THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression

Minghao Li, *Harvard University;* Ran Ben Basat, *University College London;* Shay Vargaftik, *VMware Research;* ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu, *Harvard University*

https://www.usenix.org/conference/nsdi24/presentation/li-minghao

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression

Minghao Li[††], Ran Ben Basat[†], Shay Vargaftik[§], ChonLam Lao[††],
Kevin Xu[††], Michael Mitzenmacher[††], Minlan Yu[††]

Harvard University [††], University College London [†], VMware Research [§]

Abstract

Deep neural networks (DNNs) are the de facto standard for essential use cases, such as image classification, computer vision, and natural language processing. As DNNs and datasets get larger, they require distributed training on increasingly larger clusters. A main bottleneck is the resulting communication overhead where workers exchange model updates (i.e., gradients) on a per-round basis. To address this bottleneck and accelerate training, a widely-deployed approach is compression. However, previous deployments often apply bi-directional compression schemes by simply using a uni-directional gradient compression scheme in each direction. This results in significant computational overheads at the parameter server and increased compression error, leading to longer training and lower accuracy.

We introduce Tensor Homomorphic Compression (THC), a novel bi-directional compression framework that enables the direct aggregation of compressed values and thus eliminating the aforementioned computational overheads. Moreover, THC is compatible with in-network aggregation (INA), which allows for further acceleration. Our evaluation shows that training representative vision and language models with THC reaches target accuracy by 1.40× to 1.47× faster using INA and 1.28× to 1.33× faster using a software PS compared with state-of-the-art systems.

1 Introduction

In the past decade, the scale of machine learning training and data volume has increased dramatically due to the growing demand for various ML applications [11,13,23,54,62,70,72,76]. Alibaba's general-purpose ML platforms also reported a rapid increase in ML training data, from hundreds of gigabytes to tens or even hundreds of terabytes, at an internet scale, within a few years [75]. This trend is expected to continue in the future [59] with the rapid advancements of giant models [11, 13, 44, 52, 60]. To support these large-scale models, we need large-scale distributed training [19, 30, 51, 71].

However, distributed training incurs high communication overhead. Recent research [73] has shown that the synchronization cost of GPT2 [50] and BERT-base [14] in a 8-worker setting can be as high as 42% and 49% of the total time during training, even with state-of-the-art frameworks. As the number of workers increases, the communication overhead rises substantially [59]. Meanwhile, computing devices are pushing more data into the network with specialized ML accelerators [18, 42, 47, 77] and more advanced GPU/TPU hardware, which further increases the communication overhead [65, 80].

To reduce the communication overhead, many compression schemes have been conceived [6, 10, 15, 64, 73, 74]. One common problem of these solutions is that they apply bi-directional compression. For example, in the Parameter Server (PS) architecture, the PS nodes first decompress all gradients, aggregate them, and then compress the aggregated gradients again. Such compression and decompression operations result in significant computational overheads and affect the training convergence time and attainable accuracy (see Section §2.1).

To address these problems, we introduce Tensor Homomorphic Compression (THC), a novel bi-directional compression framework enabling the direct aggregation of compressed tensors (e.g., gradients) without first decompressing them, eliminating much of the aforementioned computational overhead. Additionally, direct aggregation can also run on programmable switches for further acceleration.

This paper focuses on developing a THC framework to reduce the communication overhead of data parallelism for the parameter server architecture. Importantly, PS is effective in GPU/CPU hybrid clusters [28, 39, 46, 73], which are common in clouds [27]. Furthermore, when we colocate a PS with each worker, it essentially functions as an AllReduce [28].

From an algorithmic standpoint, the technical challenge is designing an algorithm that enables workers to accurately compress their gradients in a way that allows aggregating their results without decompressing each worker's message. We propose *Homomorphic Compression* – a property that enables this and develops efficient schemes that satisfy it while optimizing the accuracy.

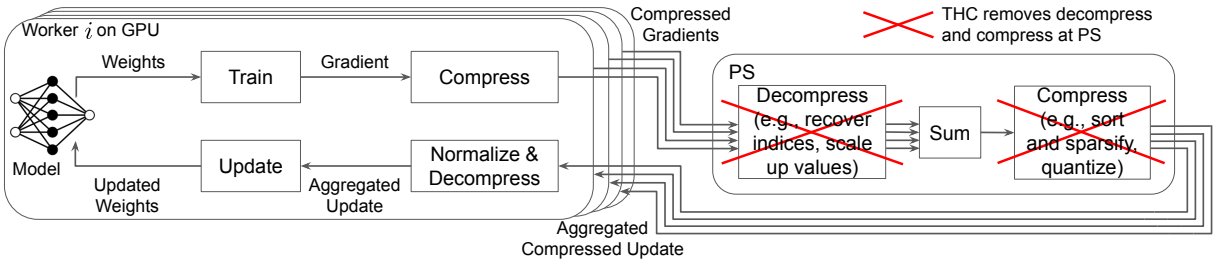
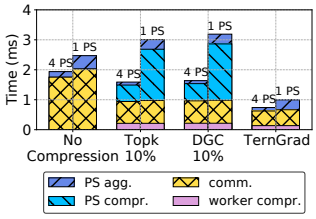
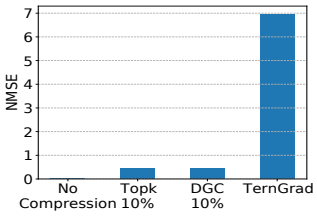


Figure 1: End-to-end workflow when using typical compression algorithms.



(a) Communication round time of one 4MB partition.



(b) NMSE of various compression schemes with four workers.

Figure 2: Communication time and error.

A key idea behind THC's ability to satisfy this property is an initial communication stage with minimal information exchange between the workers at the beginning of each round that allows them to *coordinate* and ensure that their compressed gradients are directly aggregable. To ensure high accuracy, THC does not directly encode the gradients but rather pre- and post-processes them with the GPU-friendly Randomized Hadamard Transform (RHT) to transform the gradients to a different representation that is amenable to accurate quantization. Since RHT preserves the tensor sizes (i.e., norms), by merely exchanging these norms during a preliminary light communication stage (a single float per client), the clients can align their quantization values such that they can be averaged without decompression. Furthermore, this communication step overlaps with applying the RHT transform and thus does not increase the compression time. THC also employs an advanced non-uniform quantization technique we developed and an error-feedback mechanism to further improve the bandwidth to accuracy tradeoff.

We built THC on top of the BytePS [28] PyTorch extension. Our PS can run on either software or programmable switches. We perform extensive evaluation over seven representative DNN models on a local testbed and AWS EC2. Testbed results show that THC achieves the target accuracy 1.42× to 1.47× faster with aggregation on a programmable switch and 1.28× to 1.33× with a software PS, compared to the state-of-the-art distributed training framework (Horovod RDMA). THC with the programmable switch also improves the training throughput by up to 54% over Horovod RDMA. [1]

[1] THC is available at https://github.com/SophiaLi06/BytePS_THC.

2 Background and Motivation

In this section, we give the background on compression and in-network aggregation and motivate the need for direct aggregation on compressed data. Gradient compression, a well-studied approach to lower the network communication of distributed training, reduces the volume of gradients transmitted in synchronization steps at the expense of convergence speed and accuracy. There are two key techniques: sparsification and quantization. Sparsification may filter out coordinates of small magnitude in the gradient tensor. TopK [64] is a straightforward sparsification algorithm that only sends the top k percent (by magnitude) of coordinates and their indices. Sparsification becomes lossy when many coordinates are nonzero. Quantization reduces the size of each element by reducing the precision of gradients. For example, TernGrad [74] reduces the bit length of the gradient coordinates to two bits by converting each float into a value $x \in \{-1, 0, 1\}$. Different compression techniques offer various accuracy, bandwidth, and time complexity tradeoffs.

2.1 Compression Cost and Tradeoffs

Figure 1 shows the bi-directional compression process in existing PS systems [6, 73]. Workers send compressed gradients to the PS. The PS decompresses and aggregates gradients. Then, to reduce the traffic back to workers, the PS compresses the aggregated results again before transmission. In such a design, while compression reduces the communication cost, decompressing and compressing data on PS nodes introduce high compute overhead and additional compression errors.

To quantize the computational overhead and estimation error of compression schemes, we run a microbenchmark transmitting a single 4MB partition (the recommended partition size that balances pipelining efficiency and system overheads as specified in BytePS repository [12]) on our local testbed. Training frameworks usually batch gradients and chunk them into same size partitions before communication [35, 46]. Since communication time grows linearly with the number of partitions, we simply measure the times for a single partition in our microbechmark. We measure the worker-side *compression and decompression time ("worker compr.")*, the PS side *compression and decompression time ("PS compr.")*, the PS *aggregation time ("PS agg.")* and the worker-PS *communication time ("comm.")* as shown in Figure 2a.

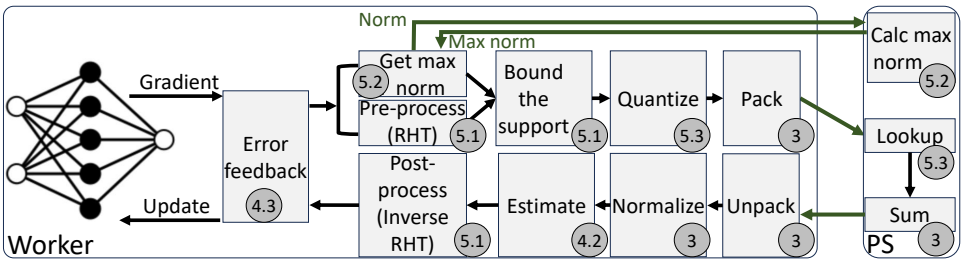


Figure 3: An illustration of THC, with the section numbers of each component.

With one PS and four workers, two sparsification schemes TopK 10% and DGC 10% [38] that communicate the top 10% coordinates by magnitude slow down the end-to-end time by 19.3% and 27.1% of the no-compression round time. This is due to the high PS computational overhead of compression and decompression that contributes up to 56.9% of the round time even when the communication time is reduced. Note that the computational overhead of TopK and DGC grow as the PS aggregates more coordinates, making it a poor choice when the worker-to-parameter server ratio is imbalanced.

When we use colocated PS (i.e., have four PS in total), TopK 10% reduces the communication time by 60.4% of that of no compression but takes an extra 0.54 ms (34.0% of the round time) to run compression/decompression on the PS. The end-to-end round time reduction is therefore diluted to 20.6% of that of no compression.

One can use other compression schemes (e.g., TernGrad) that take less time for decompression/compression at the PS side. However, these schemes have larger quantization errors. Figure 2b shows the NMSE (Normalized Mean Squared Error, $NMSE(x,\hat{x}) = \frac{||x-\hat{x}||_2^2}{||x||_2^2}$), which quantifies the difference between the actual vector x and the vector restored after compression $\hat{x}$. TernGrad results in an NMSE that is by an order of magnitude larger than that of TopK 10% (i.e., 6.95 vs. 0.46 with four workers). This large gap in NMSE means that TernGrad requires more iterations to reach the target accuracy or might fail to reach the target accuracy at all. In fact, provable convergence rates for distributed SGD have a linear dependence on NMSE (e.g., [29]), rendering quantization schemes with large NMSE less appealing for distributed training.

To address these limitations, THC allows direct aggregation of compressed gradients, which eliminates decompression and compression operations at PSes while ensuring high accuracy. A detailed comparison between THC and other compression schemes under different bit budgets is in Section §8.4.

2.2 In-network Aggregation

In-network aggregation [45] is another option to reduce communication overhead. Recent research [33, 57] has demonstrated that programmable switches can aggregate gradients from multiple workers, reducing the switch-to-PS traffic and resulting in a substantial training performance improvement. However, using switches does not reduce the traffic volume generated by the workers as gradient compression does.

Most existing compression solutions are incompatible with in-network aggregation solutions because switches can not easily decompress and compress the data due to their programmability and resource limitations [33, 49]. Since THC supports direct aggregation over compressed data, it only requires the PS to do summation, which programmable switches can readily perform with their ALUs. This also offers new opportunities for incorporating compression with in-network aggregation to further improve training performance.

3 THC Overview

We propose the *Tensor Homomorphic Compression (THC)* framework, which allows the PS to merely aggregate the incoming compressed gradients and transmit the (still compressed!) aggregated values back to the workers. THC hence enables us to avoid the computational overhead of compression and decompression at PS while still having accurate estimation of the gradients' average. We first introduce the homomorphic compression property to model such system constraints. Consider n workers and let ∇_i be the i-th worker's gradient. We define the Uniform Homomorphic Compression (UHC) property as follows:

Definition 1 (Uniform Homomorphic Compression)

$$\widehat{\nabla}_{avg} = \frac{1}{n} \cdot \sum_i \texttt{Decompress}(\texttt{Compress}(\nabla_i))$$
$$= \texttt{Decompress}\left(\frac{1}{n} \cdot \sum_i \texttt{Compress}(\nabla_i)\right).$$

That is, with the Uniform Homomorphic Compression property, the average of the decompressed gradients is mathematically equivalent to decompressing the average compressed gradient. Leveraging this property, the PS simply sums the compressed gradients and sends the result back (still in compressed form). Finally, each worker averages the result and applies the decompression to derive the update $\widehat{\nabla}_{avg}$.

The key challenge for THC is to design compression algorithms that retain the UHC property while ensuring high accuracy. Non-homomorphic compression techniques require the PS to decompress the gradients before aggregation because they rely only on worker-local information. When workers use different quantization ranges (e.g., [4, 53]), the PS must decompress each gradient separately, sum them up, and compress again, increasing processing delay at the PS side and

the errors caused by compression. We know of one previous compression scheme, SignSGD [10], that is homomorphic as it simply counts, for each coordinate, the number of workers which had a positive value for it. However, this scheme is biased, and thus its error does not decrease with the number of workers, making it yield large errors in practice. In THC, we retain an accuracy that is similar to that of the uncompressed baseline and achieve the UHC property.

We present the THC framework, illustrated in Figure 3. The THC workflow starts by having workers compress their gradients, which commonly consist of 32-bit floats. Each coordinate is quantized and then encoded into a *table index* (formally introduced in Section §4.2) that requires a small number of bits. The table indices are then packed and sent to the PS. The PS looks the table indices up in a lookup table to restore the corresponding *table values* and sums up the looked-up table values coordinate-wise. After summing values from all workers, PS packs the result and broadcasts it. Finally, each worker decompresses and normalizes the result to obtain the average gradient's estimate.

The table here serves two purposes: first, it allows an efficient expansion of the indices to wider values that allow summation without overflows. Second, we can use the table to minimize the quantization error, as we later show, by picking the table values in correspondence to the underlying data distribution. Since the table is small (of size 2^b, where b is a small constant) and hardcoded (does not depend on the gradients or number of workers), and lookups do not require arithmetic operations, we consider it as part of the direct aggregation.

Figure 4 visualizes the THC implementation we adopt in our system prototype. Namely, each 32-bit coordinate is encoded into a 4-bit *table index*, which then gets converted into a 8-bit *table value* on the PS. The broadcast summation result also uses 8 bits per coordinate. Therefore, our system prototype provides a $\times 8$ bandwidth reduction from workers to the PS and a $\times 4$ bandwidth reduction in the other direction.

The THC algorithm is described in detail in Section §4; in Section §5, we introduce further optimizations for THC and the end-to-end training procedure; then, Section §6 shows how THC can be seamlessly used in conjunction with in-network aggregation to further accelerate the training.

4 Tensor Homomorphic Compression

In this section, we explain how to achieve the UHC property effectively. We start by giving background on stochastic quantization, a core building block of our THC approach. We then show (Section §4.2) that stochastic quantization with uniform intervals, a technique that has been used previously in compression, has the UHC property. However, its performance in terms of the accuracy per bit is relatively poor, because it is unoptimized. This poor compression performance can lead to worse training time and/or model accuracy than an uncompressed baseline. We, therefore, introduce further optimizations that improve the compression performance. Our

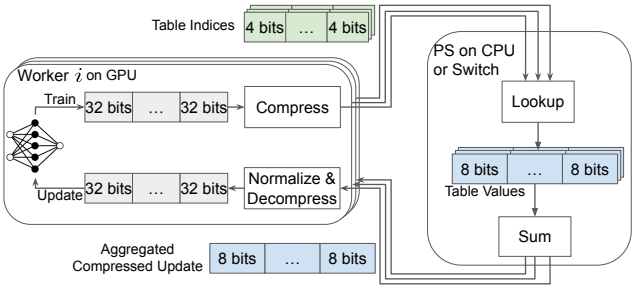


Figure 4: THC's workflow illustration.

main conceptual advance is to use non-uniform quantization intervals while maintaining the UHC property; that is, we show how to optimize the choice of quantization values in Section §4.3. Finally, Section §5 provides important technical optimizations for speed and accuracy.

4.1 Background on Stochastic Quantization

A main building block that is used for gradient compression is *quantization*, a technique that allows representing gradient entries (e.g., 32-bit floats) using a small (e.g., 4) number of bits while bounding the error. At a high level, our Tensor Homomorphic Compression (THC) framework leverages Stochastic Quantization (SQ) that rounds a given real-value a to one of two quantization values q_0, q_1 such that $q_0 \leq a \leq q_1$. SQ quantizes a to q_0 with probability $(q_1 - a)/(q_1 - q_0)$ and to q_1 otherwise. An appealing property of SQ is that the expected value of the quantization is exactly a, i.e., it is unbiased. This is especially useful in distributed scenarios where using SQ results in a decrease in the error of estimating the average as the number of workers increases [68]. Our focus in what follows is minimizing the error introduced by quantization while maintaining the UHC property.

4.2 The Uniform THC Algorithm

As we now show, it turns out that using a variation of SQ with uniform intervals *where all workers use the same set of intervals* already yields a solution that is both unbiased and homomorphic. However, the accuracy per-bit of this solution leaves much to be desired, which is where we focus our efforts in the following sections.

The most popular form of SQ is *Uniform* SQ (USQ), in which the quantization values are uniformly spaced. For example, given the range $[m, M]$ and using 2^b quantization values, their locations are $q_0 = m, q_1 = m + (M-m)/(2^b - 1), \ldots, q_{2^b} = M$. USQ quantizes a value $a \in [m, M]$ to one of its nearest quantization values. We first show that USQ, when all workers globally use the same range $[m, M]$ and b, satisfies the UHC property. Note that for implementation this requires all workers to first obtain the global minimum and maximum to perform quantization, which is different than the standard use of USQ (where each worker quantizes based on their own local minimum and maximum value). For convenience, we henceforth denote $\langle i \rangle = \{0, \ldots, i-1\}$ for any $i \in \mathbb{N}$.

Algorithm 1 Uniform THC

Input: bit budget b

Preliminary stage

Worker i:
1: Compute $m_i = \min\{\nabla_i\}$ and $M_i = \max\{\nabla_i\}$
2: Send (m_i, M_i) to the PS
PS:
3: Compute $m = \min_i\{m_i\}$ and $M = \max_i\{M_i\}$
4: Send (m, M) to workers

Main stage

Worker i:
5: Compute $X_i = USQ(\nabla_i, m, M, b)$
6: Send X_i to PS
PS:
7: Compute $X = \sum_{i\in\langle n\rangle} X_i$
8: Send X to workers
Worker i:
9: Estimate $\widehat{\nabla}_{avg} = m + \frac{1}{n}\cdot X \cdot \frac{M-m}{2^b-1}$

Definition 2 (Homomorphic USQ) *Let $\nabla_0, \nabla_1, \ldots, \nabla_{n-1} \in \mathbb{R}^d$ be the input gradients, and let $m_i = \min\{\nabla_i\}$ and $M_i = \max\{\nabla_i\}$ be the i'th gradient's minimum and maximum. Let $m = \min_i\{m_i\}$ and $M = \max_i\{M_i\}$ and consider a set of uniformly spaced quantization values $\left\{m + k\cdot\frac{M-m}{2^b-1} \mid k \in \langle 2^b\rangle\right\}$. The workers perform stochastic quantization using these quantization values on all input gradients.*

This approach is homomorphic (Definition 1) since (C and D stand for `Compress` and `Decompress`):

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(C(\nabla_i)) = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle}\left(m + C(\nabla_i)\cdot\frac{M-m}{2^b-1}\right)$$
$$= m + \left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} C(\nabla_i)\right)\cdot\frac{M-m}{2^b-1} = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} C(\nabla_i)\right).$$

With this primitive at hand, we introduce a simplified (uniform) variant of the THC framework, which we generalize to a non-uniform setting to obtain better performance.

The pseudo-code of Uniform THC is given by Algorithm 1. It begins with a preliminary communication round where each worker computes and sends the smallest and largest gradient entry to the PS (lines 1-2). In turn, the PS computes the extreme global values and distributes them back to the workers (lines 3-4). This communication round is light and requires each worker to transmit and receive only two floats. Next, each worker quantizes its gradient using the global extremes (i.e., perform Homomorphic USQ) and sends the result to the PS (lines 5-6). Then, the PS sums all the quantized vectors and sends their sum to the worker (lines 7-8). Finally, each worker divides the sum by the number of workers to obtain the estimate of the average (line 9).

As detailed in Section 5, we can reduce the quantization error by pre-processing the input gradient prior to its quantization and post-processing the average's estimate at the end.

4.3 The Non-uniform THC Algorithm

In many cases, it is possible to choose the quantization values in a non-uniform manner to optimize the accuracy-bandwidth tradeoff. However, it is unclear how to leverage existing non-uniform quantization methods (e.g., [8, 9, 53, 66, 67]) to improve our homomorphic compression. Namely, in non-uniform methods, the sender transmits a *table index* z that is then converted to the *table value* $T[z]$ by the receiver. Here, T is a *lookup table* that converts indices to values that may not be uniformly spaced, i.e., $T[z+1] - T[z]$ may not equal $T[z'+1] - T[z']$ for different indices z, z'.

For example, consider quantizing values in the range $[-1, 1]$ using four quantization values. Using USQ, the lookup table is $T_0[0] = -1, T_0[1] = -1/3, T_0[2] = 1/3, T_0[3] = 1$. In this case, any sum of table indices corresponds to a single sum of quantization values. For example, suppose two senders send indices z, z' and consider two cases: (1) $z = 0, z' = 3$ and (2) $z = 1, z' = 2$. In both cases, $T_0[z] + T_0[z'] = 0$. That is, $z + z' = 3$ implies that $T_0[z] + T_0[z'] = 0$. Intuitively, the receiver can sum the indices and deduce the sum of sent values instead of performing two table lookups.

In contrast, consider non-uniform quantization that uses the table $T_1[0] = -1, T_1[1] = -1/2, T_1[2] = 1/2, T_1[3] = 1$. Consider the two cases in which $z + z' = 4$: (1) $z = 1, z' = 3$ and (2) $z = z' = 2$. In the first case, $T_1[z] + T_1[z'] = 1/2$ while in the second $T_1[z] + T_1[z'] = 1$. That is, the sum of table indices does not determine the sum of table values.

Our insight is that we can overcome this by using a subset of the uniformly spaced quantization values. To that end, for a bit-budget of b bits per coordinate, we define a hyperparameter $g \geq 2^b - 1$ called *granularity*. We then use a table $T : \langle 2^b\rangle \to \langle g+1\rangle$ that maps table indices to values that are integers in the larger range. Intuitively, one can think about this as running USQ with $g+1$ quantization values, but where all senders are only allowed the same 2^b indices. A table value $T[z]$ then corresponds to the quantization value $m + T[z]\cdot\frac{M-m}{g+1}$, same as in USQ. For instance, in the above example of quantizing values in the range $[-1, 1]$, the PS can use a table $T_2[0] = 0, T_2[1] = 1, T_2[2] = 3, T_2[3] = 4$ to map each table index into (the larger) range $\langle 5\rangle$. The benefit is that the table values are now directly aggregable; for example, consider three senders with the two cases (1) $z = z' = z'' = 1$ and thus $T_2[z] = T_2[z'] = T_2[z''] = 1$, i.e., all quantization values are $-1 + \frac{1-(-1)}{4} = -\frac{1}{2}$; (2) $z = z' = 0, z'' = 2$ and thus $T_2[z] = T_2[z'] = 0, T_2[z''] = 3$, i.e., the first two quantization values are -1 and the third is $-1 + 3\times\frac{1-(-1)}{4} = \frac{1}{2}$. Notice that the sum of table values is the same in both cases, and so is the sum of quantization values; in contrast, the sum of table indices differ.

Intuitively, g introduces a tradeoff where larger g results in more fine-grained quantization values and lower error but also requires more bits to represent the summation and higher bandwidth requirement from the PS or switch to the workers.

The pseudocode for the non-uniform variant of THC appears in Algorithm 2 (the Preliminary stage is the same as in Algorithm 1). Notice that the lookup table $T_{b,g}$ depends on bit-budget b and the granularity g. When clear from context, we omit the subscripts and write T.

Each worker i then calculates the set of quantization values $Q \subset [m,M]$ given the (global) m,M and the granularity g (Line 1). Next, it stochastically quantizes each coordinate to a value in Q, giving the result $X_i \in Q^d$ (Line 2). The worker then transforms each quantized coordinate into its uniform-HC value in $\langle g+1 \rangle$ using the linear transformation (Line 3). The next stage involves computing which b-bit table indices (in $\langle 2^b \rangle$) would map to the uniformly partitioned values by applying the inverse mapping (Line 4). Finally, it sends the result Z_i to the PS (Line 5).

The PS then gets the set of vectors $\{Z_i\}_{i\in\langle n\rangle}$. It looks up each vector (coordinate-wise) and sums them up, corresponding to the sum of the Y_i's (Line 6). Then, the PS sends the result, Y, to the workers, still in compressed form (Line 7). Observe that the PS only performs table lookups and summation without decompressing the vectors and without additional processing and re-compression that increases the error.

The final part takes place in parallel, where each worker i first computes the average of $\sum_{i\in\langle n\rangle} Y_i$ by dividing Y by n (Line 8). It then applies the inverse transformation to Line 3 to obtain an estimate $\widehat{\nabla}_{avg}$ of the average gradient (Line 9).

To show that our algorithm is homomorphic, we generalize Definition 1 to account for the lookup table (C,D and T stand for `Compress`, `Decompress` and `table lookup`)

Definition 3 (Non-uniform homomorphic compression)

$$\widehat{\nabla}_{avg} = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(C(\nabla_i))) = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} T(C(\nabla_i))\right).$$

That is, a non-uniform HC (NUHC) scheme generalizes UHC by allowing the PS to apply a lookup table T to the compressed gradients before aggregating them.

Notice that if $g = 2^b-1$ and T is the identity mapping, NUHC is identical to UHC (in that case, the lookup table is redundant). As shown above, for T that is the identity mapping, the compression is uniform homomorphic. We now show that if $0 = T(0) < T(1) < \ldots < T(2^b-1) = g$ then Algorithm 2 is homomorphic. [2] This is because

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(C(\nabla_i))) = \frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(T(Z_i)) =$$

$$\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} D(Y_i) = \frac{1}{n}\cdot\left(\sum_{i\in\langle n\rangle} m + Y_i\cdot\frac{M-m}{g}\right) =$$

$$m + \left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} Y_i\right)\cdot\frac{M-m}{g} = D\left(\frac{1}{n}\cdot\sum_{i\in\langle n\rangle} T(C(\nabla_i))\right),$$

where we used (i) $C(\nabla_i) = Z_i$, (ii) $T(C(\nabla_i)) = T_{b,g}[Z_i] = Y_i$, and (iii) $D(Y_i) = m + Y_i\cdot\frac{M-m}{g}$.

[2] In fact, it is sufficient that T is injective and satisfies $0, g \in Im(T)$; the above definition is without loss of generality.

Algorithm 2 Non-uniform THC

Input: bit budget b, granularity g, and their lookup-table $T_{b,g}$

Main stage ▷ Preliminary stage same as in Algorithm 1

Worker i:

1: Compute $Q = CalcQuantizationValues(m,M,T_{b,g})$
2: Compute $X_i = SQ(\nabla_i, Q)$
3: Compute $Y_i = (X_i - m)\cdot\frac{g}{M-m}$ ▷ $Y_i \in \langle g+1\rangle^d$
4: Compute $Z_i = T_{b,g}^{-1}[Y_i]$ ▷ $Z_i \in \langle 2^b\rangle^d$
5: Send Z_i to PS

PS:

6: Compute $Y = \sum_{i\in\langle n\rangle} T_{b,g}[Z_i]$ ▷ $Y \in \langle g\cdot n+1\rangle^d$
7: Send Y to workers

Worker i:

8: Compute $Y_{avg} = \frac{1}{n}\cdot Y$ ▷ $Y_{avg} \in [0,g]^d$
9: Estimate $\widehat{\nabla}_{avg} = m + Y_{avg}\cdot\frac{M-m}{g}$

The above result shows that Algorithm 2 is homomorphic for many choices of lookup tables. As we later demonstrate, very small lookup tables (e.g., for $b=4$ we can usually use $g \in \{16,\ldots,51\}$) are sufficient to obtain accurate quantization.

5 Optimizing THC

We next describe optimizations that improve THC's bandwidth-accuracy tradeoff (§5.1-§5.2) and speed (§5.3).

5.1 Pre- and Post-processing Using the Randomized Hadamard Transform

For pre-processing, we utilize the Randomized Hadamard Transform (RHT) that improves quantization accuracy by reducing the expected range and transforming coordinates to approach a normal distribution. The RHT of a vector $x \in \mathbb{R}^d$ is defined as $\frac{1}{\sqrt{d}}\cdot H\cdot D\cdot x$, where H is the Hadamard matrix [25] and D is a diagonal matrix with i.i.d. Radamacher variables (taking ± 1 with equal probabilities) on its diagonal. An important property RHT is that the special recursive structure of H allows a fast GPU-friendly $O(d\log d)$ time implementation, significantly faster than general matrix multiplication. The post-processing is then the inverse transform, i.e., $RHT^{-1}(x) = \frac{1}{\sqrt{d}}\cdot D\cdot H\cdot x$, which has identical complexity to RHT.

RHT has two key benefits: First, RHT reduces the range of coordinate values, improving accuracy. More concretely, a key quantity that determines the error of a quantization scheme is its range (the difference between the largest and smallest quantization values), $M-m$ (see Appendix A.2 for more details). Intuitively, when the range is large, one is forced to quantize to values that are further away from the encoded quantity, thus increasing the error. As proven by [3], and using M', m' to denote the maximal and minimal value after the RHT transform, $\mathbb{E}[M'-m'] = O\left((M-m)\cdot\sqrt{\log d \;/\; d}\right)$. This decrease in the expected range significantly improves the quantization accuracy.

To further decrease the range, and thus the quantization error, we leverage known results about the distribution of the transformed coordinates to derive a threshold t_p, for an appropriate $p \in (0,1)$, such that approximately a p fraction of the transformed coordinates are expected to fall outside $[-t_p, t_p]$. Namely, it is known that each coordinate in the RHT of a vector $x \in \mathbb{R}^d$ follows a distribution that approaches (for a large enough d) the normal distribution $\mathcal{N}(0, 1/\|x\|^2)$ [68]. In particular, this means that the probability of a coordinate landing outside the range diminishes exponentially in t_p, giving an opportunity to significantly reduce the range at the expense of a small bias and allowing us to pick an appropriate p, as we describe below in § 5.3. Our algorithm then optimizes the quantization values to minimize the error of the coordinates in $[-t_p, t_p]$, and truncates the rest by rounding those larger than t_p to t_p and those smaller than $-t_p$ to $-t_p$.

To address this small bias, we compensate for it using a technique called error-feedback (EF) [29], which includes sending the vector $x = \nabla + e$ that adds the previous error e to the current gradient ∇, and later updating e to account for the quantization error. It is known that when the bias is not too large, EF guarantees the convergence of the training [36]. The effect of rotation and error feedback is evaluated in detail in Appendix D.3, along with comparison to uniform THC.

The second key benefit of RHT is that, since the distribution of the coordinates after applying RHT is known, we can compute the optimal lookup table T *offline*, as we explain next.

5.2 Constructing the Optimal Lookup Table

As explained, after applying RHT and truncating, our goal is to design a lookup table that minimizes the quantization error of the resulting vector. This vector has two types of coordinates: (1) *the truncated coordinates* have no error (beyond the truncation) since, by design, there are always quantization values at $\{-t_p, t_p\}$; (2) the *non-truncated coordinates* have a distribution that approaches the truncated normal distribution. Intuitively, this allows us to design the lookup table T to minimize the quantization error of a truncated normal random variable and thereby minimize the NMSE.

Formally, the optimal lookup table needs to minimize the error in quantizing a truncated normal random variable $(A \sim \mathcal{N}(0,1) \mid A \in [-t_p, t_p])$, where $t_p = \Phi^{-1}(1 - p/2)$ and Φ is the CDF of the normal distribution. As we later show, we can use this pre-computed table for a normal random variable *with any variance* by scaling the quantization values.

Formally, denoting by $P(a,z)$ the probability of sending the index $z \in \langle 2^b \rangle$ given a value $a \in [-t_p, t_p]$, the optimization problem aims to find the table T and probabilities P. Let us further denote by ϕ the pdf of the normal distribution. Then, the following optimization problem gives the optimal lookup table as $T_{b,g,p}[z] = T[z]$:

$$\underset{P,T}{\text{minimize}} \int_{-t_p}^{t_p} \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot (a - T[z])^2 \cdot \phi(a) \cdot da$$

subject to

$$(\textit{Unbiasedness}) \quad \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot T[z] = a \qquad \forall a \in [-t_p, t_p]$$

$$(\textit{Probability}) \quad \sum_{z \in \langle 2^b \rangle} P(a,z) = 1 \qquad \forall a \in [-t_p, t_p]$$

$$P(a,z) \geq 0 \qquad \forall a \in [-t_p, t_p], z \in \langle 2^b \rangle$$

$$(\textit{Granularity}) \quad T[z] \in \left\{ \tfrac{2t_p}{g} \cdot i - t_p \mid i \in \langle g+1 \rangle \right\} \qquad \forall z \in \langle 2^b \rangle$$

As we elaborate in Appendix B, we optimally solve the above problem. To that end, we wrote a specialized ILP solver that leverages various properties of the above optimization problem to reduce the search space and speed up the computation of T. Recall that for any b, g, p, we compute the optimal $T_{b,g,p}$ table only once offline and thus the solver's runtime does not affect THC's performance.

5.3 Accelerating the Preliminary Stage

Heretofore, we discussed a preliminary stage in which the workers exchange information that depends on the transformed vectors to determine the quantization range. A natural implementation would, therefore, include transforming the vectors using RHT and then exchanging the information required for setting M and m.

Instead, we leverage special properties of RHT, namely that (i) it preserves the *norm* of the transformed vector and (ii) that there is a tight connection between the *maximal norm* and the values M, m we are seeking. Accordingly, each client i first computes the norm of its vectors x_i and then parallelizes the following operations: performing the RHT and obtaining the maximum norm among the workers' gradients using the PS. Then it can set $M = (t_p/\sqrt{d}) \cdot \ell, m = -M$ where $\ell = \max \|x_i\|^2$ and proceed to the quantization.

5.4 Putting It All Together

We are now ready to describe our complete THC training process, whose pseudocode is given Algorithm 3, color coding the different steps. In the first learning step, each worker computes its local gradient and adds the error-feedback. Next, the preliminary stage, in which the clients exchange their norms, is parallelized with the RHT part of the pre-processing; later, the transformed vector is normalized and clamped. Next, during main stage, the workers and PS follow our Non-uniform THC algorithm (see Algorithm 2) to obtain an estimate of the average of the pre-processed vectors. Then, each worker post-processes the estimate using the inverse transform. Finally, in the second learning step, the workers update the error-feedback and model.

Algorithm 3 Training with THC

Input: Support parameter p and its threshold t_p. Bit budget b, granularity g, and lookup-table $T_{b,g,p}$.
for $r = 0, 1, \ldots$ **do**
  for worker $i \in \langle n \rangle$ in parallel **do**
    Compute local gradient, ∇_i
    $x_i = \nabla_i + e_i^r$ ▷ Error-feedback for round r
    in parallel
      (i) **Send** $\|x_i\|$ to PS
        Receive $\ell = \max_i \|x_i\|$ from PS
      (ii) $R_i = \texttt{RHT}(x_i)$ ▷ $R = HD^r x_i$
    end in parallel
    $M = (t_p/\sqrt{d}) \cdot \ell \;\; ; \;\; m = -M$
    $x_i' = \texttt{clamp}(R_i, \min = m, \max = M)$ ▷ Truncation
    $Q = CalcQuantizationValues(m, M, T_{b,g,p})$
    $X_i = SQ(x_i', Q)$ ▷ Stochastic Quantization
    $Y_i = (X_i - m) \cdot \frac{g}{M-m}$ ▷ $Y_i \in \langle g+1 \rangle^d$
    $Z_i = T_{b,g,p}^{-1}[Y_i]$ ▷ $Z_i \in \langle 2^b \rangle^d$
    Send Z_i to PS
    Receive $Y = \sum_{i \in \langle n \rangle} Y_i = \sum_{i \in \langle n \rangle} T_{b,g,p}[Z_i]$ from PS
    Compute $Y_{avg} = \frac{1}{n} \cdot Y$ ▷ $Y_{avg} \in [0, g]^d$
    Estimate $\widehat{x}'_{avg} = m + Y_{avg} \cdot \frac{M-m}{g}$
    Compute $\widehat{\nabla}_{avg} = \texttt{RHT}^{-1}(\widehat{x}'_{avg})$ ▷ Global update
    $e_i^{r+1} = x_i - \texttt{RHT}^{-1}(X_i)$ ▷ Quantization error
    Update model using $\widehat{\nabla}_{avg}$
  end for
end for

6 THC with Other System Opportunities

In this section, we discuss how THC leverages the opportunities of in-network aggregation and explore potential optimizations to address issues such as packet loss and stragglers.

Aggregation at Programmable Switches In our THC framework, we can offload PS completely to programmable switches for further hardware acceleration. Our THC design simplifies the PS by removing the compression and decompression operations. Since THC already compresses floating-point gradients to integer table indices, it fits programmable switches well. We do not need additional conversions from floating points to integers at workers as used in previous work [33, 57, 79].

Packet Loss and Stragglers THC is a compression algorithm that tolerates data loss, allowing it to ignore the tail outliers that can negatively impact performance during training introduced by packet loss [69] or straggler problems [22, 78].

Packet losses and stragglers may occur between workers and the PS. For each gradient, if a worker doesn't receive the corresponding aggregation result packet within a specified time threshold, the worker could fill in the missing data with zeros and continue with the received aggregation results. This practice may lead to some workers updating their models with different information. To mitigate the impact, we can implement a synchronization scheme, where workers coordinate their model parameters after every epoch by choosing to copy the parameters of another worker when encountering severe packet loss. Our simulation results shows that there is no significant impact on THC model accuracy or convergence within the reasonable data center packet loss rate range (less than 1% [21, 37]) (see Section §8.4). This result aligns with the observations in [69].

To handle packet losses from workers to the PS, we can perform partial aggregation: the PS broadcasts partial aggregation results once it hears from the majority (e.g., 90%) of workers. We evaluate the impact of stragglers and partial aggregation in Section §8.4 and show that THC with partial aggregation over 90% of workers reach the baseline accuracy.

7 Implementation

THC Worker. We develop THC prototype of workers atop BytePS's PyTorch extension [28]. During each iteration, the BytePS worker receives the calculated gradient from the front-end PyTorch and passes it to our *compression module*. Our compression module runs the THC algorithm on GPUs and employs a GPU-friendly implementation of RHT. It also keeps the error-feedback records to compensate for the biased quantization as mentioned in Section 5.

The communication module is a C++ module developed based on Data Plane Development Kit (DPDK), which provides kernel bypassing so that applications can directly receive data from the NIC using busy polling. The communication module assembles packets based on compression results and communicates with the PS.

Another approach is to adopt the RDMA protocol, such as RoCEv2 [5]. However, adopting the RDMA protocol requires additional header parsing functions on the switch side. SwitchML [57] has demonstrated that the RoCEv2 protocol can be used with in-network aggregation by carefully parsing the header. For THC, we consider this as future work.

THC Parameter Servers. We implement two versions of parameter servers: the software version written in C++ and the programmable switch version implemented on the Intel Tofino switch [26]. In the programmable switch version, the PS performs table lookup using the "Table" control block. After the table lookup, the switch sends packets carrying table values through recirculation ports. The "Register" extern then takes care of value aggregation. Please see Appendix C.2 for the resource usage of the programmable switch PS.

8 Evaluation

We evaluate our THC prototype by training popular computer vision and language models on a local four-worker testbed and AWS EC2. Experiments show that employing THC gives shorter time-to-accuracy (1.40× to 1.47× speedup) and higher throughput (25% to 54% increase) on our local

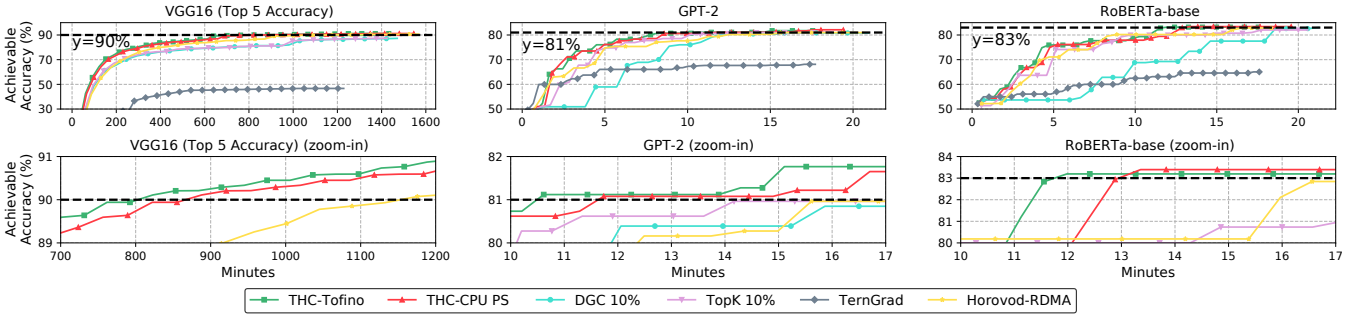


Figure 5: Time to accuracy (TTA) comparison over one image processing task (i.e., VGG16) and two NLP tasks (i.e., GPT-2 and RoBERTa-base). The second row zooms in on the competitive results.

testbed with a programmable switch, compared to state-of-the-art training frameworks. THC also outperforms DGC-10%, TopK-10%, and TernGrad in time-to-accuracy as it has little impact on accuracy and eliminates the overhead at the PS.

Testbed Setup. Our local testbed has four GPU machines as workers, each with one NVIDIA A100 GPU and one NVIDIA MCX516A-CCAT ConnectX-5 100G dual-port NIC connecting to a Tofino2 switch. For large-scale experiments, we use eight AWS EC2 p3.16xlarge instances, each with eight NVIDIA V100 GPUs and 25 Gbps network bandwidth.

Systems for Comparison. We run three versions of THC: (1) with a programmable switch as the PS (labeled as *THC-Tofino*), (2) with the software PS process running on a single stand-alone CPU machine (connected to the Tofino2 with a ConnectX-5 100G dual-port NIC) (*THC-CPU PS*), and (3) with colocated PSes for each worker (*THC-colocated*), which build on BytePS's PS architecture and uses BytePS's RDMA module for a fair comparison. We compare THC with two state-of-the-art systems: (1) Horovod which uses RDMA for communication (*Horovod-RDMA*) and (2) BytePS with colocated PSes for each worker (*BytePS*). We also compare THC against three popular compression algorithms: DGC [38] and TopK [64] are sparsification algorithms that only communicate the top k% of coordinates by magnitude (here we set $k = 10$ and refer to them as *DGC 10%* and *TopK 10%*); and TernGrad [74], a quantization algorithm that converts each coordinate into a value $x \in \{-1, 0, 1\}$ (we refer to it as *TernGrad*). TernGrad represents a stream of quantization algorithms [4, 10] with small differences in design. *BytePS*, *DGC 10%*, *TopK 10%*, and *TernGrad* all use BytePS's colocated PSes and RDMA module. We also tried Espresso [73] but faced convergence issues.[3]

On AWS EC2, we deploy THC with software PS built on top of BytePS [28] servers. We compare THC against BytePS with colocated PS (*BytePS*) and Horovod [58]. All systems use the TCP protocol to communicate. Unless noted otherwise, we use the following THC configuration: granularity 30, p-fraction 1/32, and 16 quantization levels. This configuration avoids overflow for up to eight workers, saturates the worker to PS bandwidth of our system prototype, and consistently achieves high accuracy across various models.

Workloads. We evaluate THC with network-intensive [33, 57] computer vision models (VGG16 and VGG19 [62]) and language models (RoBERTa-base, RoBERTa-large [40], Bart-large [34], BERT-base [14], and OpenAI GPT-2 [50]). We train the vision models with the ImageNet1K dataset [55] and train the language models with the GLUE (General Language Understanding Evaluation benchmark) SST2 (The Stanford Sentiment Treebank) task [63]. Unless noted otherwise, we set the per-GPU batch size as 32. We include results for computation-intensive models that don't benefit much from accelerated inter-machine communication in Appendix D.

Metrics. We first measure time-to-accuracy (TTA) as the training time needed to reach a target validation accuracy. We set the target accuracy based on the convergence of our no-compression baseline. We then present the training throughput (images per second or tokens per second referred to as *samples per second*) for all training tasks. We also show the breakdown of computation and communication time to highlight the factors that contribute to THC's improvements.

8.1 End-to-End Training Performance

Time-to-accuracy. Figure 5 shows that for GPT-2, THC-Tofino reaches the 81% target accuracy 1.47× faster than the Horovod-RDMA baseline, and THC-Software PS reaches the target accuracy 1.33× faster. For RoBERTa-base, THC-Tofino achieves the 83% target accuracy 1.43× faster than the Horovod-RDMA baseline; and THC-Tofino also achieves a 1.40× speedup to reach the 90% target accuracy for VGG16. Note that even though our system prototype uses DPDK, which has similiar performance with RDMA, we still notably outperform Horovod-RDMA. THC-software PS reduces network communication with minimal impact on model convergence, thanks to the optimizations explained in Section 5. Using the programmable switch (THC-Tofino) further accelerates the training by reducing the volume of transmitted data through in-network aggregation.

[3] We followed the instructions in Espresso's git repository and installed all desired versions of software. Unfortunately, we couldn't get models to converge (training loss became "nan" within three iterations and the accuracy stalled around 0.1%). We contacted the authors but they do not have time to fix it. See https://github.com/zhuangwang93/Espresso/issues/3.

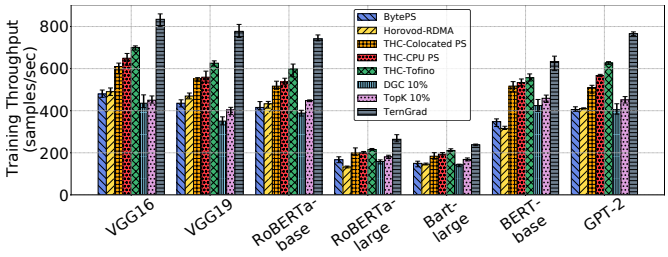


Figure 6: Training throughput with 100Gbps links over different network-intensive architectures.

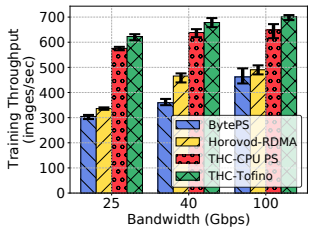


Figure 7: Training throughput for different bandwidths.

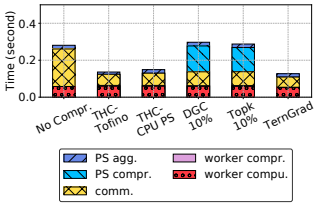


Figure 8: Average training round time breakdown.

Although training with DGC 10% and TopK 10% also approach the target accuracy, they suffer from the high PS compression overhead that leads to longer training epoch time and TTA. TernGrad stalls at low accuracy for all three models despite its high training throughput (Figure 6). This is because TernGrad loses information during its compression and thus cannot improve end-to-end TTA with more training epochs.

Training throughput. To examine the synchronization round time reduction we achieve with THC, we measure the throughput in Figure 6. THC-Tofino provides higher throughput than all alternatives (except TernGrad). For example, THC-Tofino achieves 54% improvement over Horovod-RDMA for GPT-2. THC-colocated has 11% to 37% higher throughput than TopK because THC eliminates the PS-side compression operations. TernGrad provides the highest throughput because it uses fewer bits per coordinate and has shorter PS time and compression overhead. However, TernGrad does not improve TTA as Figure 5 shows due to its low accuracy (Section 8.2).

Effectiveness with different bandwidth. We train the VGG16 architecture under different network bandwidth settings (25, 40, and 100Gbps) in Figure 7. THC-Tofino achieves 1.85×, 1.45×, and 1.43× training throughput over Horovod-RDMA at 25Gbps, 40Gbps, and 100Gbps respectively. When the bandwidth decreases from 40Gbps to 25Gbps, the throughput of Horovod-RDMA drops significantly because it faces more network bottlenecks. Meanwhile, the performance of THC-Tofino and THC-CPU PS downgrades gracefully as the bandwidth decreases, leading to higher training speedups at low bandwidths. We expect THC to offer more benefits under slower networks (e.g., 10Gbps, 1Gbps) which we might encounter in federated learning settings.

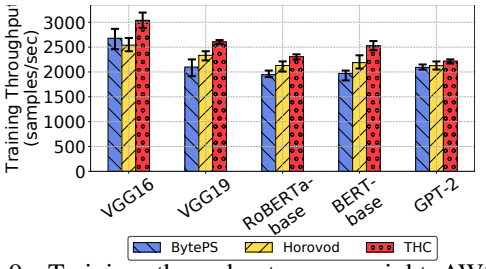


Figure 9: Training throughput across eight AWS EC2 p3.16xlarge instances.

8.2 Breakdown of Network and Compute Time

In Figure 8, we break down the time per iteration into the time spent at the PS, workers, and the communication, when training VGG16 at 100Gbps with THC-Tofino, THC-CPU PS, TopK 10%, and TernGrad. At the PS, we measure the compression time and aggregation time. At the workers, we measure the training time and compression time.

THC-CPU PS reduces the gradient communication time (comm.) to 32.5% of that of the no-compression baseline. As a tradeoff, we introduce compress and decompress operations on the GPU on the worker side. However, these operations only increase the overall worker time by 9.5%. The result demonstrates that in distributed training settings where workers periodically synchronize a large amount of data, saving bandwidth by slightly increasing the worker GPU computation time is worthwhile. THC-Tofino achieves more savings by further reducing communication time through in-network aggregation and offloading the PS to the switches.

For DGC 10% and TopK 10%, they have to run expensive sorting operations on the PS (DGC 10% additionally requires local gradient accumulation), introducing a significant overhead at the PS side. Therefore, although TopK10% gives similar communication time as THC-CPU PS, the overall round time is 46.5% higher than that of THC-CPU PS.

TernGrad uses only two bits per coordinate and requires simple summation at the PS. Thus, it has a short communication and PS time. However, TernGrad produces high NMSE and correspondingly can produce poor TTA results.

8.3 THC Performance on AWS EC2

We measured throughput on AWS EC2 instances equipped with workers containing multiple GPUs at a larger scale (Figure 9). Since AWS instances have 8 GPUs per worker, we have a higher intra-machine communication overhead compared to our local testbed setting. This means that inter-machine communication overhead, which THC optimizes for, takes a smaller portion of training time. Despite this, THC consistently outperformed all the baseline models, resulting in throughput improvements of 1.05× to 1.16×.[4]

[4]Note that Bart-large and RoBERTa-large are not displayed in Figure 9 due to the V100 GPU's memory limitations on the EC2 environment. We used a smaller batch size and reported it in Appendix D separately.

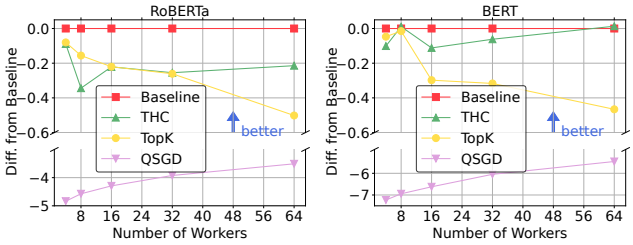


Figure 10: Scalability of THC. Displaying the difference in training accuracy from baseline after two epochs (i.e. if the final training accuracy for 4 worker RoBERTa baseline is 95%, then the accuracy for 4 worker RoBERTa QSGD would be 90%).

8.4 THC Simulation Results

We run simulations to understand THC under different system configurations. As the number of workers increase, the performance of THC increases due to the UHC property. We show that the error of THC scales well from 4 to 64 workers, in contrast to biased compression algorithms like TopK whose error can increase by 9.9× over the same margin. Furthermore, under the synchronization and partial aggregation schemes, THC shows less than a 0.5% drop in training accuracy in the presence of packet loss and straggling workers.

Simulation Environment. We simulate THC on an academic cluster with 4 A100/V100 GPUs per node. Multiple worker training is modeled by storing multiple passes of the backpropagation before performing an update step. This allows us to compress and decompress the aggregated gradient with THC's algorithm (and others) before updating the model, reproducing the communication steps in actual systems.

For the scalability experiments, we train BERT and RoBERTa [40] on SST2 [63] with batch size 8. The configuration uses granularity 36, p-fraction 1/32, and bit budget 4. We choose language tasks for the scalability results because they are more sensitive to small compression errors in the gradient. The other simulations train ResNet50 [23] models on the CIFAR100 [32] dataset with a batch size 128, workers 10, granularity 20, p-fraction 1/512, and bit budget 4.

Scalability. We fine-tune a model for two epochs for each compression algorithm with 4, 8, 16, 32, and 64 workers and then compute the percentage difference from the uncompressed baseline accuracy. We track the difference in accuracy rather than the absolute accuracy because machine learning effects can alter the baseline accuracy as the batch size (number of workers) changes. We compare THC with bit budget 4 against baseline (no compression), TopK [64] and Quantized Stochastic Gradient Descent (QSGD) [4]. QSGD is chosen because we want to compare compression algorithms that have the same compression ratio: QSGD is analogous to an unbiased version of TernGrad/SignSGD with a tunable compression ratio. We choose the k-value and number of intervals of TopK and QSGD respectively to match the overall compression ratio of THC with bit budget 4.

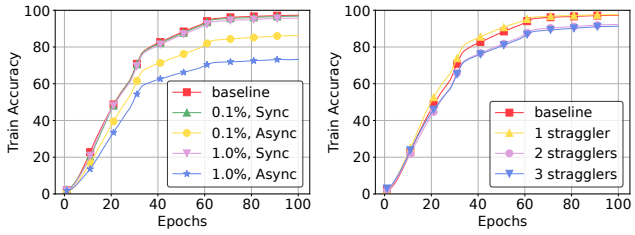


Figure 11: Resiliency to Gradient Losses. Displaying attainable accuracy with packet drops and stragglers.

Figure 10 shows that for BERT, the accuracy of THC actually improves as the number of workers increases, with the error decreasing from −0.1% to 0 (no difference from baseline) from 4 to 64 workers. Although there seems to be some outlier at 8 workers that is likely due to randomness in the training process, these trends match our predictions in Section 4 and can be attributed to the increased accuracy of the estimate of the average gradient. In comparison, the error of TopK quickly inflates by 9.9× from −0.047% to −0.46% in the same region because bias in the compression dominates and causes larger compression errors. The data for RoBERTa show a similar trend, with THC becoming the most accurate at 32 workers and beyond.

Such results are promising for THC because actual system performance depends on both throughput and compression accuracy. As we showed in §8.1, THC has a higher throughput in training than most compression algorithms. At 16 workers and beyond, THC also shows the highest accuracy, implying that THC will have better time-to-accuracy results for a system with many workers. The advantage over other compression algorithms is more apparent in larger scale systems because biased algorithms such as TopK lose accuracy at scale.

However, we cannot increase the number of workers to an arbitrary large size without incurring costs to accuracy. From each worker, the largest value per coordinate aggregated at the switch is equal to the granularity, so the maximum aggregated result is $m = g \times (\#\ of\ workers)$ and the number of bits needed to send this value downstream is $\lceil \log_2 m \rceil$. If we keep the number of bits sent downstream constant, we must decrease the granularity for a larger number of workers to prevent overflow, which increases the error. This can be seen from our results in Figure 15 in Appendix D. One advantage however is that as the granularity decreases, we can also decrease the bit budget for THC, sending fewer bits per coordinate upstream. On the other hand, if we keep the granularity constant, then we must send more bits per coordinate downstream, decreasing the throughput. In these experiments, we kept the granularity constant and instead adjusted the compression ratio of the downstream data (including for TopK and QSGD for fair comparison). It is likely that the optimal strategy is to employ a combination of the options depending on the specifics of the system.

Packet Loss. We simulate packet losses in both directions between workers and the PS. Figure 11 shows that even with

a packet loss rate of 1%, which greatly decreases the final training accuracy with no synchronization, our proposed synchronization scheme reduces the training accuracy drop from 24% to 1.5%. With 0.1% packet loss, synchronization reduces the accuracy discrepancy from 11% to 0.5%, which is nearly indistinguishable from baseline. The corresponding test accuracies are shown in Figure 16 of Appendix D and demonstrate similar results.

Stragglers. To mitigate stragglers, we model a scheme where the PS only waits for the top n% of the workers for aggregation. Figure 11 shows the effect of randomly choosing a 1/2/3 stragglers during each round and dropping their gradients. With 10 workers, this corresponds to waiting for 90%/80%/70% of the workers. Waiting for the top 90% reaches the baseline accuracy, whereas 80% and 70% show only a 5-6% decrease in final training accuracy.

9 Related Work and Discussions

Systems Support for Gradient Compression. Gradient compression systems have been developed to train large models that are increasingly bottlenecked by communication, given the slower growth of bandwidth compared to GPU capability. Previous systems (e.g., HiPress [6] and Espresso [73]) focus on compression awareness and finding compression strategies and work division (e.g., compression on GPU or CPU). These systems maximize the overlap of efficient communication and compression to hide existing overhead. THC, on the other hand, mitigates compression overhead by reducing the number of compress/decompress operations. THC is complementary with these works.

In-network Aggregation for ML. SwitchML [57] and ATP [33] have demonstrated the benefits of aggregating gradients within networks, but they do not support compression as the data is restricted to the format that can be directly aggregated. OmniReduce [17] and ASK [24] propose using a key-value data structure for in-network aggregation. However, these approaches require compressing the gradient data into the key-value format and make assumptions about the data itself. In THC, we support aggregation directly on compressed data, making it orthogonal to previous works. This leads to efficient in-network aggregation with compression.

Supporting Other AllReduces. An important future research direction is incorporating homomorphic compression in other types of all-reduce like ring-based or tree-based. Currently, compression schemes fail to improve the performance of these types [2]. For example, in the widely-deployed ring all-reduce that requires $O(n^2)$ aggregation operations, existing schemes would need an excessive number of decompression and re-compression operations, leading to poor accuracy and slowdown compared to an uncompressed baseline. THC makes the first step towards making compression algorithms ring-based or tree-based all-reduce friendly. For example, we may run the reduce operation directly on gradients compressed with Uniform THC using the same number of bits required for the PS aggregation (e.g., 8). However, this method is not compatible with our various optimizations, such as sending just b (e.g., 4) bits or using the lookup table, and is thus sub-optimal.

Colocated with Other Training Paradigms. While THC primarily focuses on data parallelism, it can seamlessly integrate with state-of-the-art training paradigms that use hybrid approaches combining tensor, pipeline, and data parallelism [43, 61]. Since all these optimizations are clearly separated in different dimensions, THC can be applied to the line of data parallelism without additional adaption. Moreover, Megatron-LM [43] reports that data parallelism remains the dominating factor in training throughput; thus, gradient exchanges will continue to make a significant contribution to computation and communication costs. We believe that increasing the degree of data parallelism is still a better choice, emphasizing THC's crucial role in training optimization.

Compatibility with Security. An extensively studied application of homomorphism is Homomorphic Encryption (HE) [1] in the security field. Although we consider HE as orthogonal to our work, it might be feasible to combine THC with other security practices. For example, applying differential privacy [16] techniques first and then compressing the tensors with THC can be practicable.

10 Conclusion

THC is a novel framework that formally defines homomorphic compression. As homomorphic compression supports direct aggregation of compressed data, it also allows an elegant combination of gradient compression and in-network aggregation. To demonstrate THC's generalizability, we build a distributed DNN training system prototype that employs both the THC algorithm and in-network aggregation to accelerate gradient synchronization. Testbed experiments with four GPU workers, one programmable switch, and 100Gbps network show that our system prototype achieves up to 1.47× TTA improvement when we enable both gradient compression and in-network aggregation.

11 Acknowledgment

We thank the NSDI reviewers and our shepherd, Qun Huang, for their invaluable feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Ran Ben Basat was supported by the Meta Network for AI faculty award. Michael Mitzenmacher was supported in part by NSF grants CCF-2101140, CNS-2107078, and DMS-2023528. We thank Vyas Sekar for proposing the term 'Homomorphic Compression'.

References

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), jul 2018.

[2] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 652–672, 2022.

[3] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 557–563, 2006.

[4] Dan Alistarh, Demjan Grubic, Jerry Z. Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1707–1718, Red Hook, NY, USA, 2017. Curran Associates Inc.

[5] InfiniBand Trade Association. InfiniBand Trade Association. RoCE v2 Specification. . https://cw.infinibandta.org/document/dl/7781, 2014.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Ran Ben Basat, Michael Mitzenmacher, and Shay Vargaftik. How to send a real number using a single bit (and some shared randomness). In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 25:1–25:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] Ran Ben Basat, Shay Vargaftik, Amit Portnoy, Gil Einziger, Yaniv Ben-Itzhak, and Michael Mitzenmacher. QUIC-FL: Quick Unbiased Compression for Federated Learning. *arXiv preprint arXiv:2205.13341*, 2022.

[9] Ran Ben-Basat, Yaniv Ben-Itzhak, Michael Mitzenmacher, and Shay Vargaftik. Optimal and Near-Optimal Adaptive Vector Quantization. *arXiv preprint arXiv:2402.03158*, 2024.

[10] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] ByteDance. BytePS Environment Variables. https://github.com/bytedance/byteps/blob/master/docs/env.md, 2021.

[13] Aaron Daniel Cohen, Adam Roberts, Alejandra Molina, Alena Butryna, Alicia Jin, Apoorv Kulshreshtha, Ben Hutchinson, Ben Zevenbergen, Blaise Hilary Aguera-Arcas, Chung ching Chang, Claire Cui, Cosmo Du, Daniel De Freitas Adiwardana, Dehao Chen, Dmitry (Dima) Lepikhin, Ed H. Chi, Erin Hoffman-John, Heng-Tze Cheng, Hongrae Lee, Igor Krivokon, James Qin, Jamie Hall, Joe Fenton, Johnny Soraker, Kathy Meier-Hellstern, Kristen Olson, Lora Mois Aroyo, Maarten Paul Bosma, Marc Joseph Pickett, Marcelo Amorim Menegali, Marian Croak, Mark Díaz, Matthew Lamm, Maxim Krikun, Meredith Ringel Morris, Noam Shazeer, Quoc V. Le, Rachel Bernstein, Ravi Rajakumar, Ray Kurzweil, Romal Thoppilan, Steven Zheng, Taylor Bos, Toju Duke, Tulsee Doshi, Vincent Y. Zhao, Vinodkumar Prabhakaran, Will Rusch, YaGuang Li, Yanping Huang, Yanqi Zhou, Yuanzhong Xu, and Zhifeng Chen. Lamda: Language models for dialog applications. In *arXiv*. 2022.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Ron Dorfman, Shay Vargaftik, Yaniv Ben-Itzhak, and Kfir Yehuda Levy. Docofl: Downlink compression for cross-device federated learning. 2023.

[16] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[17] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ACM SIGCOMM '21, page 676–691,

New York, NY, USA, 2021. Association for Computing Machinery.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[19] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

[20] Kaja Gruntkowska, Alexander Tyurin, and Peter Richtárik. EF21-P and Friends: Improved Theoretical Communication Complexity for Distributed Optimization with Bidirectional Compression. *arXiv preprint arXiv:2209.15218*, 2022.

[21] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev.*, 45(4):139–152, aug 2015.

[22] Wang Hao, Qin Yuxuan, Lao ChonLam, Le Yanfang, Wu Wenfei, and Chen Kai. Preemptive switch memory usage to accelerate training jobs with shared in-network aggregation. In *2023 IEEE 30th International Conference on Network Protocols (ICNP)*, pages 1–11, 2022.

[23] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[24] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 33–47, New York, NY, USA, 2023. Association for Computing Machinery.

[25] A Hedayat and Walter Dennis Wallis. Hadamard matrices and their applications. *The Annals of Statistics*, pages 1184–1238, 1978.

[26] Intel. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[29] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pages 3252–3261. PMLR, 2019.

[30] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ACM SIGCOMM '21, page 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Jakub Konečnỳ and Peter Richtárik. Randomized distributed mean estimation: Accuracy vs. communication. *Frontiers in Applied Mathematics and Statistics*, 4:62, 2018.

[32] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[33] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[34] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith,

Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[36] Xiaoyun Li, Belhal Karimi, and Ping Li. On distributed adaptive optimization with gradient compression. In *International Conference on Learning Representations*, 2021.

[37] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[39] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 486–498, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[41] Yurii Lyubarskii and Roman Vershynin. Uncertainty Principles and Vector Quantization. *IEEE Transactions on Information Theory*, 56(7):3491–3501, 2010.

[42] Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 449–460, New York, NY, USA, 2019. Association for Computing Machinery.

[43] 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. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[45] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). . https://docs.nvidia.com/networking/display/SHARPv200, 2020.

[46] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Lucian Petrica, Tobias Alonso, Mairin Kroes, Nicholas Fraser, Sorin Cotofana, and Michaela Blott. Memory-efficient dataflow inference for deep cnns on fpga. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 48–55, 2020.

[48] Constantin Philippenko and Aymeric Dieuleveut. Bidirectional compression in heterogeneous settings for distributed or federated learning with partial participation: tight convergence guarantees. *arXiv preprint arXiv:2006.14591*, 2020.

[49] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[51] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. Cassini: Network-aware job scheduling in machine learning clusters. *arXiv preprint arXiv:2308.00852*, 2023.

[52] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 17–23 Jul 2022.

[53] Ali Ramezani-Kebrya, Fartash Faghri, Ilya Markov, Vitalii Aksenov, Dan Alistarh, and Daniel M Roy. Nuqsgd: Provably communication-efficient data-parallel sgd via nonuniform quantization. *J. Mach. Learn. Res.*, 22:114–1, 2021.

[54] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[55] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[56] Mher Safaryan, Egor Shulgin, and Peter Richtárik. Uncertainty Principle for Communication Compression in Distributed and Federated Learning and the Search for an Optimal Compressor. *arXiv preprint arXiv:2002.08958*, 2020.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[58] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[59] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*, 2022.

[60] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[61] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[62] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[63] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[64] Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 4452–4463, Red Hook, NY, USA, 2018. Curran Associates Inc.

[65] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes, 2019.

[66] Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. Distributed Mean Estimation With Limited Communication. In *International Conference on Machine Learning*, pages 3329–3337. PMLR, 2017.

[67] Shay Vargaftik, Ran Ben Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben Itzhak, and Michael Mitzenmacher. Eden: Communication-efficient and robust distributed mean estimation for federated learning. In *International Conference on Machine Learning*, pages 21984–22014. PMLR, 2022.

[68] Shay Vargaftik, Ran Ben-Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben-Itzhak, and Michael Mitzenmacher. Drive: One-bit distributed mean estimation. *Advances in Neural Information Processing Systems*, 34:362–377, 2021.

[69] Hao Wang, Jingrong Chen, Xinchen Wan, Han Tian, Jiacheng Xia, Gaoxiong Zeng, Weiyan Wang, Kai Chen, Wei Bai, and Junchen Jiang. Domain-specific communication optimization for distributed dnn training. *arXiv preprint arXiv:2008.08445*, 2020.

[70] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016.

[71] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[72] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[73] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 867–882, New York, NY, USA, 2023. Association for Computing Machinery.

[74] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1508–1518, Red Hook, NY, USA, 2017. Curran Associates Inc.

[75] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960. USENIX Association, 2022.

[76] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[77] Xilinx. Vitis AI. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, 2023.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, ACM SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the power of inline Floating-Point operations on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.

[80] Xiang Zhou, Ryohei Urata, and Hong Liu. Beyond 1 tb/s intra-data center interconnect technology: Im-dd or coherent? *Journal of Lightwave Technology*, 38(2):475–484, 2020.

A Uniform THC Preliminaries

Given a bandwidth budget of b bits per coordinate (e.g., per gradient entry), we define a compression scheme using a pair of compression and decompression operators.

Definition 4 (compression operator) *A compression operator $C : \mathbb{R}^d \to \{0,1\}^{b\cdot d}$ takes a real-valued d-dimensional vector and outputs a $(b\cdot d)$-bits compressed representation.*

Definition 5 (decompression operator) *A decompression operator $D : \{0,1\}^{b\cdot d} \to \mathbb{R}^d$ takes a $(b\cdot d)$-bits compressed representation and outputs a real-valued d-dimensional estimate of the input vector.*

More generally, a compression scheme may require sending some additional information. We, therefore, allow $b\cdot d + O(1)$ bits in practice.

For a vector $x \in \mathbb{R}^d$, we denote its estimate by $\widehat{x} = D(C(x))$. The goal of a compression scheme is then, given a bandwidth budget b, to minimize some error metric, e.g., the expected squared error, $\mathbb{E}[\|x-\widehat{x}\|^2]$. Before describing THC, for ease of presentation, we present a simplified (uniform) version of the THC framework and later generalize it.

A.1 Uniform Homomorphic Compression

In distributed deep learning, at each training round, the mean of the workers' gradients forms the update of the model's parameters for the next round. Without compression, we could add all the workers' gradients and divide the results by the number of workers.

To reduce the bandwidth with a minimal impact on accuracy, the Distributed Mean Estimation (DME) problem has been extensively studied [4, 31, 41, 56, 66–68]. Namely, in DME, workers compress their gradients before sending them for aggregation. Most DME works only consider compression in this direction, while messages from the parameter server to workers remain uncompressed. To achieve bidirectional compression, several works further suggest that the server, after decompressing and aggregating the gradients, will recompress the result before sending it back (e.g., [20, 48]), introducing additional delay and error. Avoiding these problems motivates the following definition; for convenience, we henceforth denote $\langle i \rangle = \{0,\ldots,i-1\}$ for any $i \in \mathbb{N}$.

Definition 6 (Uniform Homomorphic Compression) *We say that a compression scheme (C,D) is* uniform homomor-

phic *if for any* $n, d \in \mathbb{N}$ *and* $x_0, x_1, \ldots, x_{n-1} \in \mathbb{R}^d$, *it satisfies*

$$\frac{1}{n} \cdot \sum_{i \in \langle n \rangle} D(C(x_i)) = D\left(\frac{1}{n} \cdot \sum_{i \in \langle n \rangle} C(x_i)\right).$$

That is, a Uniform Homomorphic Compression (UHC) scheme allows averaging the compressed representations and applying a single decompression invocation.

That is, by using UHC, the parameter server can sum up the compressed gradients and send $\sum_i C(x_i)$ back (in compressed form) without increasing the delay and error.

A.2 Uniform Stochastic Quantization

Two desired properties of gradient quantization schemes in a distributed setting are *unbiasedness* (i.e., $\mathbb{E}[\widehat{x}] = x$) and *independence* (i.e., each worker makes the random choice of their quantization values independently). These features are especially useful in distributed deep learning as the errors of the different workers then cancel out on average rather than add up, leading to a better estimation of the mean.

One of the most fundamental compression techniques that offer both properties is *uniform stochastic quantization* (USQ). Intuitively, given a vector, $x \in \mathbb{R}^d$, and denoting its minimum by m and maximum by M, using USQ to quantize each coordinate $x[j]$ to a single bit means rounding it to m with probability $(x[j]-m)/(M-m)$ and to M with probability $(M-x[j])/(M-m)$. That is, the sender encodes the coordinate $x[j]$ using one bit $C(x)[j] \in \{0,1\}$ and the receiver estimates it as $\widehat{x} = D(C(x)) = m + (M-m) \cdot C(x)$. Note that m and M need to be sent to the receiver as well. However, considering that d is large, this overhead is negligible.

This idea generalizes to any number of bits per coordinate b by partitioning the *range* $[m, M]$ into $2^b - 1$ *uniform* (i.e., equal-length) intervals where each entry is rounded to one of its nearest endpoints c_0, c_1 with probabilities $(x[j]-c_0)/(c_1-c_0)$ and $(c_1-x[j])/(c_1-c_0)$ to make the estimate unbiased. That is, when the message for coordinate $x[j]$, $C(x)[j] \in \langle 2^b \rangle$, is sent to the receiver that estimates the coordinate as $m + C(x)[j] \cdot (M-m)/(2^b-1)$.

Despite its popularity and simplicity, in our context, USQ has two main drawbacks. First, USQ is not homomorphic; this is because each worker has *its own* minimum and maximum values, and accordingly, the b-bit messages describing the same coordinate by different workers are not amenable to aggregation without decompression. Second, USQ's error highly depends on the input vector's distribution, e.g., the difference between the minimum and maximum. For example, for $b = 1$, if the input vector is $(1, -1, 0, 0, \ldots, 0)$, all the zero-valued coordinates will be rounded with an error of 1, and the vector's estimate will greatly differ from the input.

Several recent works propose pre-processing each gradient and post-processing the average's estimate. The insight is to change the vector's distribution prior to quantization to avoid bad cases. For example, [66] proposes to preprocess by applying the randomized Hadamard transform to ensure that the range is small with high probability and postprocess using the inverse transform. As another example, Kashin's representation [41, 56] allows projecting the vector into a higher-dimensional space with similar magnitude coefficients.

B Optimally Solving the Lookup Table Optimization Problem

As described in Section 5.2, the solution of the following optimization problem yields the optimal lookup table $T_{b,g,p} = T$. Observe that the problem depends on the parameters b, g, p, where b is the number of bits workers send for each quantized coordinate, g is the granularity (the range of values that the table can take), and p is the expected fraction of transformed and scaled coordinates that are not taken into account when determining the truncation range $[-t_p, t_p]$.

$$\underset{P,T}{\text{minimize}} \int_{-t_p}^{t_p} \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot (a - T(z))^2 \cdot \phi(a) \cdot da$$

subject to

$$\begin{aligned}
&(\textit{Unbiasedness}) && \sum_{z \in \langle 2^b \rangle} P(a,z) \cdot T(z) = a && \forall a \in [-t_p, t_p] \\
&(\textit{Probability}) && \sum_{z \in \langle 2^b \rangle} P(a,z) = 1 && \forall a \in [-t_p, t_p] \\
& && P(a,z) \geq 0 && \forall a \in [-t_p, t_p],\ z \in \langle 2^b \rangle \\
&(\textit{Granularity}) && T(z) \in \left\{ \tfrac{2t_p}{g} \cdot i - t_p \mid i \in \langle g+1 \rangle \right\} && \forall z \in \langle 2^b \rangle
\end{aligned}$$

Recall that, without loss of generality, we may assume that $0 = T(0) < T(1) < \ldots < T(2^b - 1) = g$, which significantly narrows down the search range from $(g+1)^{2^b}$ (which is the number of options to choose a table value for each table index). Namely, this observation means that the number of possible options for T is $SaB(g - 2^b - 1, 2^b - 1)$ (*SaB* stands for the *stars-and-bars*), where $SaB(n,k) = \binom{n+k-1}{k-1}$ is the number of options for throwing n identical balls into k distinct bins. This is because we can think of $2^b - 1$ bins representing the values $\{T(i+1) - T(i) \mid i \in \langle 2^b - 1 \rangle\}$; this way, 'throwing a ball' into the i'th bin corresponds to increasing the difference by 1, and we have $g - 2^b - 1$ balls as all bins must be non-empty to enforce the strict monotonicity. The number of options is therefore $SaB(g - 2^b - 1, 2^b - 1) = \binom{g-3}{2^b-2} \ll (g+1)^{2^b}$. For example, if $b = 4, g = 51$, we reduce the number of options from $52^{15} \approx 5.5 \cdot 10^{25}$ to $\binom{48}{14} \approx 4.8 \cdot 10^{11}$. We note that these b, g values are the largest ones that we have found to be of interest as they yield a solution whose accuracy is on par with an uncompressed baseline.

Algorithm 4 Stars-and-Bars (n,k) Enumeration

Initialize $B[0]=n, \quad B[1]=B[2]=\ldots,B[k-1]=0$
`Yield`(B) ▷ This is the 0'th option
for $j=1,2,\ldots,SaB(n,k)-1$ **do**
  $a=\min\{i \mid B[i]>0\}$
  $B[a+1]=B[a+1]+1$
  $S=B[a]-1$
  $B[a]=0$
  $B[0]=S$
  `Yield`(B) ▷ B is the j'th option
end for

To further reduce the number of options, if g is odd, we leverage the symmetry of the normal distribution (i.e., that $\phi(a)=\phi(-a)$ for any $a\in\mathbb{R}$). In particular, together with the fact that the number of table indices is even, this implies that a table index $T(z)$ exists in the optimal table if and only if there exists z' such that $T(z')=g-T(z)$. In particular, this can be manifested as the following additional constraint:

$$(\textit{symmetry}) \qquad T(z)=T(z+2^{b-1})-\tfrac{g+1}{2} \quad \forall z\in\langle 2^{b-1}\rangle$$

Notice that this further reduces the number of options to $SaB\left(\frac{g+1}{2}-2^{b-1}-1,2^{b-1}-1\right)$. Using the $b=4,g=51$ example, we reduced the number of options to just 100947.

This allows us to efficiently solve the problem optimally by computing the target integral for every possible value of T. For each value, we use the fact that stochastic quantization (picking one of the two closest quantization values with probabilities that make it unbiased) is optimal given the quantization values [7]. In particular, this means that we can easily derive the optimal P values and thus compute the integral. For example, if $b=2,g=4$ and $T(0)=0,T(1)=1,T(2)=3,T(3)=4$, we can compute the integral as:

$$\int_{-t_p}^{-t_p/2}\left(\frac{a-(-t_p)}{t_p/2}\cdot(a-(-t_p/2))^2+\frac{-t_p/2-a}{t_p/2}\cdot(a-(-t_p))^2\right)\cdot\phi(a)\cdot da$$
$$+\int_{-t_p/2}^{t_p/2}\left(\frac{a-(-t_p/2)}{t_p}\cdot(a-t_p/2)^2+\frac{t_p/2-a}{t_p}\cdot(a-(-t_p/2))^2\right)\cdot\phi(a)\cdot da$$
$$+\int_{t_p/2}^{t_p}\left(\frac{a-t_p/2}{t_p/2}\cdot(a-t_p)^2+\frac{t_p-a}{t_p/2}\cdot(a-t_p/2)^2\right)\cdot\phi(a)\cdot da$$

Enumerating over the options: The last ingredient in our solver is a simple method to iterate over all the stars-and-bars options. Namely, consider wanting to throw n balls into k bins and let $B[i]$ denote the number of balls in the i'th bin. Then, the enumeration process is given by Algorithm 4.

The resulting solver is quite efficient: we ran it once for each of over 4000 different (b,g,p) combinations and computed all the optimal tables within mere minutes.

C Additional Parameter Server Details

C.1 Pseudocode of Parameter Server (PS)

Pseudocode 1 THC PS processing logic

Input: Gradient packet *pkt* from worker. Workers generate *pkt*.round_num and insert the *pkt*.num_worker along with gradient data for each packet before sending.

1: **if** *pkt*.roundnum < expected_roundnum[*pkt*.agtr_idx] **then**
2: Notify straggler
3: **else**
4: **if** *pkt*.roundnum = expected_roundnum[*pkt*.agtr_idx] **then**
5: recv_count[*pkt*.agtr_idx] += 1
6: **else**
7: recv_count[*pkt*.agtr_idx] = 1
8: expected_roundnum[*pkt*.agtr_idx] = *pkt*.roundnum
9: **end if**
10: Table indices lookup
11: Aggregate table values
12: **if** recv_count[*pkt*.agtr_idx] = *pkt*.num_worker **then**
13: Multicast back aggregation result
14: **else**
15: Drop *pkt*
16: **end if**
17: **end if**

The PS progressing logic is demonstrated as shown in Pseudocode 1. When workers' compressed gradient packets arrive, the PS will first check whether the `pkt.roundnum` is less than the `expected_roundnum` it stores. If so, then this packet is carrying obsolete data, and the PS will discard this packet and notify the sender that it is likely straggling (Line 1-2). Otherwise, the PS regards it as a normal case and updates the corresponding `recv_count` counter. (Line 5-7) After the PS finishes the table lookup and aggregation process for each packet, it will check if the aggregation is complete by comparing its `recv_count` counter and the `pkt.num_worker` (Line 12). If the aggregation is complete, it will multicast the aggregated gradient packet back, or drop the packet (Line 13-16).

C.2 Switch Resources Usage

The programmable switch version of THC Parameter Server has 32 aggregation blocks. Each aggregation block has a copy of the lookup table and can aggregate 32 bits (i.e., four 8-bit table values) in one pass. Overall, the programmable switch PS consumes 39.9 Mb of SRAM and 35 ALUs. THC workers send packets of 1024 table indices, so each packet needs $\frac{1024}{(32\times4)}=8$ passes to have all 1024 elements aggregated. Therefore, we recirculate a packet twice through each of the four pipelines (eight passes in total) and consume up to two recirculation ports per pipeline. When we have N workers, the first $N-1$ packets are dropped after aggregation while the N-th packet is recirculated through all four pipelines once again to collect the aggregation results.

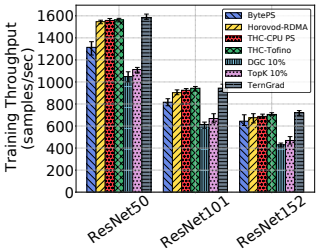


Figure 12: Throughput of training ResNet models on the local testbed.

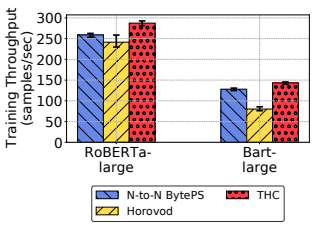


Figure 13: Throughput of training RoBERTa-large and Bart-large on AWS EC2.

D Evaluation Figures

D.1 Computational-intensive Model Training with THC

Figure 12 demonstrates the throughput results of training ResNet [23] models on our local testbed. Due to their computational-intensive nature, ResNet models don't experience much network bottleneck and hence don't benefit from accelerating inter-machine communication. Even with the most aggressive TernGrad compression, we are only able to improve the training throughput by up to 4.5% of that of Horovod-RDMA. So, computational-intensive models are poor candidates for gradient compression and should be trained with full-precision gradients unless the network bandwidth is low.

D.2 AWS EC2 Large Language Models Training Results

Figure 13 show the throughput results of training RoBERTa-large and Bart-large on AWS EC2. We achieve a 1.11× throughput improvement for RoBERTa-large and a 1.12× throughput improvement for Bart-large.

D.3 Optimizations of THC

To see how THC performs with the different optimizations mentioned in Section 4, we train both uniform and non-uniform THC with different optimizations. We use 4 workers on SST2 [63] with RoBERTa. To measure the performance, we enable all optimizations on THC, and then we run uniform THC (UTHC) with and without rotation and error feedback. We keep all optimizations for THC because the algorithm assumes rotation and error feedback to be enabled a prior, so it would not make sense to disable them for the test.

From Figure 14, we see that THC performs the best overall as expected, nearly reaching baseline accuracy. UTHC with and without error feedback seems to perform similarly, although error feedback seems to increase the performance slightly. The largest difference is disabling rotation, which drops the final accuracy by about 5%. This is expected, since removing rotation introduces a large bias into the algorithm, resulting in a large error.

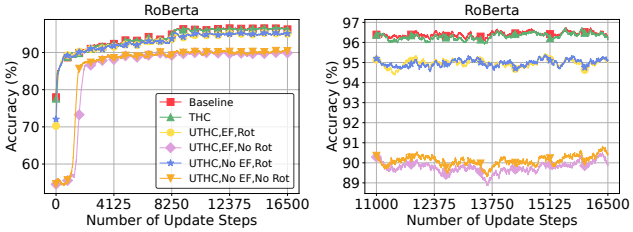


Figure 14: Accuracy of THC with Optimizations. The figure on the right is zoomed in on the last epoch, and accuracies are calculated on a sliding window of 336 batches.

D.4 NMSE for Different Granularities

We repeatedly compute the NMSE of the compression under different granularities. A gradient is first drawn from a lognormal distribution (which well approximate gradients in neural networks) and then copied multiple times to match the number of simulated workers. We run THC on the copies of the gradient and compute the NMSE. Repeating 100 times, we report the average NMSE as we increase the granularity. Figure 15 varies the granularity of THC, while maintaining 10 workers and used a p-fraction of 1/1024. Three curves are plotted with bit budget 2/3/4.

In Figure 15, we first note that the largest discrepancy in NMSE is between different bit budgets where the error decreases by almost an order of magnitude between bit budgets of 2 to 3 to 4. The bit budget corresponds directly to the compression ratio of the algorithm, so it is expected that the accuracy should increase as we use more bits for compression. Furthermore, the NMSE of THC also decreases as the granularity increases since larger granularity values allows for more fine-grained choices of quantization values, though this effect is more difficult to see.

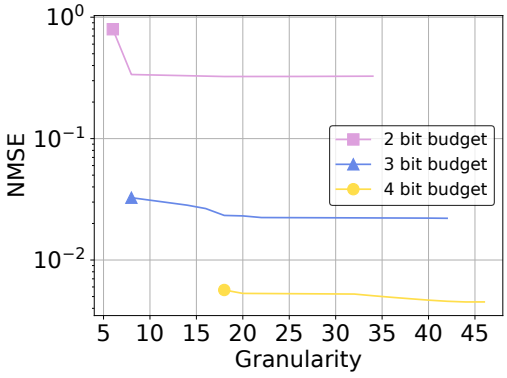


Figure 15: NMSE under different granularities and number of workers (plotted on log scale)

D.5 Test Accuracies for Resiliency Results

Similar to the training accuracy results, the test accuracy shows that (1) synchronization is beneficial when the system is lossy and (2) waiting for the top 90% of workers does not affect the final accuracy. Figure 16 shows that under 1%/0.1% loss, the discrepancy from baseline drops from 6%/3.2% to 1.5%/0.4%. For 80%/70% stragglers, the error from baseline is roughly 0.5%.

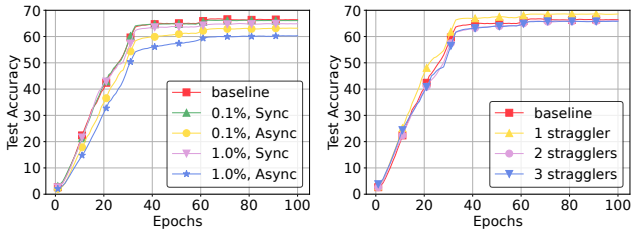


Figure 16: Test Accuracy with Gradient Losses