When ML Training Cuts Through Congestion: Just-in-Time Gradient Compression via Packet Trimming

Xiaoqi Chen*
Purdue University

Shay Vargaftik
VMware Research

Ran Ben Basat
UCL

Abstract

Distributed training of ML models generates significant network traffic when exchanging gradients and is sensitive to packet drops and retransmission caused by congestion when other traffic is sharing the network. While training processes can tolerate gradient compression to reduce traffic volume, it does not eliminate queue buildup in shallow-buffer switches, causing high latency and extended training time.

Our observation is that the emerging capability of handling temporal congestion through packet trimming provides the perfect opportunity to enable just-in-time gradient compression. Accordingly, we propose a novel approach where gradient packets are specifically constructed to be compressible via packet trimming, thus avoiding retransmission delays. Preliminary evaluations indicate that our trimmable encoding based on the Randomized Hadamard Transform has low computational overhead and that it achieves good training quality and shorter time to accuracy at high trim rates of up to 50%.

CCS Concepts

• **Networks** → **In-network processing**; **Data center networks**; *Data path algorithms*; Transport protocols; • **Computing methodologies** → **Artificial intelligence**; **Machine learning**; *Distributed computing methodologies*.

Keywords

Packet Trimming, Distributed Training, AllReduce, Gradient Compression, Collective Communications

ACM Reference Format:
Xiaoqi Chen*, Shay Vargaftik, and Ran Ben Basat. 2024. When ML Training Cuts Through Congestion: Just-in-Time Gradient Compression via Packet Trimming . In *The 23rd ACM Workshop on Hot Topics in Networks (HotNets '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3696348.3696880

* This work was completed while the author was a postdoctoral researcher at VMware Research Group.

HotNets '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696880

1 Introduction

Distributed ML training requires a significant amount of data transfer over the network: when running distributed stochastic gradient descent with data parallelism, after each GPU processes its own batch of input data, GPUs need to quickly exchange their local gradients to compute a global average, which is then used to update the model weights. While gradient communication is often predictable, and scheduling techniques can leverage that (e.g., [18]), in many settings, background traffic adds significant uncertainty (e.g., when the training does not use an isolated network). For example:

(1) For hyperscalers with hardware dedicated to distributed training, the rapidly growing scale of today's Large Language Model training (25,000 GPUs or more) already exceeded the traditional limit of densely connected clusters with dedicated network fabric (4,000-10,000 GPUs) [27, 29, 34]. A large-scale training job might thus span across multiple such clusters, connected by an over-subscribed second-layer network fabric [15].
(2) For many small and medium-sized businesses and academic institutions, due to limited power density in existing enterprise data centers, GPUs are often spread across different racks, and there might not be another dedicated network fabric for GPU traffic alongside the existing, shared network.
(3) When an ML trainer bids for the *cheapest* GPUs in a public cloud using spot instances, the underutilized (and therefore cheap) GPU nodes could be located anywhere, scattered across different racks in a data center far away from each other, or even across multiple data centers in a region. There might also be network jitter caused by virtualization infrastructure. Similar situation arises when a hyperscaler harvests [41] idle GPUs, which may be scattered across different racks of a dedicated training cluster due to fragmentation, to run low priority distributed training tasks using low-priority network traffic.

In all these cases, the network paths between GPU servers are longer and more unpredictable due to cross-traffic: paths may be shared by other (possibly higher priority) distributed training jobs and other applications, and collisions between different traffic flows lead to occasional congestion, high queuing delays, or even packet loss [4, 26].

Meanwhile, the congestion control logic in today's network stack is decoupled from how ML training frameworks send data. Gradients are sent as a black-box binary blob between network endpoints, using bulk send/receive calls. The network

rarely modifies the gradients; even with in-network aggregation [21, 23, 31], where gradients are added on switches, the servers or switches do not adjust the gradient compression level based on network congestion. The transport protocols for ML training, also referred to as the *collective communication library* or **ccl*, provide strict reliability semantics to the upper layer training process. They also require lossless delivery from the underlying network, either via a lossless fabric with Priority Flow Control (pause frames) or use retransmission to ensure data integrity [15, 19].

However, retransmission is costly: Reacting to packet loss by re-sending the same gradient at the same precision exacerbates the congestion, and more importantly, waiting for retransmissions creates slow-finishing *stragglers* among the many GPUs running a synchronous training batch. Other GPUs must all wait in an idle mode for the slowest GPU to complete sending its gradient unless and until a straggler mitigation strategy kicks in. Therefore, the tail latency, i.e., the slowest flow completion time, is especially important in achieving high-performance distributed ML training.

Our observation is that modern data center network supports packet trimming (e.g., for implementing NDP [17], EQDS [28], and Ultra Ethernet [35, 36]) in shallow-buffer switches to react to congestion. Namely, when a switch's queuing buffer fills up, instead of dropping incoming packets, the switch can "trim" the majority of bytes in a packet and preserve only a short header. The switch can then forward the header with high priority, bypassing other payload-carrying packets, such that the receiver and sender can both quickly react to the congestion at this network hop and reduce sending, similar to receiving Explicit Congestion Notification (ECN) tags while also scheduling timely retransmission. Compared to ECN, trimming allows switches to quickly mitigate queue overflow caused by incast traffic; this also enables senders to immediately ramp up new flows' sending rate without waiting for connection setup and congestion signals, ultimately achieving low end-to-end latency and flow completion time across the data center network.

Another extensively studied approach to improve the flow completion time in distributed ML training is gradient compression (e.g., [5–7, 12, 16, 20, 22, 24, 25, 33, 42]). By encoding gradients at a lower precision before sending them across the network, we can control the tradeoff between the number of bits sent and the resulting model accuracy. However, such methods need to decide on the compression ratio when encoding the data at the sender and cannot help in-flight packets react to the *unpredictable* congestion and packet losses. Also, directly coupling existing congestion control algorithms with compression ratios may lead to occasional over-compressing and sending too few bytes, not fully utilizing the available bandwidth for the best accuracy.

In this paper, we set to achieve the best of both worlds. Ideally, ML training should use more network bandwidth when the network is free while compressing the gradients whenever the network becomes congested. The challenge here is that it is hard to predict whether a packet will encounter congestion inside the network and whether reactions to a congestion signal will be delayed. Also, unnecessary compression may lead to an accuracy drop or prolonged training time. Therefore, the best place to compress gradients is within the congested network switch itself: shrinking the gradient packets only when the queue is congested, similar to packet trimming. This allows all gradient transmission to finish on time without retransmission and immediately ramp up to use all the available network bandwidth while compressing only when necessary.

Yet, implementing adaptive just-in-time gradient compression algorithms inside high-speed network switches is a challenging algorithmic and engineering feat. Running gradient compression at line rate may require building many arithmetic calculation circuits into the switch's chipset. Meanwhile, packet trimming is already supported by many existing data center switches using different chipsets, including Intel Tofino, Broadcom Trident 4, and NVIDIA Spectrum 2 [3].

We, therefore, propose the *trimmable gradients* design, illustrated in Figure 1, to allow widescale deployment of in-network gradient compression. By rethinking how GPUs prepare and send gradients, the network devices along the path can compress the gradients whenever congestion is forming in its queues. This way, the bandwidth-heavy distributed ML training jobs can coexist with other bursty traffic in a shared network while achieving consistent flow completion time with no stragglers. The key idea of trimmable gradients is to fit compressed gradients at the front of the packet and the remainder terms in the back of the packet, such that switches can activate compression simply by packet trimming.

Trimmable gradient achieves the right split of labor: the compute-efficient GPUs are responsible for pre-computing gradient transform and quantization, while the network switches can rapidly decide and activate gradient compression upon congestion, with minimal computational required.

The rest of this paper is arranged as follows. In Section 2, we present how trimmable gradient packets are formatted, trimmed, and received. In Section 3, we discuss our trimmable compression algorithm for gradients based on stochastic quantization, subtractive dithering, and Randomized Hadamard Transform (RHT). In Section 4, we present a prototype evaluation showing our trimmable gradient compression scheme incurs low computation overhead, and RHT encoding leads to faster time-to-accuracy when encountering heavy congestion trimming despite the longer time spent in encoding. In Section 5, we discuss some interesting future research questions related to the wider adoption of trimmable gradients.

2 Arranging Gradients in Trimmable Packets

Next, we investigate how senders should lay out gradients in packets to best prepare for trimming. Naively, gradient coordinates are expressed as floating point numbers and packed one by one following the packet header, as illustrated in Figure 2(a). In this case, packet trimming leads to keeping several whole floating point numbers while discarding the rest.

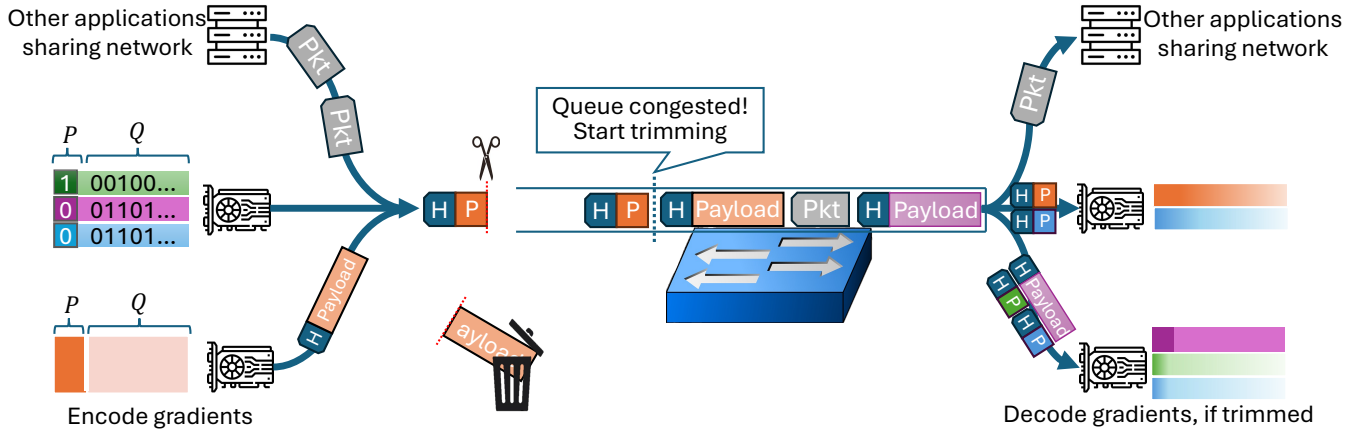

Figure 1: Our framework. The senders format the packets such that if the switch decides to trim a packet due to congestion, the receiver gets the compressed form of the gradients and can still aggregate them without needing retransmissions.

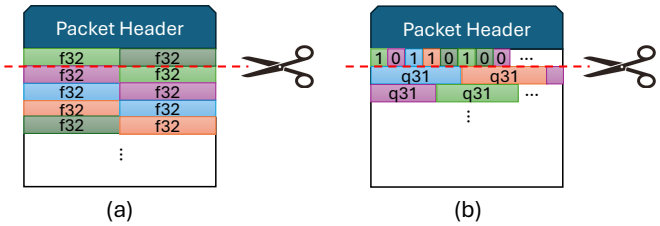

Figure 2: Instead of (a) keeping and discarding whole floating point numbers upon packet trimming, we can (b) rearrange the packet payload such that the first P=1 bits for each gradient coordinate are kept upon trimming.

As shown by MLT [40], distributed neural network training can often tolerate a certain fraction of lost gradient, e.g., up to 2.5%, without affecting training time and accuracy. Some gradient coordinates are more important: if we sort all coordinates by their magnitude, the training can typically tolerate losing as much as 20% (the smallest 20% coordinates), but only 0.4% of the largest 20% coordinates. Unlike MLT, which exploits priority queueing to encourage switches to drop the smaller coordinates, here we need to arrange the coordinates in each packet such that trimming will cause the smaller coordinates to be discarded. Thus, in the packet layout, the coordinates with larger magnitudes are closer to the header and vice versa: we can spread the largest-magnitude coordinates across the beginning of multiple packets while spreading the smaller coordinates across the tail of packets. Still, reducing the packet size by at most 20% provides a limited effect in alleviating queue congestion, and a real-world network may require much more aggressive trimming.

Instead, we split each gradient coordinate into two parts: A P-bit *head* and a Q-bit *tail*, and arrange all the heads in the front of the packet before all the tails. That is, if we pack n coordinates in a packet, the first $P \cdot n$ payload bits contain the compressed coordinates while the remainder is the information needed to recover the coordinates' original precision. The switch is then allowed to trim the packet to include only the first $P \cdot n$ bits of payload, effectively enabling in-network compression. This way, upon congestion, the switch can easily trim its size by approximately $\frac{Q}{P+Q}$.

As an example, we can keep $P = 1$ bit for every full-precision (32-bit) floating point. A typical MTU-sized packet of 1500 bytes can accommodate about $n = 365$ coordinates. With $P = 1$, the trimmed packet contains 45 bytes of compressed payload. Accounting for a 42-byte standard header (Ethernet, IP, UDP), we should configure the switches to trim packets at 87 bytes upon congestion, achieving a compression ratio of 94.2%. This should be sufficient for a switch to manage congestion and avoid packet loss.

Our proposal of in-network just-in-time compression to combat the *unpredictable* congestion and packet loss is orthogonal to many traditional gradient compression bandwidth decisions made *ahead of time* to reduce the amount of traffic sent in the first place. That is, if a sender knows about the extent of congestion at the time of transmission based on a coarser-grained congestion control feedback loop, it may adjust Q to change its bandwidth demand.[1] Subsequently, if a network switch nevertheless runs out of queueing buffer due to unpredicted congestion, it still trims the packet to keep $P < Q$ bits per coordinate. In the next section, we discuss how to efficiently encode gradients into $P = 1$ bit heads.

3 Trimmable Gradient Compression

Although there is a rich literature discussing how to quantize neural network weights and gradients (e.g., [5, 6, 8, 20, 22, 23, 33, 42]), they mostly discuss fitting into a pre-defined bit budget, e.g., communicating $1-4$ bits per coordinate. In this work, we have a unique need to construct a two-part encoding with $P+Q$ bits: the first P-bits "head" should be an efficient standalone compression when the tail bits are trimmed, while the remaining "tail" of Q bits should not carry redundant information that is already included in the head.

More generally, the trimmable quantization problem requires efficiently encoding the gradient into two or more parts of predetermined length, such that a decoder can decode using any number of parts forming a prefix of the encoding. In this paper, we focus on the two-part case, although our

[1] We note that in this case, even for non-trimmed packets, the receiver cannot obtain the original coordinate's precision.

solution can be expanded to more parts. We now discuss how to produce a two-part trimmable gradient packet.

3.1 Scalar Quantization

We first introduce a simple scheme for producing trimmable gradient packets: applying quantization independently to each gradient coordinate, a method known as scalar quantization.

For every gradient coordinate value v, we need to produce a P-bit quantization value $h(v)$ as the head. In this work, we focus on the $P = 1$ bit case and discuss future extensions in Section 5. We explain three natural approaches.

Sign-magnitude Quantization: The most straightforward 1-bit encoding we could use is the sign bit, i.e., $h(v) = sign(v)$, a natural choice assuming the gradients are symmetrically distributed around zero (we later remove this assumption (Section 3.2). The remaining $Q = 31$ bits are the mantissa and exponent of the original floating point value v, i.e., the floating point number without its sign bit.

If trimming occurs, the receiver gets a mix of full precision coordinates and quantized heads, i.e., sign bits. To help the receiver scale these sign bits back to match the magnitude of the original gradient, the sender also sends the standard deviation of the original gradient σ separately in a small packet that will not be trimmed. Using σ, the receiver then decodes the sign bits $\{-1, +1\}$ into $\{-\sigma, \sigma\}$.

However, as we later demonstrate, this simple method severely affects training convergence, even with only 2% of packets being trimmed. We also implement two alternative algorithms for calculating the 1-bit encoding, adapted from prior works in gradient compression.

Stochastic Quantization (SQ): for a parameter $L \in \mathbb{R}^+$, after clipping gradient value to range $v \in [-L, L]$, we encode v to +1 with probability $p_+ = \frac{L+v}{2L}$, and to -1 with probability $p_- = \frac{L-v}{2L}$. During decoding, $\{-1, +1\}$ are decoded into $\{-L, L\}$. Compared to sending the sign-magnitude quantization, Stochastic Quantization produces an *unbiased* encoding for the unclipped coordinates (i.e., with values in $[-L, L]$), meaning the expectation of the decoded values is equal to the original. Similar to TernGrad [43], we set $L = 2.5\sigma$, which is also sent separately and reliably via a small packet.

Subtractive Dithering (SD): When considering worst-case error, SQ can be improved by SD. For each coordinate x, let $\epsilon_x \in \mathbb{R} \sim \mathcal{U}(-L/2, L/2)$ be its 'dither' where $\mathcal{U}$ is the uniform distribution. Then, the quantization process employs, $Q(x) = L \cdot sign(x + \epsilon_x)$, and the de-quantization process employs, $\widetilde{x} = Q(x) - \epsilon_x$. The assumption here is that both the sender and receiver can compute the same ϵ_x by using shared randomness without extra communication; in our implementation, we use a pseudo-random generator on both sides sharing the same seeds. Intuitively, the added dither helps to decorrelate the quantization error with respect to the input, spreading the error across the range and making it less perceptible. SD improves the variance of the quantization in the sense that it makes it smaller compared to SQ in the worst case and independent of the input.

3.2 Exploiting Random Rotations

For those situations when the percentage of trimmed packets is significant, we propose an improved encoding based on random rotations, a technique that many state-of-the-art gradient compression methods use [9, 10, 30, 33], such that different gradient coordinates can share the impact of trimming while each suffers from only a small inaccuracy.

In particular, to improve the decoding accuracy for 1-bit trimmed gradients, we adapt the principles from DRIVE [38] a state of the art 1-bit gradient compression algorithm.[2] DRIVE applies the Randomized Hadamard Transform (RHT) to the gradients before quantization, as RHT can be implemented as a fast, in-place transform on GPUs. Intuitively, after RHT, the coordinates are symmetrically centered close to zero, which results in both a smaller quantization error and a smaller worst-case error in any single coordinate.[3]

Based on our measurement, applying RHT to an entire collective communication message gradient blob (e.g., default size 25MB, in Pytorch DDP) incurs a noticeable slowdown; therefore, we further optimize the RHT step by splitting up each collective communication message blob into smaller rows of, e.g., $2^{15} = 32,768$ entries, such that each row can fit within the GPU's L1 shared memory, and independently perform RHT to each row in parallel. This not only saves GPU computation but also reduces the communication latency.

Encoding: For each row $V = \{v_0, v_1, \ldots, v_{2^{15}-1}\}$, we first perform RHT using a pseudo-random seed s to obtain the rotated axis coordinates: $R_s(V) = \{r_0, r_1, \ldots, r_{2^{15}-1}\}$. We also calculate a scaling factor $f = \frac{\|V\|_2^2}{\|R_s(V)\|_1}$ which helps us decode in an unbiased fashion. As with the standard deviation discussed earlier, the row scaling factors are sent in small, reliable packets to avoid getting trimmed.

We note that in the $P = 1$ case, sending the sign bit $sign(r)$ as the head is beneficial since each rotated coordinate r follows a symmetric normal distribution with zero mean [38]. Therefore, for r represented as a 32-bit floating point, we once again rearrange each packet to contain the sign bits before including all the remaining 31-bit tails (mantissa and exponent bits). This means for the non-trimming case we achieved precise encoding of the original 32-bit number, without using any additional space overhead.

Decoding: The receiver decodes packets in groups based on each row that performed RHT. If none of the packets were trimmed, the receiver simply performs Inverse Randomized Hadamard Transform (IRHT) on the received rotated row to obtain the original row: $V = IRHT(r_0, r_1, \ldots r_{2^{15}-1})$. For a trimmed packet, it only contains the sign bits of some coordinates: $sign(r_i) \in \{-1, +1\}$; we scale these sign bits using the unbiased scale f. We can then decode an estimate of the original row: $\tilde{V} = IRHT(\widehat{r_0}, \widehat{r_1}, \ldots, \widehat{r_{2^{15}-1}})$, where $\widehat{r_i} = r_i$ if r_i is not trimmed or $\widehat{r_i} = f \cdot sign(r_i)$ otherwise.

[2] DRIVE was extended to any number of bits by EDEN [37]. For simplicity, we explain the 1-bit version.

[3] The system benefits of RHT were also recently demonstrated by THC [23].

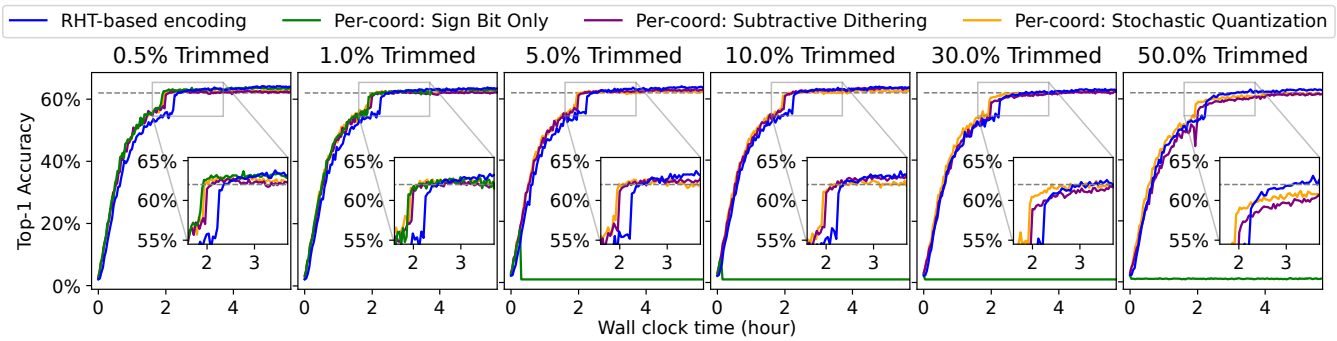

Figure 3: Time To Accuracy (TTA) for different trimming rates. The sign-bit-only method diverges for trimming rates higher than 1%. Also, the benefits of the more accurate RHT become evident for higher timing rates, especially 50%, where it is the only one to reach baseline accuracy.

4 Implementation and Evaluation

In this section, we present our Pytorch-based prototype and preliminary evaluation results indicating that trimmable gradients achieve low overhead and high resulting model accuracy under different levels of simulated congestion.

We prototype the SD, SQ, and RHT-based trimmable gradient encoder and decoder using customized communication hooks in the Pytorch Distributed Data-Parallel framework (version 2.3.0+cu121) to modify the gradient aggregation communication step.

We use the `fast-hadamard-transform` Python library (version 1.0.4) [11] for RHT implementation in CUDA. We set `torch.cuda.manual_seed` with a combination of training epoch number and collective communication message ID to create a shared pseudo-random number generator across different GPU servers. Our prototype implementation uses about 570 lines of Python code.

4.1 Testbed and Benchmark Setup

Our testbed consists of two server nodes, each hosting an Nvidia BlueField-2 SmartNIC (ConnectX-6) and an Nvidia A16 GPU, with CUDA 12.5 and Nvidia driver 5.5.0 installed. The two servers are connected via 100Gbps Ethernet Direct Attach Copper cables. Due to NCCL's proprietary nature, we cannot easily change the wire format; instead, for this preliminary evaluation, we simulate the effect of congestion using pre-set random probabilistic dropping/trimming, both in the software layer and on our SmartNIC. When a gradient coordinate is assigned as trimmed, we manually wipe its data entry and replace it with its quantized encoding, according to the decoding procedure discussed in Section 3.

For our training quality benchmark, we train the VGG-19 network from scratch on the CIFAR-100 visual recognition dataset, using a standard training setup.[4] In all settings, we train the network for 150 epochs, which is sufficient for it to converge, and measure its final top-1 and top-5 accuracy as well as its time to convergence.

[4]SGD with momentum 0.9, initial learning rate 10^{-3} with StepLR scheduler, cross-entropy loss, with batch size 64 and data augmentation.

The key of our experiment is to hold all hyper-parameters constant while only changing how gradients are aggregated between different servers. We use this benchmark to demonstrate the effect of trimmable gradient encoding and gather insights for eventual deployment in large-scale training.

4.2 Time To Accuracy

Depending on the different amounts of cross traffic and their priority, distributed training could experience vastly different fractions of packets dropped/trimmed. Here, we benchmark different drop/trim packet percentages ranging from 0.1% to 50%. A low percentage corresponds to normal priority traffic sharing a well-managed network with minimal incast or oversubscription, while the higher percentages correspond to scenarios when distributed ML training traffic was sent at low priority, sharing bottlenecks with much other traffic while trying to salvage unused bandwidth on a best-effort basis.

In Figure 3, we measure the top-1 accuracy, as a function of wall clock time, of the trained network with gradients sent using different encoding schemes. As we can see, for all trim rates, the RHT method learns slower in wall clock time due to more computational overhead per epoch; however, at a high trimming rate, it reaches higher accuracy faster. The sign-magnitude quantization is a quick solution for when the trimming rate is low, but its training diverges when the rate is 2% or higher. We note that a promising direction is, therefore, to dynamically choose the quantization method based on the anticipated congestion/trim rates.

4.3 Effect of Trimming on Training Time

We next compare the approaches by the time they require to reach the accuracy of the uncompressed baseline. As shown in Figure 4, when there is little to no congestion, and thus the trim rate is less than $\approx 0.5\%$, all solutions are slower than the baseline, consistently with Figure 5. For medium rates of trimming (0.5%-20%), the computationally lightweight methods of subtractive dithering and stochastic quantization offer faster training than the RHT-based one. Finally, when encountering a high level of congestion that causes substantial trimming, the improved decoding accuracy of the RHT-based

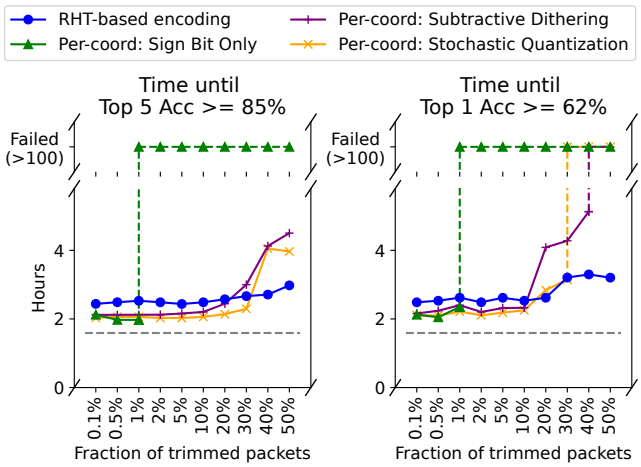

Figure 4: Time-To-Baseline-Accuracy for different trimming rates. The horizontal gray line is the NCCL baseline with no congestion. The benefit of RHT becomes evident for higher trimming rates, as expected.

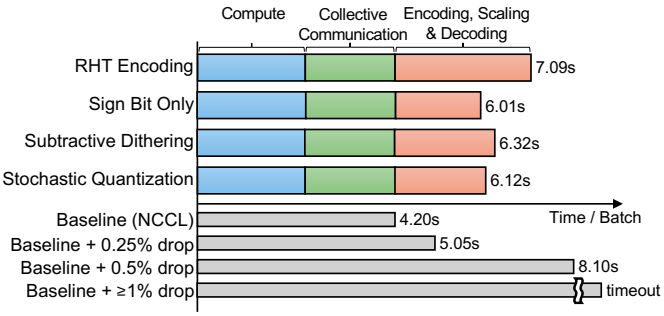

Figure 5: Time breakdown for each training round. RHT has high computation overhead, which is compensated by fewer training rounds needed to reach target accuracy.

compression comes in handy, and it is the fastest to reach the target accuracy and the only one that can cope with large trim rates (e.g., 50% for top-1).

4.4 Encoding Overhead

We next measure the computational overhead involved in DDP collective communication steps following the additional trimmable gradient encoding. As shown in Figure 5, using trimmable encoding in our prototype implementation adds 42%-68% of time for one distributed training round. We believe a significant fraction of this is due to the overhead in the hooking callback interface and can be further optimized. The RHT encoding is about 18% slower compared to the simpler per-coordinate scalar quantization methods, corresponding to the computational overhead of the rotations. We also note the baseline setup (unmodified NCCL) cannot tolerate much congestion: it can only tolerate 0.15%-0.25% packet drops (retransmissions) without disproportional slowdown, and with only 1%-2% drops, the training round becomes 5x-10x slower or start reporting timeout errors.

The encoding and scaling computation can be potentially fused with the neural network's backward pass computation, further reducing the computational overhead. That is, it may be possible to employ kernel fusion (e.g., [13]) such that the gradient computation during the backward pass already outputs the gradient in a format that supports trimmable encoding, requiring no further overhead.

5 Discussion and Future Work

This paper presents the initial benefit of trimmable gradients. Here, we discuss promising directions for future investigation.

5.1 Multi-Level Trimming

In this paper, we evaluated two-tier encoding with $P = 1$ bit as the trimming level. More generally, a switch can have multiple trimming actions to account for different congestion levels. For example, a switch can be configured to trim packets to either 25% size or 3% size (corresponding to trimming 32-bit floating point gradients into either 1 or 8 bits) or a mixture of both. Thus, we need to design versatile trimmable encodings that support different bit lengths and trimming levels.

Such a capability introduces two exciting challenges. First, one needs to design an accurate encoding that would allow the switch to choose the trimming level. Second, this opens the door to switch algorithms that decide which packets to trim and by how much. For example, is it better to have more packets trimmed to 50%, or have fewer packets trimmed, but to 3%? In a closed-loop setting, when a switch handles bursty incoming traffic, the two different trimming sizes lead to different congestion control behaviors and, therefore, different fractions of packets trimmed. We plan to run full-scale simulations or testbed studies to fully understand the confounding effect between queuing and congestion, trimming size, and the resulting fraction of trimmed packets. These results can help us solve the nontrivial optimization problem of achieving the shortest time-to-accuracy in the presence of congestion.

5.2 Other Gradient Compression Schemes

While quantization is a prominent approach to gradient compression, other works follow other techniques such as sparsification (e.g., [12, 40]) or low-rank decomposition (e.g., [39, 44]). It is an interesting future work to design trimming schemes for these approaches and shed light on what is the best method or a combination of methods.

In sparsification methods, workers decide on a subset of the gradient coordinates to communicate in a way that minimizes the error for a given sparsification ratio. For example, In MLT [40], the importance of gradient elements are decided by their magnitude; the authors observe they can discard as much as 20% of the smallest gradients without noticeable impact on training accuracy.

In Low-rank decomposition methods, gradients of parameter matrices are decomposed into low-rank representations

and, given a different communication bandwidth budget, different compression levels can be constructed by choosing the ranks with the largest eigenvalue.

These methods require presetting the compression ratio ahead of time, and it is an interesting research challenge to adapt them into or combine them with trimmable encoding for just-in-time compression, as we discuss next.

5.3 Interacting with Congestion Control

Congestion feedback signals allow senders to sense bandwidth over-subscription in their bottleneck link and adjust their sending rates accordingly. For a distributed training setup, we could adjust Q in our trimmable encoding, or apply other gradient compression methods discussed in Section 5.2, to change the size of data sent based on expected congestion in the network and the desired accuracy.

However, the switches in the network may still suffer from *unpredictable* congestion caused by new flows ramping up or new incast starting. In this case, an additional trimming-based just-in-time compression must be applied separately, even if gradients are already compressed ahead of time. Thus, we seek a gradient encoding design that supports both ahead-of-time compression and just-in-time trimming of any fraction of packets. This is an exciting direction for future work.

For example, if we use gradient sparsification, the sender can first discard a certain ratio of gradient coordinates according to the congestion control signal and subsequently send them using RHT-based trimmable encoding. If we use low-rank decomposition, we need to find a certain encoding format for laying out different ranks in the packet payload, such that trimming arbitrary packets always affects only the ranks with the least importance (smallest eigenvalue).

Also, we note conventional congestion control algorithms are designed to avoid queue buildup using more conservative sending rates. Performing ahead-of-time compression based on this logic will lead to slight over-compression and under-sending (unused link capacity). Instead, we want to adjust the congestion control logic to always slightly under-compress and over-send so that the gradient traffic always saturates the link. We can let the switch decide how much additional trimming-based just-in-time compression needs to be applied.

5.4 Reproducibility

With trimmable gradient encoding, every distributed training run becomes unique due to the unpredictable nature of network congestion and how it affects which packets are trimmed and to what extent. However, ML practitioners are increasingly advocating for the reproducibility of training runs for both debugging and transparency purposes.

To assist in reproducing the training results with packet trimming, the distributed training framework can record the indices of packets that were trimmed across the entire training episode. This allows the replaying of this trimming "transcript" in a future training run. During a replay, the distributed training framework should send all the collective communication messages in the slower, reliable channel (with trimming disabled and retransmission enabled), while simulating the congestion and trimming effect on the receiver end. This, together with GPU-side reproducibility techniques (e.g., see [2]), allows the user to reproduce a previous run with high fidelity.

5.5 Fully Sharded Data Parallel

To train even larger models that can no longer fit into a single GPU server's VRAM, we must use Fully Sharded Data Parallel (FSDP) [1, 32], where a single copy of model weights is stored across multiple servers. Besides gradient all-reduce, FSDP frequently performs *gathering*: Before calculating a matrix multiplication, a GPU must gather the weights of a matrix stored on other GPUs through scatter-gather collective communication. Thus, FSDP performance is heavily bounded by communication bandwidth between GPU servers.

We believe a small fraction of imperfection in copied weights has limited impact on training quality, due to the redundant nature of large neural networks (e.g., [14]). Thus, trimmable packets are also useful in FSDP training across shared network fabric with unpredictable congestion. By avoiding retransmission for both scatter-gather and all-reduce steps, we can avoid stragglers and maintain tighter synchronization between GPUs, which has the potential to reduce overall training time-to-accuracy for FSDP. This is an exciting future direction to explore and experiment.

5.6 Trimmable Inference

In this paper, we investigated the benefits of just-in-time gradient compression in accelerating distributed ML training. An intriguing research question is whether just-in-time compression can be useful for ML inference use cases as well.

6 Conclusion

In this paper, we introduced the trimmable gradients method, a novel approach enabling switches to perform just-in-time gradient compression via packet trimming. Trimmable gradients allow distributed ML training workloads to better utilize shared network fabric while tolerating unpredictable congestion and avoiding costly retransmissions. We designed 1-bit trimmable gradient encodings that cost no additional space and minimum computational overhead, using Stochastic Quantization, Subtractive Dithering, and Randomized Hadamard Transform. A preliminary evaluation showed Stochastic Quantization allows distributed training to tolerate up to 10%-20% trimmed packets without a noticeable increase in time-to-accuracy, and at higher trim rates, RHT-based encoding achieved the shortest time to accuracy, despite its higher computational overhead and per-round time. Our paper opens up exciting future work on tighter integration of the collective communication layer in distributed ML training and trimming-enabled data center network transport.

Acknowledgments: We thank the anonymous reviewers and Sujata Banerjee for their insights and suggestions. Ran Ben Basat was partially funded by a gift funding from Huawei.

References

[1] [n. d.]. Huggingface - Fully Sharded Data Parallel. https://huggingface.co/docs/transformers/en/fsdp. Accessed: 2024-10-21.

[2] [n. d.]. Pytorch Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html. Accessed: 2024-10-21.

[3] Popa Adrian, Dumitrescu Dragos, Handley Mark, Nikolaidis Georgios, Lee Jeongkeun, and Raiciu Costin. 2022. Implementing packet trimming support in hardware. *arXiv preprint arXiv:2207.04967* (2022).

[4] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2024. Harmony: A Congestion-free Datacenter Architecture. In *NSDI*. 329–343.

[5] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2022. On the utility of gradient compression in distributed training systems. In *MLSys*, Vol. 4. 652–672.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *SOSP*. 359–375.

[7] Ran Ben Basat, Yaniv Ben-Itzhak, Michael Mitzenmacher, and Shay Vargaftik. 2024. Optimal and Approximate Adaptive Stochastic Quantization. In *NeurIPS*.

[8] Ran Ben Basat, Michael Mitzenmacher, and Shay Vargaftik. 2021. How to Send a Real Number Using a Single Bit (And Some Shared Randomness). In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*.

[9] Ran Ben Basat, Amit Portnoy, Gil Einziger, Yaniv Ben-Itzhak, and Michael Mitzenmacher. 2024. Accelerating Federated Learning with Quick Distributed Mean Estimation. In *ICML*.

[10] Sebastian Caldas, Jakub Konečný, H Brendan McMahan, and Ameet Talwalkar. 2018. Expanding the Reach of Federated Learning by Reducing Client Resource Requirements. *arXiv preprint arXiv:1812.07210* (2018).

[11] Tri Dao. 2024. Fast Hadamard Transform in CUDA, with a PyTorch interface. https://pypi.org/project/fast-hadamard-transform/.

[12] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *SIGCOMM*. 676–691.

[13] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[14] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *ICLR*.

[15] Adi Gangidi. 2023. Scaling RoCE Networks for AI Training. https://atscaleconference.com/videos/scaling-roce-networks-for-ai-training/. https://atscaleconference.com/videos/scaling-roce-networks-for-ai-training/ In Networking @ Scale 2023 Conference.

[16] Wenchen Han, Shay Vargaftik, Michael Mitzenmacher, Brad Karp, and Ran Ben Basat. 2024. Beyond Throughput and Compression Ratios: Towards High End-to-end Utility of Gradient Compression. In *HotNets*.

[17] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*. 29–42.

[18] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. Tictac: Accelerating distributed deep learning with communication scheduling. In *MLSys*, Vol. 1. 418–430.

[19] Ziheng Jiang et al. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *NSDI*. 745–760.

[20] Jakub Konecnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* 8 (2016).

[21] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *NSDI*. 741–761.

[22] Haoyu Li, Yuchen Xu, Jiayi Chen, Rohit Dwivedula, Wenfei Wu, Keqiang He, Aditya Akella, and Daehyeok Kim. 2024. Accelerating Distributed Deep Learning using Lossless Homomorphic Compression. *arXiv preprint arXiv:2402.07529* (2024).

[23] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. 2024. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *NSDI*. 1191–1211.

[24] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *ICLR*.

[25] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An efficient statistical-based gradient compression technique for distributed training systems. In *MLSys*, Vol. 3. 297–322.

[26] Jeffrey C Mogul and Lucian Popa. 2012. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review* 42, 5 (2012), 44–48.

[27] Timothy Prickett Morgan. 2023. Inside the Infrastructure that Microsoft Builds to Run AI. https://www.nextplatform.com/2023/03/21/inside-the-infrastructure-that-microsoft-builds-to-run-ai/. Accessed: 2024-06-25.

[28] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *NSDI*. 761–777.

[29] Mark Russinovich. 2023. Inside Microsoft AI Innovation with Mark Russinovich. https://build.microsoft.com/en-US/sessions/984ca69a-ffca-4729-bf72-72ea0cd8a5db. In Microsoft Build 2023 Conference.

[30] Mher Safaryan, Egor Shulgin, and Peter Richtárik. 2020. Uncertainty principle for communication compression in distributed and federated learning and the search for an optimal compressor. *Information and Inference: A Journal of the IMA* (2020).

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with in-network aggregation. In *NSDI*. 785–808.

[32] Hamid Shojanazeri, Yanli Zhao, and Shen Li. [n. d.]. Getting Started with Fully Sharded Data Parallel (FSDP). https://pytorch.org/tutorials/intermediate/FSDP_tutorial.html. Accessed: 2024-10-21.

[33] Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. 2017. Distributed mean estimation with limited communication. In *ICML*. 3329–3337.

[34] P.K Tseng. 2023. TrendForce Says with Cloud Companies Initiating AI Arms Race, GPU Demand from ChatGPT Could Reach 30,000 Chips as It Readies for Commercialization. https://www.trendforce.com/presscenter/news/20230301-11584.html. Accessed: 2024-06-25.

[35] Ultra Ethernet Consortium. 2024. UEC Progresses Towards v1.0 Set of Specifications. https://ultraethernet.org/uec-progresses-towards-v1-0-set-of-specifications/. Accessed: 2024-10-21.

[36] Ultra Ethernet Consortium. 2024. Ultra Ethernet Specification Update. https://ultraethernet.org/ultra-ethernet-specification-update/. Accessed: 2024-10-21.

[37] Shay Vargaftik, Ran Ben Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben Itzhak, and Michael Mitzenmacher. 2022. Eden: Communication-efficient and robust distributed mean estimation for federated learning. In *ICML*. 21984–22014.

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

[39] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *NeurIPS*, Vol. 32.

[40] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. 2024. Towards Domain-Specific Network Transport for Distributed

DNN Training. In *NSDI*. 1421–1443.

[41] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *EuroSys*.

[42] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *EuroSys*. 867–882.

[43] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: ternary gradients to reduce communication in distributed deep learning. In *NeurIPS*, Vol. 31. 1508–1518.

[44] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. 2017. On compressing deep models by low rank and sparse decomposition. In *CVPR*. 7370–7379.