)} - Y_k^{(i+n)}) \quad \forall k, i, \\ Avg(\epsilon_k^{(i)}) = 0, Avg(Y_k^{(i)} - Y_k^{(i+n)}) = 0 \quad \forall k, i,\end{aligned} \tag{14}$$

is derived from Eq. (12).

The first condition (Eq. (13)) is intuitively difficult to be true, because it requires all errors to be almost equivalent to each other in their values. Moreover, if it is true, compressing the forward activations with lazy error propagation should also work. However, in our experiments, compressing the forward inter-stage activation yielded divergence in the model, even with a very low compression rate.

On the other hand, the second condition (Eq. (14)) can be easily true, especially in the existence of batch/input normalization, which makes the average zero. In Section 9.3, we will show that the second condition (Eq. (14)) is empirically true. In practice, the error from the last micro-batch is lazily propagated in the first micro-batch of the last minibatch which further reduces the discrepancy. Finally, please note that different k values can be chosen to derive the same conditions, which we omit for brevity.

？ ？ ？

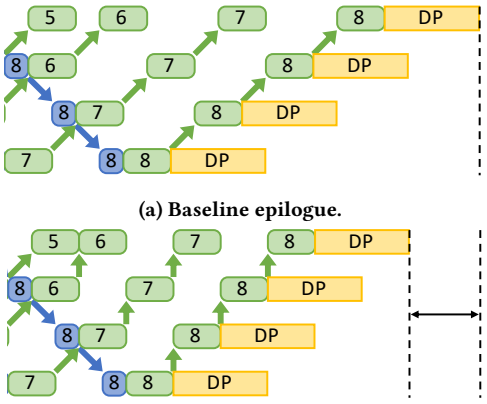
5.2 Epilogue-Only Compression

Epilogue-only compression is an additional technique in compressed backpropagation that provides better model quality without sacrificing the training speed. While compression techniques dramatically reduce the communication time, too much compression always has a risk of a drop in the model quality, which essentially becomes a trade-off between training speed and model quality.

Surprisingly, with pipeline parallelism, we can choose to compress the data that lie on the critical path. Under the baseline scheduling shown in Fig. 4a, many communications from inter-stage backpropagation are almost well-overlapped (hidden) with the computations. However, the communications from stages that lie on the critical path (called *epilogue*) are not hidden by the computations, as depicted in Fig. 6a.

With the above observation, epilogue-only compression only compresses the communication in the epilogue part, as illustrated in Fig. 6b. This causes less error in the training, and provides better quality to the model training. As shown in Fig. 6b, epilogue-only

lazy可以抵消压缩的损失



(a) Baseline epilogue.

(b) Compressed epilogue.

Figure 6: Illustration of epilogue-only compression.

compression does not reduce the speedup from the inter-stage communication compression when the inter-stage communication time is less than that of the corresponding stage's backward pass processing time. We found that this is true for high-end interconnects (i.e., ≥ 100Gbps), and find the epilogue-only compression method useful.

只压缩最后的也有加速

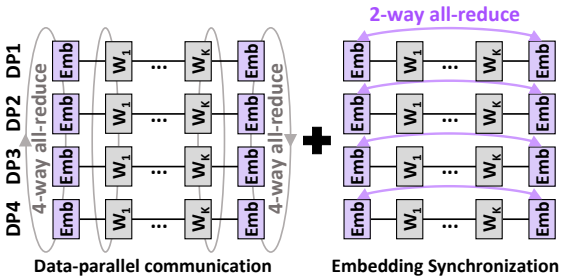
6 FUSED EMBEDDING SYNCHRONIZATION

- **Compression target**: Pipeline parallelism — embedding synchronization.
- **Interacts with**: Data parallelism.
- **Method**: Fuse two all-reduce communications of the shared embedding layer into a single all-reduce.

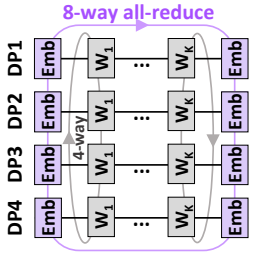
When any of the weight parameters in the model are used multiple times in the network, the parameters aggregate gradients from multiple paths. If the multiple paths lie within a single GPU, this does not incur any issue. However, if they are executed on different GPUs (either with tensor parallelism or pipeline parallelism), they require synchronization of the parameter gradients, which is another type of communication.

In large NLP models that we primarily target, such a structure commonly appears with the embedding layers as depicted with purple boxes in Fig. 7. If the output of the network takes a text format (as in the pretraining phase), the embedding layer is used twice: Once at the input for converting words into embedding values, and once more for converting the output vector into words. Because they are at the beginning and at the end of the model, they always generate an inter-node communication if pipeline parallelism is used. In existing solutions shown in Fig. 7a, the embedding layer is duplicated in both the first and last stages. After the layer-wise gradients are collected from data parallel communication, the duplicated embedding layers share the gradients in an all-reduce pattern in a separate communication phase.

Effectively, the functionality of the embedding synchronization is to share the gradients, identical to that of the data-parallel communication (i.e., all-reduce). Thus, we can fuse the two all-reduce



(a) Baseline. Two all-reduce communications.



(b) Fused embedding synchronization.

Figure 7: Illustration of fused embedding synchronization.

communications associated with the embedding layers (one from the data-parallel ways and the other from embedding synchronization) into a single all-reduce as illustrated in Fig. 7b. It is known that, for R ranks participating in an all-reduce communication for communication volume V, the cost is $2V \cdot (R-1)/R$ [72]. Because the number of ranks for embedding synchronization is always two, the conventional cost for the embedding layer C_{Emb} becomes

$$C_{Emb} = 2V \cdot \frac{D-1}{D} + 2V \cdot \frac{1}{2} = V \cdot \frac{3D-2}{D}, \tag{15}$$

where D is the number of data-parallel groups. With fused embedding synchronization, the number of ranks for the fused synchronization becomes $2 \cdot D$, and the cost becomes

$$C_{Emb\_fused} = V \cdot \frac{2D-1}{D}. \tag{16}$$

Thus, as D becomes large, the improvement approaches 50% over the baseline embedding synchronization time. For $D = 4$ used in our settings, the theoretical benefit already reaches 42.9%.

7 SELECTIVE STAGE COMPRESSION

- **Compression target**: Data parallelism.
- **Interacts with**: Pipeline parallelism.
- **Method**: Compress the current bottleneck stages for the trade-off between speed and the model quality.

In Optimus-CC, we provide selective stage compression as an optional technique to obtain a trade-off between model quality and speed when compressing data-parallel traffic. As shown in Fig. 3, compression of data-parallel traffic results in a big drop in the model quality, making the technique unacceptable despite its large benefit in the execution time. Even with a low compression rate, the drop is

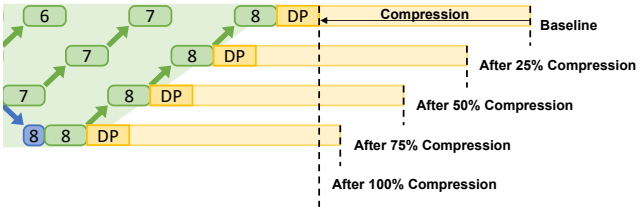


Figure 8: Illustration of selective stage compression.

still severe. We believe the reason comes from the weight staleness issue [29]. While the error from compression can be sent to the next iteration [12, 75], unlike compressed backpropagation, the error is applied after the weight update, and has a stale effect on the weight.

Instead of relying on the compression rate, selective stage compression provides a better knob by considering the pipeline schedules into account for the stage selection. In conventional data parallel-only distributed training, the gradient exchange communication for data parallelism happens after the backward propagation, and lies on the critical path. However, because of pipeline parallelism, the later stages finish early from the backpropagation and they can start the communication before others. Then, selectively compressing the stages, starting from the earlier stages, can adjust how much data we compress and therefore how much quality we lose. For example, in Fig. 8, the rightmost edge of the light-yellow box on stage 1 represents the finishing time for uncompressed data-parallel communication. When the first stage is compressed, the new bottleneck becomes stage 2, and further compressing the traffic from stage 2 yields the new bottleneck to stage 3. This scheme provides a much better trade-off, as we will demonstrate in Section 9.4.

8 IMPLEMENTATION

Optimus-CC has been implemented over the publicly available Megatron-LM code [9]. The interleaved pipeline scheduling [65] has been applied to reduce the scheduling bubbles. For the low-rank approximation, we adopted PowerSGD [75] implementation. We used the low-rank approximation for both compressed backpropagation and data-parallel gradient compression, because top-k methods are not suitable for point-to-point communications as shown in Section 3, and [75] shows good performance compared to other compression methods [1]. All of our additional implementations have been made using native PyTorch [53] 1.8 APIs, such that it does not require external libraries or recompilation of PyTorch.

For the compressed backpropagation, we wrote a custom low-rank compression code to support point-to-point communication, and integrated it into the methods of Megatron-LM's `p2p_communications.py`. To realize lazy error propagation, private variables were declared in the `PowerSVD` class to store the errors between micro-batches. For epilogue-only compression, `schedule.py` was modified to apply compression on the epilogue part of the communications. For the selective stage compression, we inherited the `DistributedDataParallel` class and overrode the `allreduce_gradients()` method. For the fused embedding synchronization, we again modified the `allreduce_gradients()` function. We detected the embedding layer by searching for the word *word_embeddings* in the name of the layer, and replaced the communication with the custom method.

Table 1: Experimental Environment

HW	Server Node	#Nodes	16
		CPU	2×EPYC 7543, 32 cores
		Memory	1TB DDR4 ECC
		GPU	8× Nvidia A100
	Interconnect	Intra-node	NVLink (600GBps / GPU)
		Inter-node	Infiniband HDR (200Gbps)
Model	Common	Micro-batch	8
		Total mini-batch	512
		#iterations	230K
	GPT-8.3B	#layers	72
		Hidden dim.	3072
		Ways	TP8 / DP4 / PP4
	GPT-2.5B	#layers	52
		Hidden dim.	1920
		Ways	TP8 / DP4 / PP4

9 EVALUATION

9.1 Experimental Environment and Method

We conducted our experiments based on the environments listed in Table 1. All experiments have been performed under a fixed set of nodes, such that unexpected variations could be minimized. Following [65], we pretrained GPT with 8.3B parameters (GPT-8.3B) and GPT with 2.5B parameters (GPT-2.5B) model to figure out the convergence of the model when using the proposed methods. The GPT-8.3B has 72 layers and a hidden dimension size of 3072, while the GPT-2.5B model has 52 layers, with a hidden dimension size of 1920. To utilize 128 GPUs, both models had eight tensor parallel groups that match the number of GPUs within a node, four data-parallel groups, and four pipeline stages unless otherwise stated. We pretrained models for 230K iterations, where the baseline model reaches the LAMBADA [52] task accuracy reported in [65]. All perplexity data are the result of training for 230K iterations unless otherwise stated.

Following [65], we executed pretraining of chosen NLP models with RealNews [86], Wikipedia [19], CC-stories [73] and OpenWebtext [56] datasets. We concatenated all these datasets and created a corpus. The datasets were preprocessed using the original Megatron-LM code [9], including the elimination of short documents and deduplication. For validation metrics, we also followed [65] (using 5% of the dataset as a validation set) including the holdout, splitting documents into training and validation at the beginning.

For the number of ranks in low-rank approximation-based compression, we used 128 for data-parallel gradient compression and 16 for compressed backpropagation unless otherwise stated, following the settings of the transformer-based model (around 10× compression) in [75]. We followed [75] because compression algorithms are well-studied research areas. We empirically chose 75% stage compression for selective stage compression. We ran 30K of warm-up iterations for all models, also following the practice from [75].

Table 2: Pretraining (230K iterations) training time speedup and validation set perplexity using 128 GPUs.

		Baseline	CB (Speedup)	CB+FE (Speedup)	CB+FE+SC (Speedup)
GPT-8.3B	Training Time	37.27 days	34.83 days (+7.01%)	32.84 days (+13.49%)	**25.72 days (+44.91%)**
	Val. Perplexity	**8.10**	**8.10**	**8.10**	8.20
GPT-2.5B	Training Time	14.72 days	13.63 days (+8.00%)	12.79 days (+15.09%)	**12.55 days (+17.29%)**
	Val. Perplexity	**9.31**	**9.31**	**9.31**	9.55

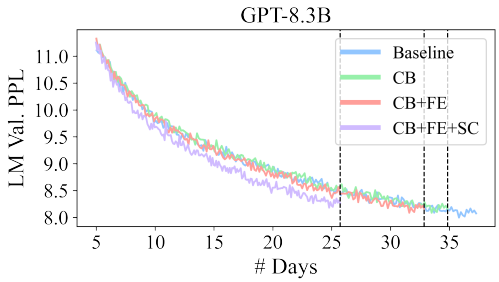


Figure 9: Pretraining validation LM perplexity of the proposed methods and the baseline.

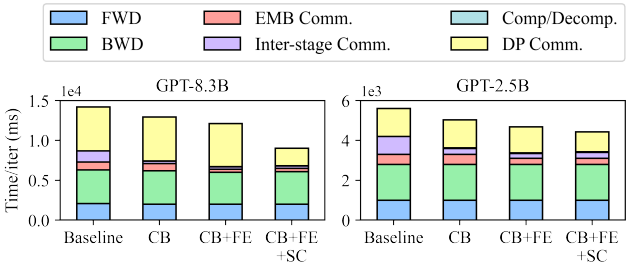


Figure 10: Breakdown of the execution times using 128 GPUs, in ablation of the proposed techniques.

9.2 Training Performance

Table 2 show the training speed, and the validation perplexity of Optimus-CC, on the chosen models. In the table, 'Baseline' refers to the original Megatron-LM without any communication compression. In addition, 'CB' refers to compressed backpropagation where the lazy error propagation and epilogue-only compression are used together, 'FE' refers to fused embedding synchronization, and 'SC' refers to selective stage compression. On an 8.3B parameter model, Optimus-CC achieves 44.91% speedup over the baseline (no compression) with marginal perplexity increase on CB+FE+SC, or 13.49% speedup without compromising perplexity on CB+FE. A similar trend can be seen from the 2.5B model, with a 17.29% speedup with a small perplexity increase on CB+FE+SC or a 15.09% speedup without a perplexity increase on CB+FE.

One interesting trend is the relatively larger speedup of SC in the 8.3B model than in the 2.5B model. Because the same number of GPUs are used, so the number of parameters per GPU, which affects the data-parallel gradient communication volume, becomes larger in the 8.3B model. It increases the portion of the communication over the inter-stage communication. Therefore, the speedup from compressing the data parallel communication in 8.3B becomes relatively larger than the 2.5B case.

Fig. 9 shows the curves of validation perplexity over the training of the 8.3B model. With the use of compressed backpropagation (CB) and fused embedding synchronization (FE), the perplexity remains mostly the same compared to the baseline, and sometimes even performs better depending on which iteration the perplexity is measured. This is because compressed backpropagation successfully restricts the impact of compression errors to be resolved within the same iteration, especially using lazy error propagation. Fused embedding synchronization does not induce any mathematical changes to the baseline, and thus it is guaranteed to have no perplexity increase. Selective stage compression provides a large amount of speedup at the cost of some perplexity trade-off. Even with the error feedback techniques [75], it is inevitable that the error is applied after the update, which causes the staleness effect. Nonetheless, selective stage compression controls the perplexity to have a marginal increase, while providing the most speedup.

We also conducted a language model validation of five zero-shot downstream tasks to validate our work based on [24], as shown in Table 3. Zero-shot tasks directly evaluate the pretrained model on some tasks (e.g., QnA) without fine-tuning and we used them to represent the expressibility of a model. For both the 8.3B and 2.5B models, CB and CB+FE show comparable accuracy on the baseline (no compression) model. CB+FE+SC shows marginal accuracy degradation compared to baseline, which aligns with the trend of Table 2 and Fig. 9.

Fig. 10 shows the breakdown measured in the same way as in Section 3. As depicted in the purple bars, compressed backpropagation (CB) reduces most of the backward inter-stage communications, by 78.57% compared to the baseline. Much of the inter-stage communication is left uncompressed because of epilogue-only compression, but they are overlapped by the other computational stages and do not affect the training time. Some portion of inter-stage communication that still remains in the stack accounts for the forward traffic which is not the target of compressed backpropagation. The red bars represent the embedding synchronization part. The reduction is about 40%, which is almost identical to the analytic cost model provided in Section 6 of 42.9%. When all the proposed methods have been applied (CB+FE+SC), the total communication time overhead has been reduced by 63.29% in the 8.3B model, showing the effectiveness of Optimus-CC. Note that the compression and decompression overhead is negligible due to the extremely high throughput of the compression algorithm, which we will demonstrate in Section 9.6.

Table 3: Accuracies on zero-shot tasks which indicates the expressibility of pretrained models.

Tasks	GPT-8.3B				GPT-2.5B			
	Baseline	CB	CB+FE	CB+FE+SC	Baseline	CB	CB+FE	CB+FE+SC
LAMBADA [52]	**66.82%**	66.35%	66.35%	65.79%	**62.00%**	61.93%	61.93%	61.15%
PIQA [7]	**75.52%**	74.05%	74.05%	74.27%	71.76%	**72.63%**	**72.63%**	71.93%
MathQA [3]	**24.36%**	23.55%	23.55%	23.52%	24.15%	**24.25%**	**24.25%**	23.42%
WinoGrande [63]	63.22%	**63.30%**	**63.30%**	63.22%	**62.19%**	60.62%	60.62%	61.33%
RACE [37]	**37.89%**	**37.89%**	**37.89%**	37.32%	33.88%	**35.12%**	**35.12%**	34.64%

Table 4: Effect of lazy error propagation on accuracies of zero-shot tasks in GPT-2.5B.

Tasks	Baseline	CB (Non-LEP)	CB (LEP)
LAMBADA	**62.00%**	61.79%	61.93%
PIQA	71.76%	71.87%	**72.63%**
MathQA	24.15%	23.69%	**24.25%**
WinoGrande	**62.19%**	59.75%	60.62%
RACE	33.88%	33.59%	**35.12%**

9.3 Analysis of Compressed Backpropagation

Table 4 shows the effect of lazy error propagation on the model quality. 'CB (Non-LEP)' refers to compressed backpropagation without lazy error propagation, and 'CB (LEP)' refers to compressed backpropagation with lazy error propagation. Epilogue-only compression was applied to all the cases because CB without epilogue-only compression diverged. Bold accuracy is the highest, and underlined accuracy is the lowest. While applying compression to the backpropagation without lazy error propagation severely damages the model quality, which brings out the lowest accuracies, applying lazy error propagation makes the model quality comparable to the baseline non-compressed model.

Fig. 11 depicts how the conditions from Eq. (14) hold during training. The green curves represent the average values for $\epsilon^{(i)}$ over 150 micro-batches during training. In addition, the purple curves show that the average of the difference between activations ($Y^{(i)} - Y^{(i+n)}$) is also near zero. Finally, the cosine similarity between $\epsilon^{(i)}$ and $Y^{(i)} - Y^{(i+n)}$ mostly stays around zero, which indicates that the two terms are independent. This suffices that the conditions from Eq. (14) are true, leading G^* in Eq. (10) to correctly approximate Eq. (7).

Fig. 12 shows the memory overhead of compressed backpropagation by plotting peak memory consumption reported by PyTorch. To apply compression [75], a separate memory region has to be allocated for low-rank matrices. This accounts for the 5-10% overhead to the baseline. In addition, lazy error propagation requires small additional memory for storing the error between micro-batches, but the overhead is marginal, which adds only 1% additional overhead.

9.4 Analysis of Selective Stage Compression

One might wonder if the compression ratio (i.e., ranks) of the low-rank approximation can be adjusted instead of applying selective stage compression. One important aspect is that the critical path cannot be considered by merely adjusting the compression ratio.

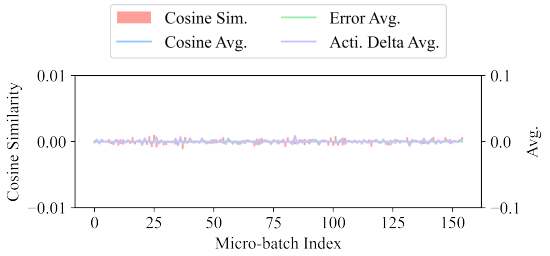


Figure 11: Cosine similarity of errors and activation differences.

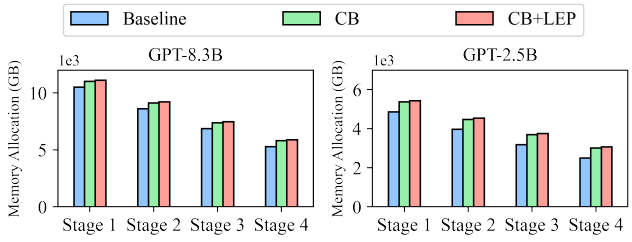


Figure 12: Maximum memory allocation per GPU of base compressed backpropagation and with lazy error propagation.

In this section, we show that selective stage compression provides a much better trade-off between the training speed and the model quality (measured with validation perplexity) on GPT-2.5B.

Fig. 13 plots the training time and the perplexity of the two methods. In the left figure, we apply selective stage compression and vary the percentage of stages being compressed. With selective stage compression, we achieve a reasonable trade-off between the speedup and validation perplexity. On the other hand, in the middle figure, we plot how the speedup and validation change by merely adjusting the rank used in the compression (compression ratio). Surprisingly, the relation between rank and perplexity is non-linear, which makes the traditional rank-adjusting infeasible as a tuning method. At rank 512 which translates to about 10× compression rate, both the perplexity and speedup significantly worsen. The reason for the speed degradation comes from the compression algorithm, where a too-high-valued rank will increase the time overhead for compression and decompression, as illustrated in Section 9.6. Thus, relying on the compression ratio for the speed-accuracy trade-off would not be a rational choice.

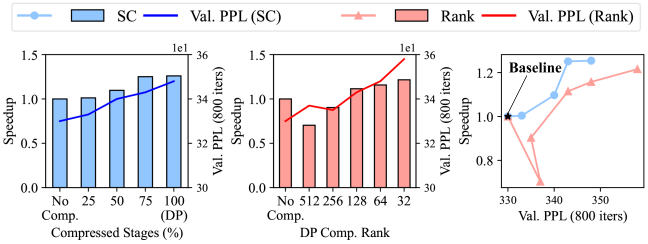


Figure 13: Effect of applying selective stage compression (left) and adjusting ranks (middle) to data-parallel communication on the speedup and the validation perplexity in GPT-2.5B.

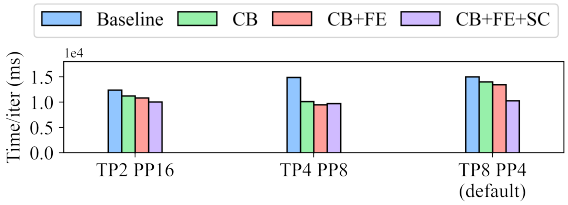


Figure 14: Tensor/pipeline-parallel configuration sensitivity of training time with the fixed data-parallel setting on GPT-9.2B.

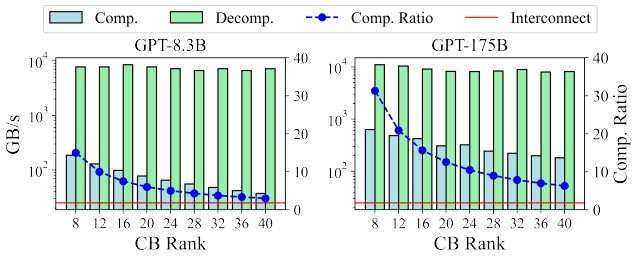


Figure 15: Throughput of inter-stage compression and decompression on GPT-8.3B (left) and GPT-175B (right).

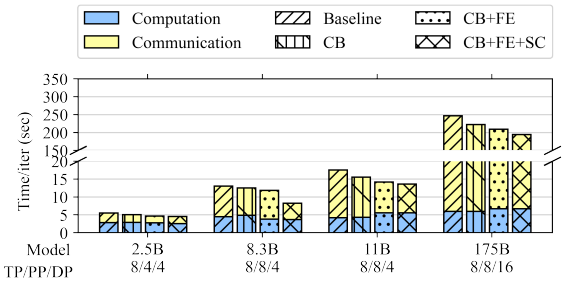


Figure 16: Scalability of our mechanisms.

Fig. 13 (right) shows a direct comparison between selective stage compression and adjusting ranks by plotting the validation perplexity and the speedup together. Considering that the upper-left (higher speedup and lower PPL) is the optimal direction, selective stage compression always provides a better trade-off than adjusting ranks. We believe an even better trade-off can be achieved by automatically choosing the right combination of the compression rank and the number of stages for selective stage compression, which we leave as future work.

9.5 Analysis on Configuration Sensitivity

Finding the best tensor/pipeline parallel configuration for training is an active research area [71]. Fig. 14 shows the training time of models on various parallel configurations. We fixed the number of data-parallel ways to four for a fair comparison and conducted experiments on possible configuration settings. We tested the configurations up to eight tensor-parallel ways because tensor-parallel ways are generally limited to the number of GPUs in a node. With 128 GPUs, this results in the number of pipeline ways from 16 to 4. To evenly divide the layers up to 16 stages for a fair comparison, we increased the number of layers to 80, which corresponds to 9.2B parameters. For selective stage compression, we used the same 75% compression for all settings. Optimus-CC provides at least 19.2% speedup for all parallel configurations. The trend is that CB has more advantage when the number of pipeline-parallel ways increases because this incurs more inter-stage communication from deeper stages in the pipeline. On the other hand, SC takes advantage as the number of pipeline-parallel ways decreases. This is because, with less number of stages, the number of parameters per GPU increases, and thus the portion of data-parallel communication becomes larger.

9.6 Analysis on Compression/Decompression Throughput

Optimus-CC uses a compression algorithm as its key component, and thus analyzing the compression and decompression throughput is critical. In Fig. 15, we show that the compression and decompression throughput are much higher than that of the interconnect bandwidth. In CB rank 16 of the 8.3B model, the compression throughput is 786.96Gbps (98.37GB/s), and the decompression throughput is 68.2Tbps (8.32TB/s), which has enough gap with the interconnection throughput of 200Gbps (25GB/s) depicted in red lines.

The compression throughput becomes higher in larger model sizes because constant setup overheads for the compression kernels become amortized with larger data. An interesting and rather counter-intuitive trend is that the throughput decreases with higher CB ranks (less compression). This is because the orthogonalization phase is the main bottleneck (about 80%) for the compression algorithm, which takes longer with a larger output size.

9.7 Analysis on Scalability

Fig. 16 shows the scalability of the proposed work with four Megatron-LM [65] based models. We fixed tensor-parallel-ways to 8 and increased the number of GPUs in larger models for a fair comparison. Optimus-CC scales well on larger model sizes even when the model grows up to 175B (GPT-3) [8].

The scalability comes from two factors. First, it is well-known that larger models suffer more from communication overheads [66];

there is more potential for the proposed work. Second, as shown in Section 9.6, the compression itself becomes more efficient when the model size becomes larger. The compression overhead was already small for 2.5B and 8.3B models, but becomes even smaller with extremely larger models.

10 DISCUSSION

10.1 Application on Other Accelerators

Aside from GPUs, we can use other DL accelerators for training large-scale models, such as TPU [34] and IPU [26]. They generally have higher computational throughputs and intra-node bandwidth than GPUs, and their inter-node speed (400Gbps for TPU and 100Gbps for IPU) is similar to that of GPUs. They also use 3D parallelism for large-scale training, which requires inter-node communication.

Optimus-CC will have more potential on these accelerators because computational throughput over inter-node bandwidth is larger than our setting. For an example of IPU-POD128, it provides 8 PFLOPS per node, while our setting provides 5 PFLOPS per node. However, its inter-node communication is 100Gbps, which is half the bandwidth of our setting. In such a case, Optimus-CC will provide more advantages.

10.2 Application on Other Models

While *3D Parallelism* is widely used to train NLP models, it can also be applied to other domains (e.g., CNN). Optimus-CC can be adopted to these models, because the mechanisms of the proposed techniques are independent to a model structure.

For example, training of AmoebaNet [61] is often done with 3D Parallelism [29] to mitigate the problem of increased model size and training time. However, the parallelism makes the inter-stage and data-parallel gradient communication overhead significant. Gradient compression methods can reduce the overhead of the data-parallel gradient communication, but suffer from an accuracy degradation problem. In this circumstance, Optimus-CC can minimize the accuracy degradation by *selective stage compression* and further accelerate the training by compressing the inter-stage communication through *compressed backpropagation*.

In fact, we believe Optimus-CC can be applied to any DNN domain that requires a model larger than a single device. For example, modern graph neural networks started adopting data parallelism [32, 80] and pipeline parallelism [76]. Optimus-CC could be applied to such cases to bring a speedup.

11 RELATED WORK

11.1 Data Parallelism

To cope with the growing size of the large-scale models [19, 69], plenty of methods were proposed to accelerate the training procedure. *Data parallelism* [16, 18, 25, 88] has been commonly used to accelerate model training in a distributed manner. Data parallelism is to copy the entire model to every worker and train with a different mini-batch while keeping an identical weight among all the workers. In order to keep exactly the same state after every step, data parallelism necessitates the synchronization of gradients. Its communication overhead grows linearly proportional to the model size which makes all-reduce [25, 72] be the main bottleneck of training. To mitigate this problem, many researchers tried to optimize communication itself [14, 38, 46, 77, 87, 89] while maximizing the overlap between communication and computation [27, 30, 31, 54].

Another effective approach to reducing communication overhead in data parallelism is *gradient compression*. By using a low-rank approximation of gradient matrices [15, 75, 78], sparse gradient update methods [6, 12, 44, 45, 64] and gradient quantization [4, 22, 85], communication volume can be effectively reduced without significant loss in accuracy. To prevent further performance degradation caused by gradient compression, most works adopt *error-feedback* which is to compensate for the difference between the compressed gradient and the original one. Recently, statistical ways [28, 55, 82] can be alternatively used to avoid additional computation overhead when conducting compression methods.

11.2 Tensor and Pipeline Parallelism

The size of transformer-based language models [8, 19, 58, 74] has been grown at an exponential rate [50]. This trend hits on limited accelerator memory capacity in addition to proportional growth of training time [8]. However, the aforementioned data parallelism cannot handle both issues by scaling larger batch size [43]. Tensor and pipeline parallelism has tried to handle both issues in their own ways.

Tensor parallelism [20, 33, 71, 79] is to hand out parameters (tensor) in the same layer to multiple workers. Tensor parallelism is mainly focused on reducing the number of synchronization points among workers sharing the same layer. On the other hand, *pipeline parallelism* [29] schedules the execution of *micro-batches* which are sampled from a mini-batch. A primary constraint of pipelined schedule is *synchronous execution* which is essential regulation in the model training process. The activation calculated in the forward pass has to be used in the corresponding backward pass. In such regard, recent works on pipeline parallelism have mainly focused on reducing memory overhead [59, 60, 62] for handling *staleness problem* [35, 83] while reducing *pipeline bubble* [23, 40, 49] by optimizing pipelined schedule.

Many transformer-based models can be successfully trained by using a combination of three distributed training methods which is called *3D parallelism* [50, 51, 65, 68]. However, to the extent of our knowledge, no attempt has been made to apply 3D parallelism-aware communication compression to a large language model. Before Optimus-CC, the effect of compressed communication in 3D parallelism has been unknown space.

12 CONCLUSION

In this work, we proposed Optimus-CC, which compresses the communications of large, distributed NLP models that utilize 3D parallelism. Because the conventional communication compression algorithms fail to exploit pipeline-related opportunities and result in a model quality drop, we proposed multiple techniques that reduce the amount of communications while maintaining the model quality. We believe the impact of Optimus-CC will be more significant with even larger models, adding value to the work in the upcoming future.

ACKNOWLEDGMENTS

This work was supported by Samsung Electronics Co., Ltd (IO210809-08878-01), the National Research Foundation of Korea (NRF) grants (2022R1C1C1011307, 2022R1C1C1008131) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2021-0-00853, 2020-0-01361). Jaeyong Song and Youngsok Kim are with the Department of Computer Science at Yonsei University and are partly supported by the BK21 FOUR (Fostering Outstanding Universities for Research) funded by the Ministry of Education (MOE) of Korea and National Research Foundation (NRF) of Korea.

DATA AVAILABILITY STATEMENT

The artifact of Optimus-CC is available at [67]. It contains NLP dataset generation code, [75]-based compression code, [9]-based training code and [24]-based evaluation code.

REFERENCES

[1] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2022. On the Utility of Gradient Compression in Distributed Training Systems. In *MLSys*.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely Large Minibatch SGD: Training Resnet-50 on Imagenet in 15 Minutes. *arXiv preprint arXiv:1711.04325* (2017).

[3] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. In *NAACL*. 2357–2367.

[4] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. 2019. Qsparse-local-SGD: Distributed SGD with Quantization, Sparsification and Local Computations. In *NeurIPS*.

[5] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. 2018. signSGD: compressed optimisation for non-convex problems. In *ICML*.

[6] Jeremy Bernstein, Jiawei Zhao, Kamyar Azizzadenesheli, and Anima Anandkumar. 2018. SignSGD with Majority Vote Is Communication Efficient and Fault Tolerant. *arXiv preprint arXiv:1810.05291* (2018).

[7] Yonatan Bisk, Rowan Zellers, Ronan Le bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about Physical Commonsense in Natural Language. In *AAAI*.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *NeurIPS*.

[9] Jared Casper, Mostofa Patwary, Boris Fomitchev, Evelina, lmcafee nvidia, Raul Puri, Nako Sung, Stas Bekman, Akhilesh Gotmare, David E. Weekly, Deepak Narayanan, Devrim, Heungsub Lee, and Kazuhiro Yamasaki. 2021. *NVIDIA/Megatron-LM: v2.5.* https://doi.org/10.5281/zenodo.5181820

[10] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-gpu Platform. *arXiv preprint arXiv:1809.02839* (2018).

[11] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. In *AAAI*.

[12] Chia-Yu Chen, Jiamin Ni, Songtao Lu, Xiaodong Cui, Pin-Yu Chen, Xiao Sun, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Wei Zhang, and Kailash Gopalakrishnan. 2020. ScaleCom: Scalable Sparsified Gradient Compression for Communication-Efficient Distributed Training. In *NeurIPS*.

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

[14] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *MLSys*.

[15] Minsik Cho, Vinod Muthusamy, Brad Nemanich, and Ruchir Puri. 2019. GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning. In *NeurIPS*.

[16] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. 2017. Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291* (2017).

[17] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent. *arXiv preprint arXiv:1602.06709* (2016).

[18] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *NeurIPS*.

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

[20] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and Filter Parallelism for Large-Scale CNN Training. In *SC*.

[21] P. G. Emma. 1997. Understanding Some Simple Processor-performance Limits. *IBM Journal of Research and Development* (1997), 215–232.

[22] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. 2020. Adaptive Gradient Quantization for Data-Parallel SGD. In *NeurIPS*.

[23] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *PPoPP*.

[24] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2021. *A framework for few-shot language model evaluation.* https://doi.org/10.5281/zenodo.5371628

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

[26] GRAPHCORE. 2022. IPU. https://www.graphcore.ai/products, visited 2022-10-31.

[27] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *MLSys*.

[28] Samuel Horváth and Peter Richtarik. 2021. A Better Alternative to Error Feedback for Communication-Efficient Distributed Learning. In *ICLR*.

[29] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *NeurIPS*.

[30] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *ASPLOS*.

[31] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *MLSys*.

[32] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *MLSys*.

[33] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks.. In *MLSys*.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*.

[35] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Koster. 2021. Pipelined Backpropagation at Scale: Training Large Models Without Batches. In *MLSys*.

[36] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997* (2014).

[37] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683* (2017).

[38] Jinho Lee, Inseok Hwang, Soham Shah, and Minsik Cho. 2020. FlexReduce: Flexible All-reduce for Distributed Deep Learning on Asymmetric Network Topology.

In *DAC*.
[39] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*.
[40] Shigang Li and Torsten Hoefler. 2021. Chimera: Efficiently Training Large-scale Neural Networks with Bidirectional Pipelines. In *SC*.
[41] Shigang Li and Torsten Hoefler. 2022. Near-Optimal Sparse Allreduce for Distributed Deep Learning. In *PPoPP*.
[42] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *VLDB Endowment* (2020), 3005–3018.
[43] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. 2020. Don't Use Large Mini-batches, Use Local SGD. In *ICLR*.
[44] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*.
[45] Ahmed M. Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An Efficient Statistical-based Gradient Compression Technique for Distributed Training Systems. In *MLSys*.
[46] Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Olli Saarikivi, Tianju Xu, Vadim Eksarevskiy, Jaliya Ekanayake, and Emad Barsoum. 2021. Scaling Distributed Training with Adaptive Summation. In *MLSys*.
[47] Microsoft. 2020. Turing-nlg: A 17-billion-parameter Language Model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.
[48] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. 2018. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233* (2018).
[49] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*.
[50] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *SC*.
[51] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *IEEE Micro* (2019), 91–101.
[52] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031* (2016).
[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.
[54] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*.
[55] Xun Qian, Peter Richtárik, and Tong Zhang. 2021. Error Compensated Distributed SGD Can Be Accelerated. In *NeurIPS*.
[56] Alec Radford, Jeffrey Wu, Dario Amodei, Daniela Amodei, Jack Clark, Miles Brundage, and Ilya Sutskever. 2019. Better Language Models and Their Implications. *OpenAI blog* (2019).
[57] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models Are Unsupervised Multitask Learners. *OpenAI blog* (2019).
[58] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-text Transformer. *arXiv preprint arXiv:1910.10683* (2019).
[59] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC*.
[60] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *SC*.
[61] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*.
[62] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *USENIX ATC*.
[63] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. WinoGrande: An Adversarial Winograd Schema Challenge at Scale. *arXiv preprint arXiv:1907.10641* (2019).
[64] Shaohuai Shi, Xianhao Zhou, Shutao Song, Xingyao Wang, Zilin Zhu, Xue Huang, Xinan Jiang, Feihu Zhou, Zhenyu Guo, Liqiang Xie, Rui Lan, Xianbin Ouyang, Yan Zhang, Jieqian Wei, Jing Gong, Weiliang Lin, Ping Gao, Peng Meng, Xiaomin Xu, Chenyang Guo, Bo Yang, Zhibo Chen, Yongjian Wu, and Xiaowen Chu. 2021. Towards Scalable Distributed Training of Deep Learning on Public Cloud Clusters. In *MLSys*.
[65] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).
[66] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990* (2022).
[67] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Him, Youngsok Kim, and Jinho Lee. 2022. *jaeyong-song/Optimus-CC: v0.2.* https://doi.org/10.5281/zenodo.7220824
[68] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. In *HPCA*.
[69] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*.
[70] Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 2021. 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. In *ICML*.
[71] Jakub Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *NeurIPS*.
[72] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *IJHPCA* (2005), 49–66.
[73] Trieu H Trinh and Quoc V Le. 2018. A Simple Method for Commonsense Reasoning. *arXiv preprint arXiv:1806.02847* (2018).
[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *NeurIPS*.
[75] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. In *NeurIPS*.
[76] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *ICLR*.
[77] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*.
[78] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. 2021. Pufferfish: Communication-efficient Models At No Extra Cost. In *MLSys*.
[79] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2018. Unifying Data, Model and Hybrid Parallelism in Deep Learning Via Tensor Tiling. *arXiv preprint arXiv:1805.04170* (2018).
[80] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.
[81] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *NeurIPS*.
[82] An Xu, Zhouyuan Huo, and Heng Huang. 2021. Step-Ahead Error Feedback for Distributed Training with Compressed Gradient. In *AAAI*.
[83] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. 2021. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*.
[84] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training In Minutes. In *ICPP*.
[85] Yue Yu, Jiaxiang Wu, and Longbo Huang. 2019. Double Quantization for Communication-Efficient Distributed Optimization. In *NeurIPS*.
[86] Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. 2019. Defending Against Neural Fake News. In *NeurIPS*.
[87] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*.
[88] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep Learning with Elastic Averaging SGD. In *NeurIPS*.
[89] Qihua Zhou, Kun Wang, Haodong Lu, Wenyao Xu, Yanfei Sun, and Song Guo. 2021. Canary: Decentralized Distributed Deep Learning Via Gradient Sketch and Partition in Multi-Interface Networks. *IEEE TPDS* (2021), 900–917.

Received 2022-07-07; accepted 2022-09-22