Training Deep Learning Recommendation Model with Quantized Collective Communications

Jie (Amy) Yang
Jongsoo Park
Srinivas Sridharan
Ping Tak Peter Tang
amyyang,jongsoo,ssrinivas,ptpt@fb.com
Facebook Inc.
Menlo Park, California

ABSTRACT
Deep Learning Recommendation Model (DLRM) captures our representative model architectures developed for click-through-rate (CTR) prediction based on high-dimensional sparse categorical data. Collective communications can account for a significant fraction of time in synchronous training of DLRM at scale. In this work, we explore using fine-grain integer quantization to reduce the communication volume of alltoall and allreduce collectives. We emulate quantized alltoall and allreduce, the latter using ring or recursive-doubling and each with optional carried-forward error compensation. We benchmark accuracy loss of quantized alltoall and allreduce with a representative DLRM model and Kaggle 7D dataset. We show that alltoall forward and backward passes, and dense allreduce can be quantized to 4 bits without accuracy loss compared to full-precision training.

ACM Reference Format:

1 INTRODUCTION
Deep Learning Recommendation Model (DLRM) [10] captures the structure of recommendation models that deliver site content tailored to users’ interests. DLRM architecture is motivated by the prevalence of high-dimensional categorical features. The categorical features are commonly represented with one- or multi-hot vectors with dimension equals to the number of items in the category, exhibiting a high sparsity. State-of-the-art DLRMs often adopt the approach of embedding tables, which map high-dimensional sparse vectors from raw categorical features onto low-dimensional dense vector representations [6, 8, 15]. Such embedding tables often have dimensions of tens of millions of rows by hundreds of columns, with sizes up to the order of GBs per table [9].

Figure 1 shows a representative DLRM architecture. Dense features are processed with a multi-layer perceptron (MLP), then joined with sparse embedding lookups in the feature interaction module (the green box). The sparse-dense interactions are then fed to the top MLP which in turn passes its output to a sigmoid function to generate a click-through-rate (CTR) prediction [10].

Figure 1: Architecture of DLRM [10]

As we continually grow the complexity of models to improve prediction quality, we investigate synchronous training leveraging collective communications so that training speed can keep up. Our synchronous training uses a combined data- and model-parallel approach for DLRM. We partition the memory intensive sparse embedding tables across nodes with model parallelism, and replicate the compute intensive MLP layers across all nodes with data parallelism [9]. Each node has a partial copy of the sparse embedding tables and a full copy of MLP layers.

Such model partitioning leads to the use of collective communication primitives to synchronize the computation nodes. Partitioning of sparse embedding tables across nodes requires nodes to aggregate sparse embedding lookups in forward passes, and their corresponding gradients in backward passes. We thus use alltoall to synchronize sparse lookups and sparse gradients. Replication of MLP with data parallelism requires nodes to aggregate MLP gradients across different parts of the mini-batch to compute the average gradients. We thus use allreduce to synchronize dense gradients [9]. Figure 2 illustrates alltoall and allreduce in DLRM training.
Synchronization of aforementioned sparse forward/backward passes and dense backward passes across all nodes during each mini-batch iteration places a significant overhead on training latency. Empirically, we observed allreduce message sizes around 10 MB and alltoall message sizes around 100 KB. Large message sizes in alltoall and allreduce communications stress the network fabric even with the presence of high-bandwidth interconnect [9].

To address the overhead of collective communication on synchronous training of DLRM, we explore integer quantization to reduce the message sizes of alltoall and allreduce. Specifically, we make the following contributions: (1) We developed a light-weight single-node numerical benchmark that enables exploratory study of mixed-precision integer quantization of alltoall and allreduce on DLRM training. (2) We integrated error compensation in quantized alltoall and allreduce algorithms, and showed the efficacy of error compensation in recovering accuracy loss due to quantized communication. (3) We showed different allreduce algorithms have different degrees of impact on accuracy loss of integer gradient quantization. (4) We demonstrated DLRM’s sensitivity to communication quantization during training, and explored the most effective quantization precisions for alltoall and allreduce, taking into account different allreduce algorithms.

The rest of the paper is organized as follows: Section 2 discusses related works on the subject of reducing communication overhead with quantization and sparsification. Section 3 explains our methodology in developing single-node numerically faithful emulations of collective communications in multi-node training. Section 4 shows our experiment results on a representative DLRM model and dataset, evaluated with different precisions and emulation settings. Section 5 summarizes our observations and directions for future work.

2 RELATED WORKS

Many recent works tackle communication latency in deep learning training by quantization and sparsification.

QSGD [1] quantized gradients to 8 bits and 4 bits in ImageNet152 training without accuracy loss compared to full precision training. TernGrad [16] quantized gradients to 3 bits without accuracy loss for AlexNet, with the addition of gradient clipping and per-layer ternarizing. An earlier work from Seide, et al. [11] quantized gradients to 1 bit without losing accuracy for a speech DNN model, with an additional technique of carrying quantization error forward and compensate error across mini-batches. DoReFa-Net [17] quantized all of weights, gradients, and activations in training AlexNet, and achieved accuracy comparable with full precision training with weights quantized to 1 bit, gradients to 6 bits, and activations to 2 bits. These works achieved varying degrees of speedup against full precision training, with the measured speedup dependent on cluster setting and model architecture.

Other works reduce communication of gradients via sparsification: gradients are communicated in full precision, except that many components are dropped [4, 7, 12, 13]. All of them incorporate a theoretically proven effective error compensation.

Similar to previous works, we aim to reduce communication overhead during training by quantizing activations and gradients, with the addition of carried-forward error compensation commonly applied with gradient sparsification work and [11].

Our work differs from prior efforts in that: (1) Model architecture: prior work focuses on application to convolutional neural network (CNN) and speech DNN models, while we focus on DLRM, which has drastically different architectures, computation/memory characteristics, and more stringent accuracy requirements. (2) We consider the idiosyncrasies of different collective communication algorithms, while prior works often assume a parameter server architecture or a “star” topology. In collective communications, quantized gradients are partially accumulated through multiple hops, and new quantization error is incurred with each additional hop. This difference is consequential as the accuracy impact of a quantization methodology may be heavily affected by different collective communication patterns, especially in the case of large cluster sizes, as we will demonstrate in Section 4.

3 METHOD

As mentioned previously, synchronous training for DLRM uses alltoall for sparse forward and backward passes, and allreduce for dense backward passes. For fast exploration of the combined effect of integer quantization with different collective communication algorithms, we implemented a lightweight single-node benchmark that emulates the numerical behavior of quantized collective communications in multi-node training. The following sections discuss our emulation implementation in details: Section 3.1 explains our implementation of integer quantization; Section 3.2 explains how we emulate the numerics of sending and receiving quantized values over network, and general error-compensated quantization; Section 3.3 and 3.4 describe our (error-compensated) emulation of quantized alltoall and allreduce algorithms.

3.1 Integer Quantization

IEEE 32-bit single precision (FP32) is the default datatype for training most DL models but alternative lower-precision datatype has been used successfully as well (c.f. [3, 5]). The integer quantization scheme we adopt here uses q-bit unsigned integers, \( q = 8, 4, 2 \) to quantize (i.e. approximate) each value \( x \) in a group of FP32 values contained in a range \([x_{\min}, x_{\max}]\) by values on the uniform grid

\[
x \approx x_{\min} + j(x_{\max} - x_{\min})/(2^q - 1) = s(j - z),
\]  

Figure 2: alltoall and allreduce in DLRM training [9]
for \( j = 0, 1, \ldots, 2^q - 1; s = (x_{\text{max}} - x_{\text{min}})/(2^q - 1) \) and \( z = -x_{\text{min}}/s \). The quantization function \( Q(x) \) takes a row of values \( x \), obtains \( x_{\text{min}}, x_{\text{max}}, s, \) and \( z \) in Equation 1 and returns \( Q(x) = \text{roundint}((x/s) + z) \), where \( \text{roundint} \) rounds values to nearest even.

Collective communications with quantized values work as follows. Before a message \( T \) (tensor) is to be transferred over the network via send, it is quantized \( Y := Q(T) \) row-wise to reduce the message size. The parameters \( s \) and \( z \) are also sent. Upon recv at the destination node, \( Y \) is dequantized back into FP32 datatype by \( D(Y) \) using the formula \( s(y - z) \) row-wise. Subsequent computations on the dequantized values are in FP32.

### 3.2 Communication and Numerics Emulation

We use a single node to emulate the communication of a \( N \)-node training system in a way to reflect faithfully the numerical effects of quantization. Each parameter or gradient tensor \( T \) to be communicated is partitioned \( N \)-ways into \( T_0, T_1, \ldots, T_{N-1} \) so as to emulate data local to each of the \( N \) nodes. Depending on the specific collective communication algorithm in question, each partition \( T_p \) is further sub-partitioned \( P \)-ways: \( T_p = [T_{(p,0)}, T_{(p,1)}, \ldots, T_{(p,P-1)}] \). Sub-partitions such as \( T_{(p,k)} \) are the smallest unit being sent and received in a collective communication primitive. We emulate the exchange of parameters/gradients over network in multi-node cluster by manipulating sub-partitions in the single-node tensor. Such emulation is flexible, easy to implement, and places light overhead in single-node training to allow fast experimental iterations.

In a nutshell, collective communication algorithms synchronize a given pool of processes via efficient orchestrations of local computations and peer-to-peer exchange of information. We now describe our fundamental primitive functions that allow us to emulate the collective communication in a numerically faithful way.

- **send and recv**: Collective communications on quantized values require these operations to send and receive quantized data over a network via point-to-point connections. Our single-node emulation has no need for them as all data reside in the same local tensor. Nevertheless we will state the key emulation algorithms in the sequel with these two primitives for clarity even though they are actually noops in our emulation algorithms.
- **In our emulation, we use the sequence of operations quantize-send-recv-dequantize to replace send-recv to achieve quantized communication. As send and recv are noops in our emulation, we can emulate the numerics of the sequence with quantize-dequantize. It therefore suffices to have the fused functionality \( DQ(T) = D(Q(T)) \) in our emulation algorithms, with the \( DQ \) output stored in FP32 format but the values have gone through quantization and dequantization. As will be shown later, quantization using too few bits can result in unacceptable precision loss of the model. We incorporate the error compensation idea in [13] to counter this problem when necessary. The main idea is that in order to mitigate the loss of precision after quantization, we calculate the quantization error and store it locally, and compensate the error in the next iteration. The \( DQ \) function in Algorithm 1 encapsulates this key operation in our emulation for error compensated collective communications. It reflects quantizing the reduction of two tensors, incorporating an existing compensatory error, and recording this newly introduced quantization error for use in the next mini-batch iteration.

We now describe the emulation for each variant of collective communication we need.

### 3.3 Quantized All-To-All

In an alltoall collective communication algorithm, the same values are relayed over the network multiple times until all nodes have the same copy of all values. A quantized alltoall is one that quantizes the values before sending and dequantizes values upon receipt. Numerically, it tantamounts to \( P_{\text{final}} := D(D(Q(x)) \cdots) \), which is equivalent to \( P_{\text{final}} := DQ(x) \). Consequently, we emulate alltoall by applying the fused quantized function \( DQ \) once to the full-precision tensor \( x \).

#### 3.4.1 Quantized Ring All-Reduce

An allreduce primitive can be implemented with different algorithms to best fit a particular interconnect network [14]. These algorithms differ in sub-partition splitting and node-to-node exchange patterns. From a numerical point of view, for an \( N \)-node cluster, each index of an allreduce tensor has a value at each node. During allreduce, for each index, \( N \) numbers are summed in different order depending on the specific allreduce algorithm, thus can lead to different accuracy characteristics. We implemented numerical emulation of these allreduce algorithms: ring and recursive-doubling. For convenience and without loss of generality, we emulate communication of \( N \) nodes where \( N \) is a power of 2.

#### 3.3.1 Quantized Ring All-Reduce

The \( N \) processes communicate as if they’re aligned on a ring. The allreduce algorithm partitions data of each node \( N \)-way. In each iteration every node receives a partition from its left neighbor, accumulates with a local partition and sends the result to its right. After \( N-1 \) steps, each node possesses the complete sum of a partition; these partitions are passed around once more in a round-robin fashion until all nodes have the complete sum of all partitions.

Algorithm 2 shows the details of the error compensated ring allreduce. A node \( p \) has local data tensor \( T_p \) and error tensor \( E_p \) partitioned \( N \)-ways: \( X_p = (X_{(p,0)}, X_{(p,1)}, \ldots, X_{(p,N-1)}) \), where \( X = T \) or \( E \). For the version without error compensation, simply replace \( DQ(T, Y, E) \) with \( DQ(T + Y) \) and consider all error terms \( E \) to be 0 or non-existent.

#### 3.3.2 Quantized Recursive-Doubling All-Reduce

Recursive doubling builds from the base case of reducing two nodes, labeled as Node 0 and Node 1. Each node splits its local data into two partitions. Node 0 sends Node 1 Partition 1, and Node 1 to Node...
Algorithm 4: Error Compensated RD allreduce

\begin{algorithmic}
\Function{RD(nodestrain)}{nodes:}\end{algorithmic}

\begin{algorithmic}
\For {$i = 0, 1, 2, \ldots, N - 2$ do}
\For {$p = 0, 1, 2, \ldots, N - 1$ do}
\State $p_t, p_r := p - 1, p + 1$ mod $N$
\State $k := p - i$ mod $N$
\State $m := k + 1$ mod $N$
\State send $DQ(T_{(p,m)})$ to Node $p_r$
\State recv $DQ(T_{(p,k)})$ from Node $p_t$
\State $(T_{(p,k)}, E_{(p,k)}) := DQE(T_{(p,k)}, DQ(T_{(p,k)}), E_{(p,k)})$
\EndFor
\EndFor
\end{algorithmic}

Now $T_{(p,p+2)}$ of each node $p$ has the reduced sum

Share $DQ$ (reduced sum) round robin

Algorithm 3: Error Compensated RD Base

\begin{algorithmic}
\Function{RD_BASE}(Node $p$, Node $q$):\end{algorithmic}

\begin{algorithmic}
\State Node $p$:
\State send $DQ(T_{(p,1)})$ to Node $q$
\State recv $DQ(T_{(p,0)})$ from Node $q$
\State $(T_{(p,0)}, E_{(p,0)}) := DQE(T_{(p,0)}, DQ(T_{(p,0)}), E_{(p,0)})$

\State Node $q$:
\State send $DQ(T_{(q,0)})$ to Node $p$
\State recv $DQ(T_{(p,1)})$ from Node $p$
\State $(T_{(q,1)}, E_{(q,1)}) := DQE(T_{(q,1)}, DQ(T_{(p,1)}), E_{(q,1)})$

\State Gather at Node $q$:
\State recv $DQ(T_{(p,0)})$ from Node $p$
\State return Node $q$
\EndFunction

Now Node $q$ has both reduced partitions

4 EXPERIMENT RESULTS

We benchmark using the Kaggle 7D dataset [2] with the default DLRM model that consists of 26 sparse features with embedding dimensions 16, 13 dense features, a 4-layer bottom MLP and a 4-layer top MLP. Sparse-dense feature interactions are implemented with pairwise dot product, with concatenation of raw dense features. MLP weight dimensions are listed in Table 1. Furthermore, in a N-node cluster, we apply allreduce emulation only to the MLP layers whose first weight dimensions are divisible by $N$. For example, in the 32-node case, we exclude the MLP layers with dims (16, 64) and (1, 256) from the allreduce emulation.

To model popular training systems where each node has 8 GPUs fully-connected with NVSwitch, our emulation for recursive doubling allreduce assumes a hierarchical system where each group of 8 processes are fully connected. Thus our recursive doubling allreduce first starts with an “intra-group” reduction within these groups of 8, as demonstrated by Algorithm 5. Assume each node $n$ has data tensor $T_n$, error tensor $E_n$, each partitioned 8-way into $(T_{(n,0)}, \ldots, T_{(n,7)}), (E_{(n,0)}, \ldots, E_{(n,7)})$, respectively. Each node $n$ reduces the $n$-th partition across the group, and then performs recursive doubling reduction with the $n$-th nodes in other groups by applying Algorithm 4.

Algorithm 5: Error Compensated RD Intra-Group Sum

\begin{algorithmic}
\For {each node $p$ do}
\For {$i = 0, 1, \ldots, 7$ do}
\State send $DQ(T_{(p,i)})$ to all Node $i \neq p$
\EndFor
\EndFor
\For {each node $q \neq p$ do}
\State recv $DQ((q,p))$ from Node $q$
\State $T_{(p,p)} := T_{(p,p)} + DQ((q,p))$
\EndFor
\EndFunction

Each Node $p$ now has reduced $p$-th partition

Table 1: Benchmark Model MLP Dimensions

<table>
<thead>
<tr>
<th>Top MLP</th>
<th>367-512-256-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom MLP</td>
<td>13-512-256-64-16</td>
</tr>
</tbody>
</table>

We implemented following single-node numerical emulations in Pytorch DLRM:
We experimented with different combinations of sparse embeddings. All DLRM training experiments are ran on a single node deterministically and sequentially, with a batch size of 1024 and learning rate of 1.0. Quantization is applied from the first iteration. Experiments are configured with varying integer quantization widths of uint8, 4 and 2 bits and cluster sizes of 32, 64 and 128. Each configuration is ran with 4 different random seeds. We report the percentage change in test accuracy against full-precision training, averaged across the random seeds:

\[ \delta_q = \frac{\text{Avg}_{\text{seed}}((\text{Acc}_q - \text{Acc}_{\text{base}})/\text{Acc}_{\text{base}} \cdot 100)}{2} \]

We set \( \delta_q > -0.02 \), that is an accuracy drop less than 0.02%, as an acceptable accuracy threshold based on empirical experience.

### 4.1 Quantized All-to-All

We experimented with different combinations of alltoall forward and backward precision while allreduce is done with full FP32 precision. Table 2 reports the averaged accuracy change \( \delta_q \) as defined in Equation 2. We highlighted the combination with the lowest bit requirements while maintaining our \( \delta_q > -0.02 \) threshold. Our hypothesis is that sparse forward pass needs more bit width than backward pass due to a wider range of value distribution in raw embedding weights than their gradients and that latter also decrease in magnitude as training progresses.

<table>
<thead>
<tr>
<th>Forward</th>
<th>Backward</th>
<th>( \delta_q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>uint8</td>
<td>0.00096</td>
</tr>
<tr>
<td>uint8</td>
<td>uint4</td>
<td>-0.00380</td>
</tr>
<tr>
<td>uint4</td>
<td>uint8</td>
<td>0.00571</td>
</tr>
<tr>
<td>uint4</td>
<td>uint4</td>
<td>-0.00222</td>
</tr>
<tr>
<td>uint4</td>
<td>uint2</td>
<td>0.01651</td>
</tr>
<tr>
<td>uint2</td>
<td>uint4</td>
<td>-0.05674</td>
</tr>
<tr>
<td>uint2</td>
<td>uint2</td>
<td>-0.06339</td>
</tr>
</tbody>
</table>

### 4.2 Quantized All-Reduce

Next, we configure alltoall to use full FP32 precision while setting allreduce to use 8, 4 or 2 bits in quantization for ring as well as RD algorithms. In addition, whenever \( \delta_q \leq -0.02 \), we repeat the experiments with error compensation, labeled as EC in Table 3 which reports the average accuracy change \( \delta_q \).

Note that one can quantize RD allreduce down to 4 bits without EC and maintain \( \delta_q > -0.02 \) on cluster sizes up to 128. Ring allreduce can be quantized to 8 bits; while quantization to 4 bits shows unacceptable accuracy drops, accuracy is fully recovered with error compensation. RD algorithm yields better accuracy than ring because (1) intra-group one-shot summation in each group of 8 nodes (2) recursive summation incurs \( O(\log_2 N) \) quantizations while ring incurs \( O(N) \). At the 2 bit level, all configurations result in unacceptable accuracy loss. We observe an apparent anomaly with EC on the ring reduction in 2 bits where error compensation worsen the situation and will investigate further.

### 4.3 Quantized All-Reduce All-to-All Combined

Our previous experiments benchmarked accuracy change of quantizing alltoall and allreduce independently. Table 4 reports the accuracy changes \( \delta_q \) when both collective communications are quantized. We use \((x, y)\) to denote the alltoall forward and backward quantization bit widths \( x \) and \( y \), respectively. EC denotes error compensation as before. Thus far, the most performant configurations that pass our accuracy requirement is 4-bit ring allreduce with EC, with alltoall forward/backward passes in 4 bits. Interestingly, we observe that 4-bit RD with error compensation performs worse than 4-bit ring with error compensation, while previous results in Table 3 shows RD performs better than ring for both 2-bit and 4-bit in the standalone allreduce experiments.

<table>
<thead>
<tr>
<th>Forward &amp; Backward</th>
<th>32 nodes</th>
<th>64 nodes</th>
<th>128 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8 ring; (f4, b4)</td>
<td>0.00031</td>
<td>-0.01395</td>
<td>-0.02631</td>
</tr>
<tr>
<td>uint4 ring EC; (f4, b4)</td>
<td>-0.01109</td>
<td>-0.00444</td>
<td>-0.00190</td>
</tr>
<tr>
<td>uint4 ring; (f4, b4)</td>
<td>0.01902</td>
<td>-1.58532</td>
<td>-0.01553</td>
</tr>
<tr>
<td>uint4 RD; (f4, b4)</td>
<td>-0.01458</td>
<td>-0.01226</td>
<td>-0.03678</td>
</tr>
</tbody>
</table>

### 5 Conclusion and Future Work

We constructed a single-node numerically faithful emulation of quantized alltoall and allreduce on top of which we can train DLRM. We investigated thoroughly the accuracy implications under different quantization bitwidths and specific communication algorithms using a default DLRM model with Kaggle 7D dataset.

When alltoall alone is quantized, forward and backward passes can be quantized to 4 and 2 bits, respectively, while maintaining accuracy on par with full-precision models – thus achieving up to
8x and 16x bandwidth savings for sparse forward pass and backward pass, respectively. When allreduce alone is quantized, that is, quantizing the dense gradients during backward passes, we are able to quantize ring and RD allreduce to 4 bits while maintaining on par accuracy compared to full-precision training. We thus can expect up to 8x bandwidth saving with quantized allreduce. When allreduce and alltoall are both quantized, allreduce-alltoall combination shows neutral accuracy with allreduce quantized to 4 bits, and alltoall forward/backward pass quantized to 4 bits, which would result in overall 8x bandwidth reduction for allreduce and alltoall combined. We also demonstrated that error compensation is generally a powerful technique as it significantly improves accuracy when bitwidth reaches down to 4 or 2 bits.

As part of future work, we will continue to analyse the numerical subtleties of quantization applied to collective communication training. For example, in one situation we observed that error compensation worsened the accuracy loss rather than recovering it. We would like to implement quantized versions of collective communication libraries, and measure the accuracy impact of quantized communication as well as error compensation on training clusters leveraging collective communications.

REFERENCES


