Cupcake: A Compression Optimizer for Scalable Communication-Efficient Distributed Training

Zhuang Wang | Xinyu Crystal Wu | Zhaozhuo Xu | T. S. Eugene Ng

Abstract
Data-parallel distributed training (DDT) is the de facto way to accelerate deep learning on multiple GPUs. In DDT, communication for gradient synchronization is the major efficiency bottleneck. Many gradient compression (GC) algorithms have been proposed to address this communication bottleneck by reducing the amount of communicated data. Unfortunately, it has been observed that GC only achieves moderate performance improvement in DDT, or even harms the performance.

In this paper, we argue that the current way of deploying GC in a layer-wise fashion reduces communication time but at the cost of non-negligible compression overheads. To address this problem, we propose Cupcake, a compression optimizer to fully unleash GC algorithms’ advantages in accelerating DDT. It applies GC algorithms in a fusion fashion and determines the provably optimal fusion strategy to maximize the training throughput of compression-enabled DDT jobs. Experimental evaluations show that GC algorithms with Cupcake can achieve up to $2.03 \times$ speedup in training throughput over training without GC, and up to $1.79 \times$ speedup over the state-of-the-art approaches of applying GC to DDT in a layer-wise fashion.

1 Introduction
Deep Neural Networks (DNN) are gaining rapid popularity in various domains, such as computer vision (He et al., 2016; Szegedy et al., 2016; Simonyan & Zisserman, 2014) and natural language processing (NLP) (Devlin et al., 2018; Kiros et al., 2015; Manning et al., 2014). Because DNN models introduce million-scale weight tensors, large batch training over these giant tensors is infeasible due to the limited GPU memory resources.

To overcome this obstacle, data-parallel distributed training (DDT) has been widely adopted to accelerate the training of DNN models. DDT uses multiple GPUs and each GPU has a replica of the training model (Li et al., 2014). The training dataset is partitioned into multiple parts and each GPU only trains on its partition. DDT scales DNN training over a large number of GPUs to reduce the total training time (Sergeev & Del Balso, 2018; ByteDance, 2020; Li et al., 2020).

However, there exists an exacerbating tension between computation and communication in DDT. The computation time of DNN training has been dramatically reduced thanks to the recent advancements in GPU architectures (Luo et al., 2018; NVIDIA, 2021) and domain-specific compiler techniques (Chen et al., 2018; Zheng et al., 2020). This trend leads to more frequent gradient synchronization in DDT and puts higher pressure on the network connecting GPUs. However, it is difficult for the cloud network bandwidth to keep up with the pace of computation-related improvement. Moreover, the number of GPUs for DNN training keeps increasing due to the ever-growing training dataset, which further worsens the communication time (Jiang et al., 2020). Communication has become a well-known efficiency bottleneck in DDT as each GPU needs to transmit the full gradients for synchronization (Fei et al., 2021; Huang et al., 2019; Aji & Heafield, 2017; Lin et al., 2017).

Gradient compression (GC) is a promising approach to alleviate the communication bottleneck in DDT by significantly reducing the amount of communicated data. A plethora of GC algorithms (Strom, 2015; Lin et al., 2017; Tsuzuku et al., 2018; Alistarh et al., 2017; Seide et al., 2014; Aji & Heafield, 2017; Wen et al., 2017; Karimireddy et al., 2019) have been proposed recently and they can save up to 99.9% of the gradient exchange while preserving the training accuracy and convergence (Wu et al., 2018; Stich et al., 2018; Jiang & Agrawal, 2018; Wang et al., 2022).

However, it is challenging to achieve the desired speedup when applying GC to DDT jobs (Bai et al., 2021; Agarwal et al., 2022). In this paper, we first analyze the practical difficulty of compression-enabled DDT jobs. The state-of-
the-art approach for applying GC to DDT is in a layer-wise fashion, i.e., tensors are compressed one by one when they are ready for communication. It can greatly shorten the communication time for gradient synchronization because of the reduced amount of traffic volume. However, we surprisingly observed that the end-to-end training speedups of DDT with the layer-wise compression are only modest, and even worse than training without GC in many cases due to the incurred prohibitive overhead of compression operations (Xu et al. 2020; Wang et al. 2022).

We then propose Cupcake to maximize the training throughput of compression-enabled DDT by reducing the amount of communicated data and minimizing the compression overhead simultaneously. Cupcake is a general compression optimizer to enable GC algorithms to unleash their benefits to accelerate DDT. Essentially, GC reduces the communication time of DDT at the cost of the compression overheads. Because of the fixed overheads to launch and execute kernels in CUDA (Arafa et al. 2019), the compression overhead is non-negligible even for a small tensor size. Fortunately, we observe that this overhead keeps constant when the tensor size is smaller than a threshold (e.g., 4 MB in our testbed) and it then linearly increases with the tensor size. This observation motivates us to fuse multiple tensors for one operation.

Fusing tensors for compression leads to a trade-off between the reduced compression overhead and the communication overhead, i.e., the communication time that cannot overlap with computation. Because communication overlaps with computation in DDT (Zhang et al. 2017; Sergeev & Del Balso 2018; ByteDance 2020; Li et al. 2020), gradient tensors can begin their communications whenever they are ready in the layer-wise fashion. However, in a fusion fashion, a tensor has to wait for its following tensors for a unified compression operation and communication operation. Therefore, fusion can delay communications, shrink the overlapping time, and thus worsen the iteration time. To address this challenge, Cupcake determines the provably optimal fusion strategy for applying GC to DDT by balancing the compression overhead and the communication overhead. It can maximize the training throughput of compression-enabled DDT jobs, regardless of different training models, GC algorithms, and training system configurations, such as the number of GPUs and network bandwidth.

Our evaluations in both computer vision and NLP demonstrate that GC algorithms applied with Cupcake can greatly improve the training throughput of DDT. Specifically, Cupcake enables GC algorithms to achieve up to 2.03 × speedup in training throughput over training without GC, and up to 1.79 × speedup over the state-of-the-art solutions that compress tensors in a layer-wise fashion.

2 BACKGROUND

2.1 Data-parallel Distributed Training

Data-intensive training of DNNs on powerful Graphic Processing Units (GPU) (Owens et al. 2008) boosts the success of deep learning (LeCun et al. 2015). Given the massive amount of training dataset, data-parallel distributed training (DDT) (Shallue et al. 2019; Ben-Nun & Hoefler 2019; Li et al. 2020) has become one of the most popular paradigms to scale out deep learning with multiple GPUs. In this paradigm, the training dataset is partitioned into multiple subsets. Each GPU has a replica of the training model that trains on a specific subset. In each iteration, each GPU consumes a mini-batch from its allocated subset as the input of the training. Next, it propagates the mini-batch through the neural network model and calculates the loss function via forward propagation. Then, it uses the loss value to compute the gradients of each parameter in backpropagation. Finally, it synchronizes the gradient updates from all GPUs to update the model parameters with a certain optimizer, such as SGD (Zinkevich et al. 2010) or Adam (Kingma & Ba 2014). Training a DNN model is a process to refine the model parameters with the above steps iteratively until its convergence. Asynchronous DDT, where GPUs do not wait for the synchronized results to begin the next iteration, can hurt the model accuracy (Chen et al. 2016). In this paper, we focus on synchronous DDT because of its wide adoption (ten 2016; ByteDance 2020; Sergeev & Del Balso 2018; Li et al. 2020).

2.2 Communication Bottleneck in DDT

The single-GPU iteration time of DNN training jobs has been significantly reduced thanks to the advancement of DNN accelerators and domain-specific compiler techniques. For example, the iteration time of ResNet50 with one GPU has decreased by 22 × in the last six years (Sun et al. 2019). However, network upgrades have not kept up with the pace of computation-related advancements. The cloud network bandwidth has only witnessed a roughly 10 × increase in the same period (Mellanox 2022). This imbalance between the fast-growing computing capability and the slower-growing communication bandwidth worsens the communication-computation tense in DDT (Wang et al. 2023).

Single precision (FP32) is a common floating point format representing the weights and gradients in deep learning. When gradients are communicated in FP32 for synchronization, it can lead to costly communication time and thus poor scalability in DDT. It has been reported that the communication time for gradient synchronization accounts for over 60% of the total time for the training of BERT (Devlin et al. 2018) or other Transformer models across 16 AWS EC2 instances, each with 8 NVIDIA V100 GPUs, in a 100Gbps network (Bai et al. 2021; Wang et al. 2023).
When the network bandwidth in GPU clouds cannot keep pace with the improvements in computation, an alternative is to shrink the communicated data volume by applying gradient compression to DDT.

2.3 Gradient Compression Algorithms

Many gradient compression (GC) algorithms have been proposed recently to reduce the amount of communicated data volume for gradient synchronization. There are two main types of GC algorithms: Sparsification and Quantization. Sparsification selects a subset of the original stochastic gradients for synchronization (Wang et al., 2022; Aji & Heafield, 2017; Lin et al., 2017) and it can save up to 99.9% of the gradient exchange (Lin et al., 2017). Quantization decreases the precision of gradients. The gradients in FP32 are mapped to fewer bits, such as 8 bits (Dettmers, 2015), 2 bits (Wen et al., 2017), and even 1 bit (Seide et al., 2014; Karimireddy et al., 2019; Bernstein et al., 2018) to reduce the communicated traffic volume by up to 96.9%. Such compression algorithms have been theoretically proved and/or empirically validated to preserve the convergence of model training and impose negligible impact on model accuracy when combined with error-feedback mechanisms (Seide et al., 2014; Wu et al., 2018; Stich et al., 2018; Lin et al., 2017; Jiang & Agrawal, 2018). There are also other types of GC algorithms, such as low-rank decomposition (Vogels et al., 2019; Wang et al., 2018a) and FFT-based compression (Wang et al., 2020).

2.4 Overlapping Communication with Computation

Because of the layered structure and a layer-by-layer computation pattern in DNN models, the wait-free backpropagation mechanism (WFBP) (Zhang et al., 2017; Sergeev & Del Balso, 2018) is widely adopted to overlap communication with computation in DDT. As illustrated in Figures 1(a) and 1(b), WFBP can significantly reduce the iteration time compared to the strawman solution, in which communication cannot begin until the completion of backpropagation. Existing distributed ML frameworks, such as PyTorch (Paszke et al., 2019), Tensorflow (ten, 2016), and Horovod (Sergeev & Del Balso, 2018), apply GC to DDT in a layer-wise fashion, i.e., tensor by tensor, to overlap communication with computation because of WFBP.

3 The Practical Performance of GC with Layer-wise Compression

Because applying GC to DDT requires computation resources, it competes for GPU resources with backpropagation and delays tensor computation, as shown in Figure 1(c). Although GC algorithms can reduce the communication time of DDT, the incurred compression overheads can dramatically dilute the benefits gained from the reduced communication time.

To demonstrate, we empirically measure the training speeds of compression-enabled DDT with several popular GC algorithms, including both sparsification and quantization. The experiments are conducted on a server equipped with 8 GPUs (NVIDIA Tesla V100 with 32 GB memory), two 20-core/40-thread processors (Intel Xeon Gold 2.1GHz), and PCIe 3.0×16. We use GRACE (Xu et al., 2020) as the framework to support compression-enabled DDT and GC algorithms are applied in a layer-wise fashion. The training model is ResNet50 (He et al., 2016) over CIFAR10 (Krizhevsky et al., 2009) and the batch size is 32.

Two sparsification algorithms, DGC (Lin et al., 2017) and Rand-k (Stich et al., 2018), are evaluated and the gradient sparsity is 99%, i.e., only 1% of gradients are exchanged during synchronization. Two 1-bit quantization algorithms,
Cupcake: A Compression Optimizer for Scalable Communication-Efficient Distributed Training

EFSignSGD (Karimireddy et al., 2019) and OneBit (Seide et al., 2014) are also evaluated.

Figure 2 shows the breakdown of the iteration time of the training. The communication overhead refers to the communication time that cannot overlap with backpropagation and compression of any tensors. FP32 is the training baseline without GC. We observe that the performance improvement with these GC algorithms is modest. Some algorithms, such as DGC, EFSignSGD, and OneBit, even surprisingly lead to a longer iteration time. This observation is on par with the findings in prior works (Xu et al., 2020; Sapio et al., 2019; Li et al., 2018; Gupta et al., 2020; Agarwal et al., 2022).

3.1 The Root Cause of the Poor Performance

When a gradient tensor is ready for synchronization in DDT without compression, it is communicated and then the aggregated results are used to update the training model. However, there are two additional operations when applying GC to DDT: encoding (encode a tensor before communication to reduce the traffic volume) and decoding (decode the received compressed tensor for model updates). These two operations can incur non-negligible computation overhead.

Figure 2b and 2c display the encoding and decoding latencies with four representative GC algorithms, i.e., DGC (Lin et al., 2017) and Rand-k (Stich et al., 2018) for sparsification, EFSignSGD (Karimireddy et al., 2019) and Onebit (Seide et al., 2014) for quantization. Both encoding and decoding latencies are non-negligible, even for tensors with small sizes. For instance, the encoding latencies of DGC, EFSignSGD, and Onebit are all greater than 0.25 ms, regardless of the tensor sizes.

DNN models typically have a large number of tensors for gradient synchronization (He et al., 2016; Devlin et al., 2018). The layer-wise compression invokes encoding and decoding operations for each tensor and leads to prohibitive compression overheads. We take training ResNet50 over CIFAR10 with 8 GPUs in our testbed as a concrete example to compare the overall compression overhead against the communication improvement. In our measurement, the iteration time of the single-GPU training is around 48 ms. Without any compression, the communication overhead in each iteration is about 56 ms. Both sparsification and 1-bit quantization algorithms can reduce the communication overhead to less than 10 ms thanks to the much smaller communicated traffic volume. However, the overall compression overheads of DGC and EFSignSGD are both larger than 60 ms, which is even higher than the communication overhead in the baseline. The costly compression overheads result in the poor practical performance of DDT with GC.

3.2 An Opportunity and a Challenge

The compression overhead of GC algorithms with a layer-wise fashion becomes the new efficiency bottleneck in DDT. We observe that there are some fixed overheads to launch and execute kernels in CUDA (Arafa et al., 2019). Figure 2 shows that the encoding and decoding latencies of GC al-
There are numerous fusion strategies to apply a GC algorithm to a DDT job and meanwhile overlap communication with computation in a layer-wise fashion. It can reduce the compression overhead, as shown in Figure 3(a). Another extreme case is to apply the layer-wise fashion to compress tensors one by one to minimize the communication overhead, but it leads to prohibitive compression overheads, as discussed in Section 3.1.

There are numerous fusion strategies to apply a GC algorithm to a DDT job and three strategies are illustrated in Figure 4. It is challenging to find the optimal one because it depends on many factors, such as the applied GC algorithms, the tensor size and the computation time of the DNN model, the number of GPUs, and network bandwidth. We must jointly consider backpropagation, compression, and communication overheads to search for the optimal strategy to maximize the training throughput of compression-enabled DDT jobs.

4 Cupcake

In this section, we first formulate the tensor fusion problem given a DDT job and a GC algorithm. We then design an algorithm to provably find the optimal fusion strategy to minimize the iteration time.

4.1 Problem Formulation

The core idea of Cupcake is to fuse multiple tensors for one compression operation, instead of applying GC to DDT in a layer-wise fashion. It can reduce the compression overhead and meanwhile overlap communication with computation to reduce the communication overhead.

Given a training model with $N$ tensors, the set of tensors is $\mathcal{T} = \{T_0, \ldots, T_{N-1}\}$. For example, Figure 1 and Figure 3 display a training model with five tensors. Cupcake partitions the model into $y$ groups and determines a fusion strategy $\mathcal{X}_y = \{x_0, \ldots, x_{y-1}\}$, where $x_i$ is a group of consecutive tensors that are compressed and communicated together. Cupcake performs an encoding operation and a communication operation for each tensor group in each iteration. After encoding, the fused tensor $x_i$ is communicated and aggregated. After communication, the encoded $x_i$ is decoded and aggregated to update the training model.

Let $A$ denote the computation time for forward propagation in an iteration and $B(T_i)$ denote the computation time of $T_i$ in backpropagation. $x_i$ is the total tensor size of $x_i$, $h(x_i)$ is the time to compress $x_i$ and $g(x_i)$ is the corresponding communication time. $P(\mathcal{X}_y)$ is the total overlapping time, i.e., the total communication time that overlaps with the compression and backpropagation of any tensors. Given a fusion strategy $\mathcal{X}_y = \{x_0, \ldots, x_{y-1}\}$, the iteration time is

$$f(\mathcal{X}_y) = A + \sum_{b=0}^{N-1} B(T_b) + \sum_{i=0}^{y-1} h(x_i) + \sum_{i=0}^{y-1} g(x_i) - P(\mathcal{X}_y). \quad (1)$$

$A$ and $B(T_i)$ can be profiled offline for a training model and they are constant across iterations (Zhang et al., 2020; Sun et al., 2019). Following the literature (Thakur et al., 2005; NCC, 2021; Fei et al., 2021; Renggli et al., 2019), we model the communication time of $x_i$ as $g(x_i) = \alpha_g + \beta_g x_i$, where $\alpha_g$ is the latency (or startup time) per tensor and $\beta_g$ is the transfer time per byte after encoding. Based on the measurement in Figure 2, we model the compression time of $x_i$ as $h(x_i) = \alpha_h + \beta_h x_i$, where $\alpha_h$ is the fixed overhead to launch and execute kernels in CUDA and $\beta_h$ is the compression time per byte. Cupcake measures $\alpha_g, \beta_g, \alpha_h, \beta_h$ offline on the system configurations, such as the GPU computation capacity, the number of GPUs, and the network bandwidth.

Given a fusion strategy, Cupcake can calculate its iteration time by deriving the timelines of its backpropagation, compression, and communication (Wang et al., 2023), as shown in Figure 3. Unfortunately, it is still challenging to formulate $f(\mathcal{X}_y)$ due to $P(\mathcal{X}_y)$, which is determined by the strategy and the intricate interactions among backpropagation, compression, and communications of all the tensors.

Instead of deriving the expression of $P(\mathcal{X}_y)$, we formulate the tensor fusion problem in a recursive way to minimize the iteration time of a DDT job with a given GC algorithm. Let $F(M, i)$ be the iteration time of the optimal fusion strategy from $T_i$ to $T_{N-1}$, given the fusion strategy for tensors from $T_0$ to $T_{i-1}$ is represented by $M$. We then have the following recurrence relation
We first introduce two pruning techniques based on two cases. The first case is that except tensors in \( x_0 \), all the other tensors are fused as one group for compression and communication, and \( x_1 \)'s communication begins right after \( x_0 \)'s communication, as shown in Figure 4a. The compression time of \( x_1 \) is perfectly overlapped with communication. If the optimistic outcome of a case is already greater than the iteration time of the best fusion strategy found so far, then this case, i.e., the recursive computation for \( F(\{\text{fuse}(0, i - 1)\}, i) \), can be pruned.

**Insight #2:** It is safe to fuse more tensors in a group based on the progress of communication of the previous group. Suppose \( x_0 \) is fused from \( T_0 \) to \( T_{i-1} \). We apply Equation (3) recursively to enumerate cases for \( x_1 \), which fuses tensors from \( T_i \) to \( T_j \). The backpropagation time and the compression time of \( x_1 \) can overlap with \( x_0 \)'s communication, as shown in Figure 4b. The smallest \( j \) can be calculated with

\[
j^* = \arg \max_j \{ B(i, j) + h(i, j) \leq g(0, i - 1) \}.
\]  

Note that the less number of tensors in \( x_1 \) means more tensors in \( x_2 \) and DDT has to communicate more tensors after the completion of backpropagation. Fusing from \( T_i \) to \( T_j \), where \( j < j^* \), is no better than fusing from \( T_i \) to \( T_j \), because it shrinks the overlapping time between communication and computation. Therefore, Cupcake can prune the strategies in which \( x_1 \) fuses tensors from \( T_i \) to \( T_j \) where \( j < j^* \) and only examine \( j \geq j^* \).

**Fusion Algorithm.** Based on the two insights of examining possible fusion strategies, we design Algorithm 1 to find the optimal fusion strategy given a DDT job and a GC algorithm. The function `Main()` checks the \( N \) cases of \( x_0 \) and it invokes `FindOptFusion()` to recursively search for the optimal strategy (lines 1-5). `FindOptFusion()` takes two inputs \( M \) and \( k: M = \{x_0, \ldots, x_{n-1}\} \), which is the fusion strategy for tensors from \( T_0 \) to \( T_{k-1} \), and \( T_k \) is the first tensor to be fused for \( x_n \).

Algorithm 1 uses `global_opt_fuse` to store the best fusion strategy found so far and `local_opt_fuse` to store the local best strategy from \( T_k \) to \( T_{N-1} \). Given \( M \), it first applies
Figure 5. A general case for the two pruning techniques given \( M \), which is the set of fused tensors from \( T_0 \) to \( T_{k-1} \). \( M.\text{comp} \) and \( M.\text{delay} \) can be derived based on the timelines of backpropagation, compression, and communication of tensors in \( M \).

the second insight to fuse the first group beginning from \( T_k \) (Lines 9-16). The example illustrated in Figure 4 only considers the case that \( x_{a-1} \)’s communication begins right after its compression, but it is likely that its communication can be delayed by communication of its previous group. The algorithm replaces \( g(0, i - 1) \) in Expression (5) with \( M.\text{delay} \), which denotes the difference between the completion time of compression and communication of \( x_{a-1} \). The two cases of \( M.\text{delay} \) are illustrated in Figure 5. We also denote \( M.\text{comp} \) as the time duration from the beginning of backpropagation to the completion of \( x_{a-1} \)’s compression, as shown in Figure 5. Given \( M \), both \( M.\text{delay} \) and \( M.\text{comp} \) can be calculated based on the timelines of backpropagation, compression, and communication, regardless of the strategy to fuse the remaining tensors.

Algorithm 1 finds \( j^* \) based on \( M.\text{delay} \) (Lines 9-16) to skip the enumerations of tensors from \( T_k \) to \( T_{j-1} \). It then uses the first insight to calculate the optimistic outcome (Lines 19-21). Similarly, the example shown in Figure 4 only considers the case that \( x_i \)’s communication begins right after its compression. However, it is also likely that its communication can be delayed by \( x_{a-1} \)’s communication, as shown in Communication case 2 in Figure 5. The algorithm considers both cases and calculates the optimistic outcome. Algorithm 1 prunes the search if the optimistic outcome is already greater than the iteration time of the current best strategy. Suppose \( x_a \) is fused from \( T_k \) to \( T_i \), \( \text{FindOptFusion()} \) recursively applies itself to find the local optimal fusion strategy from \( T_{i+1} \) to \( T_{N-1} \) (Lines 25-28). It updates \( \text{local}\_\text{opt}\_\text{fuse} \) and \( \text{global}\_\text{opt}\_\text{fuse} \) if the current strategy is better (Lines 29-36).

In practice, Algorithm 1 can use a heuristic to bootstrap \( \text{global}\_\text{opt}\_\text{fuse} \) with a relatively good fusion strategy. For example, it can partition a DNN model into multiple groups (e.g., two groups) with the same number of tensors.

Time complexity. The complexity of Algorithm 1 is \( O(2^N) \) because it has to enumerate all fusion strategies in the worst case. Fortunately, the two pruning techniques can prune most of them and enable Cupcake to find the optimal one quickly, as we will show in Section 5.3.

**Theorem 1.** Algorithm 1 finds the optimal fusion strategy that minimizes the iteration time of a DDT job given a GC algorithm.

**Proof.** Algorithm 1 recursively invokes function \( \text{FindOptFusion}(M, k) \). Let \( n = N - k \), which is the number of tensors this function considers. We use induction on \( n \) to prove that the function finds the optimal fusion strategy from \( T_k \) to \( T_{N-1} \) given \( M \).

Base case. When \( n = 1 \), the function only needs to examine one tensor and thus only one fusion strategy, which is the optimal one.

Inductive step. Assume that for any \( 1 \leq n \leq p \), \( \text{FindOptFusion}(M, k) \) returns the optimal fusion strategy from \( T_k \) to \( T_{N-1} \) given \( M \). Consider \( n = p + 1 \). Algorithm 1 divides the problem into \( p + 1 \) cases, where case \( i \) (\( 0 \leq i \leq p \)) fuses the first group from \( T_k \) to \( T_{k+i} \).

We first consider case \( i \) where \( 0 \leq i \leq p - 1 \). The function invokes \( \text{FindOptFusion}(M + \text{fuse}(k, k+i), k+i+1) \), in which the number of tensors considered is \( p - i \leq p \). Hence, it outputs the optimal strategy for case \( i \) based on the assumption. We then consider case \( i = p \) and the first group is fused from \( T_k \) to \( T_{N-1} \). This is the only fusion strategy and thus the optimal one. Because these cases are exclusive and cover the entire search space, Algorithm 1 finds the optimal fusion strategy for \( n = p + 1 \) by searching for the optimal one from these cases.

Algorithm 1 applies two pruning techniques to quickly find the optimal fusion strategy. The first one prunes the cases whose lower bounds are no better than the optimal found so far and it has no impact on the optimality. The second one prunes the cases whose first groups are fused from \( T_k \) to \( T_j \),
Algorithm 1: Optimal Fusion Strategy

Input: $N$ is the number of tensors in a DNN model.

$global\_opt\_fuse = \{\}$. $global\_opt\_time = \infty$

Output: The optimal fusion strategy $global\_opt\_fuse$.

Function Main():
  for $k \leftarrow 1$ to $N$ do
    FindOptFusion($\{\text{fuse}(0, k-1)\}$, $k$)
  end
  return $global\_opt\_fuse$

Function FindOptFusion($M(k)$):
  $local\_opt\_fuse \leftarrow \{f\text{use}(k, N - 1)\}$$
  // $f()$ is defined in Equation [3]
  $local\_opt\_time \leftarrow f(M + \text{fuse}(k, N - 1))$$
  j^* \leftarrow k$
  for $i \leftarrow k$ to $N - 1$ do
    if $B(k, i) + h(k, i) \leq M\text{.delay}$
      $j^* \leftarrow i$
    else
      break
  end
  $M\_comp = B(0, k - 1) + \sum_{x \in M} h(x)$
  for $i \leftarrow j^*$ to $N - 1$ do
    $base \leftarrow B(k, N - 1) + h(k, i) + h(i + 1, N - 1)$
    $cases \leftarrow \max\{B(k, i) + h(k, i), M\text{.delay}\} + g(k, i) + g(i + 1, N - 1)$
    $optim\_outcome \leftarrow M\_comp + \max\{base, cases\}$
    if $optim\_outcome > local\_opt\_time$
      continue
    end
    $first\_fuse \leftarrow \text{fuse}(k, i)$
    $rest\_fuse \leftarrow$
      FindOptFusion($M + first\_fuse$, $i + 1$)
    $cur\_fuse \leftarrow first\_fuse + rest\_fuse$
    $cur\_fuse\_time = f(M + cur\_fuse)$
    if $cur\_fuse\_time < local\_opt\_fuse\_time$
      $local\_opt\_fuse \leftarrow cur\_fuse$
      $local\_opt\_time \leftarrow cur\_fuse\_time$
    end
    if $cur\_fuse\_time < global\_opt\_time$
      $global\_opt\_fuse \leftarrow M + cur\_fuse$
      $global\_opt\_time \leftarrow cur\_fuse\_time$
    end
  end
  return $local\_opt\_fuse$

where $j < j^*$. Because they cannot advance communication to an earlier point than fusing from $T_k$ to $T_j$, pruning these cases does not affect the optimality.

5 EXPERIMENTS

In this section, we will first show the performance improvement of Cupcake for GC algorithms with sparsification and quantization. We then evaluate Time-to-Accuracy to demonstrate that Cupcake can preserve the accuracy of these applied GC algorithms. At last, we show that Cupcake can find the optimal fusion strategy quickly.

Setup. Two testbed setups are used for the evaluations. The first setup is the same as that described in Section 3. The second setup has 8 GPU machines connected to a 25Gbps network. Each machine has 8 NVIDIA Tesla V100 GPUs (32 GB GPU memory) connected by NVLink and 48-core/96-thread processors (Intel Xeon 8260 at 2.40GHz). The server has an Ubuntu 18.04.4 LTS system and the software environment includes PyTorch-1.8.1, Horovod-0.22.1, CUDA-11.1, and NCCL-2.9.9.

Workloads. We validate the performance of Cupcake on two types of machine learning tasks: computer vision and natural language processing (NLP). The models include ResNet50 over CIFAR10 (Krizhevsky et al., 2009) and ResNet101 (He et al., 2016) over ImageNet-1K (Deng et al., 2009), BERT-base (Devlin et al., 2018) over SQuAD (Rajpurkar et al., 2018). These models are widely used as standard benchmarks to evaluate the scalability of DDT. The batch sizes for ResNet50 and ResNet101 are 32 and for BERT-base are 1024 samples.

Compression algorithms. We use four representative GC algorithms: Rand-k (Stich et al., 2018) and DGC (Lin et al., 2017) for sparsification with 99% sparsity, and EF-SignSGD (Karimireddy et al., 2019) and OneBit (Seide et al., 2014) for quantization. Error-feedback (Karimireddy et al., 2019; Lin et al., 2017) is applied to GC algorithms to preserve the model accuracy.

Baselines. We use Horovod (Sergeev & Del Balso, 2018) as the training baseline without GC (FP32). We use GRACE (Xu et al., 2020) and HiPress (Bai et al., 2021) as the two layer-wise baselines for applying GC to DDT. GRACE applies GC to all tensors in a model and HiPress only compresses tensors greater than a threshold, which is determined by the tensor size, network bandwidth, and compression overhead.

Metrics. Suppose the training speed with $n$ GPUs is $T_n$. The scaling factor (Zhang et al., 2020) is defined as $\frac{T_1}{T_n}$. We use the scaling factor, Top-1 accuracy, and F1 score as evaluation metrics. The results for scaling factors are reported with an average of 100 iterations. We also report the standard deviation using the error bar because the training speed varies at times.

Allgather for communications. Allreduce is used for communications in FP32 (Sergeev & Del Balso, 2018; Paszke et al., 2019). Existing frameworks’ implementation of Allreduce requires tensors to be aligned and support element-wise aggregations. However, compressed tensors typically do not satisfy these requirements. For example, compressed tensors using Rand-k have different indices for selected elements, while those using OneBit cannot support addition. In contrast, the implementation of Allgather (Thakur et al., 2005) has no such restrictions. It gathers tensors from all
GPUs and allows for customized aggregation operations for compressed tensors. Therefore, we chose to use All-gather in our implementation to communicate compressed tensors (Xu et al., 2020; Wang et al., 2023).

5.1 Training Speed Improvement

Figure 6 shows the scaling factors of the three DNN models running on a server with 8 GPUs connected by PCIe 3.0 × 16. The four compression algorithms are applied with Cupcake and the two layer-wise baselines, respectively. We can see from Figure 6 that applying GC in a layer-wise fashion can even harm the performance of DDT due to the costly compression overhead. The scaling factors of both ResNet50 and ResNet101 with GRACE compression are lower than those without any compression in most cases. HiPress outperforms GRACE because it avoids encoding small tensors and incurs less compression overhead. However, its improvement in the training throughput is just modest compared to training without GC. Applying Rand-k, DGC, and EFSignSGD to the training of BERT with HiPress can improve the training speed, but it still harms the training performance when the GC algorithm is OneBit.

In contrast, Cupcake significantly improves the training speed of DDT with GC algorithms for the three DNN models compared to FP32. For the training of ResNet50, it outperforms FP32 by up to 72% (apply Rand-k). It also outperforms GRACE and HiPress by up to 130% and 64% (apply OneBit), respectively. For ResNet101, Cupcake outperforms FP32, GRACE, and HiPress by up to 39%, 70%, and 37%, respectively. For BERT-base, it outperforms FP32, GRACE, and HiPress by up to 65%, 106%, and 61%, respectively.

Figure 7 shows the scaling factors of the three DNN models running on 64 GPUs in 8 servers connected by a 25Gbps network. Because intra-machine communications are supported by NVLink, which can provide every GPU in total 1.2Tbps GPU-GPU bandwidth (Jiang et al., 2020), the performance bottleneck is inter-machine communications. Therefore, tensors are not compressed for intra-machine communications and GC is applied for inter-machine communications only. Figure 7 shows that the speedups of Cupcake over FP32 are up to 93%, 46%, and 103% for the training of ResNet50, ResNet101, and BERT-base, respectively. It also outperforms HiPress by up to 79%, 37%, and 58% for the training of the three models, respectively.

5.2 Time-to-Accuracy Improvement

Because HiPress is always better than GRACE in terms of the training throughput, we compare Cupcake to HiPress in this section. We train ResNet50 over CIFAR10 until convergence on a server with 8 GPUs connected by PCIe 3.0 × 16. The applied GC algorithm is DGC. As shown in Figures 8a, Cupcake can achieve around 1.68× speedup over no compression (i.e. FP32), and 1.30× speedup over HiPress. The achieved Top-1 accuracy with Cupcake is
Cupcake: A Compression Optimizer for Scalable Communication-Efficient Distributed Training

Figure 8. Cupcake achieves almost the same model accuracy as no compression. DGC and EFSignSGD are applied to the training of ResNet50 over CIFAR10 and ResNet101 over Imagenet-1K, respectively. Both Rand-k and DGC are applied to the training of BERT-base over SQuAD.

Table 1. Running time of Algorithm 1.

<table>
<thead>
<tr>
<th></th>
<th>ResNet50</th>
<th>ResNet101</th>
<th>BERT-base</th>
</tr>
</thead>
<tbody>
<tr>
<td># of tensors</td>
<td>161</td>
<td>314</td>
<td>207</td>
</tr>
<tr>
<td>Algorithm[1]</td>
<td>2.8 s</td>
<td>6.6 s</td>
<td>4.2 s</td>
</tr>
<tr>
<td>Only Pruning 1</td>
<td>15 s</td>
<td>68 s</td>
<td>32 s</td>
</tr>
<tr>
<td>Only Pruning 2</td>
<td>2.2 h</td>
<td>9.4 h</td>
<td>&gt; 24 h</td>
</tr>
<tr>
<td>No Pruning</td>
<td>&gt; 24 h</td>
<td>&gt; 24 h</td>
<td>&gt; 24 h</td>
</tr>
</tbody>
</table>

93.2% (with HiPress is 93.1%), which is very close to the no-compression accuracy of 93.6%. We also train ResNet101 for 120 epochs on ImageNet-1K from scratch and apply EFSignSGD to the model training. Figure 8b shows that Cupcake outperforms no compression and HiPress by 1.32× and 1.25×, respectively. The achieved Top-1 accuracy with Cupcake, HiPress, and no compression is 76.7%, 76.6%, and 77.1%, respectively. In addition, we conduct a test following the methodology in (Fei et al., 2021) to fine-tune BERT-base for the question-answering task on SQuAD (Rajpurkar et al., 2018) for two epochs and repeat the experiments ten times. Figure 8c shows that Cupcake with DGC can achieve around 1.65× speedup over no compression and it has almost the same F1 score as no compression.

5.3 Effectiveness of Cupcake

Computation time. We first measure the computation time of Algorithm 1 with the two pruning techniques. Note that the number of tensors in a DNN model, their sizes, and the computation time of backpropagation are measured in advance. The cost model of the communication time is determined by the network bandwidth. We also profile the encoding and decoding overheads of a GC algorithm, as shown in Figure 2, to model the compression time.

Table 1 shows that it only takes several seconds for Algorithm 1 to find the optimal fusion strategy for the three DNN models when training them on a server with 8 GPUs connected by PCIe 3.0 × 16. For example, the computation time is only a few seconds even for ResNet101 which has 314 tensors. However, the search cannot finish after running for 24 hours without the two pruning techniques, i.e., searching for the optimal strategy with brute force.

Compared to strawman solutions. We also compare Cupcake with the following two strawman solutions for tensor fusion.

- Bucket Fusion (Li et al., 2020; Sergeev & Del Balso, 2018). It stores tensors in a buffer and fuses tensors in the buffer when their total size exceeds a threshold. We set different thresholds from 2 MB to 64 MB and use the best performance as its performance.
- Evenly Split. It evenly splits consecutive tensors into multiple groups for fusion and each group has the same number of tensors. We set the number of groups from 2 to 32 and use the best performance as its performance.

We apply DGC with three fusion strategies, Cupcake, Bucket Fusion, and Evenly Split, to three DNN models when training them on a server with 8 GPUs. Figure 9 displays their scaling factors. Both Bucket Fusion and Evenly Split outperform the layer-wise baselines thanks to the reduced compression overheads. Cupcake outperforms them by up to 1.12× and 1.18×, respectively. Cupcake searches for the optimal fusion strategy from the whole search space. We observe that the number of tensors and the size of the fused tensor vary a lot across groups in the optimal strategy for the three evaluated DNN models. However, the two strawman solutions limit the search space and constrain that each group has to have the same number of tensors or the same fused tensor size, leading to suboptimal fusion strategies.

Figure 9. The scaling factors of three DNN models running on a server with 8 GPUs. The GC algorithm is DGC.
6 RELATED WORK

Many GC algorithms have been proposed to reduce the amount of communicated traffic volume for gradient synchronization, as discussed in Section 2.3. However, they are designed from an algorithmic perspective and have no suggestion for how to efficiently apply them to DDT. In contrast, Cupcake is not a GC algorithm, but a compression optimizer to maximize the benefits of GC algorithms from a system perspective.

Recent work quantitatively evaluated the impacts of GC algorithms in a layer-wise fashion. They observe that GC can incur non-negligible compression overheads, but they have no solution to address the challenges of applying GC to DDT. HiPress proposes a selective compression mechanism to determine whether to compress a tensor, but it still applies GC algorithms to a DDT job in a layer-wise fashion and incurs costly compression overheads. Cupcake uses a fusion fashion to minimize the incurred compression overheads.

Distributed deep learning frameworks batch multiple tensors for one communication operation to improve communication efficiency. However, this mechanism takes place after compression and is orthogonal to GC algorithms. PipeSwitch fuses tensors to pipeline model transmission over the PCIe for fast context switching of deep learning applications. In contrast, Cupcake fuses tensors to improve compression efficiency.

7 CONCLUSION

Cupcake is a general compression optimizer to enable GC algorithms to fully unleash their benefits to accelerate the training throughput of DDT jobs. Instead of compressing tensors in a layer-wise fashion, Cupcake applies GC algorithms in a fusion fashion and can find the provably optimal fusion strategy to maximize the training throughput of compression-enabled DDT. Cupcake can significantly improve the performance of DDT over full synchronization and solutions with layer-wise compression.

ACKNOWLEDGMENT

We would like to thank our shepherd Ang Li and the anonymous reviewers for providing valuable feedback. Zhuang Wang, Xinyu Crystal Wu, and T. S. Eugene Ng are partially supported by the NSF under CNS-2214272 and CNS-1815525.

REFERENCES


Wang, Z., Xu, Z., Wu, X., Shrivastava, A., and Ng, T. E.  


