FPGA-based SmartNIC for Distributed Machine Learning

Master Thesis

Author(s):
Hè, Hongyu

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000670931

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D INFK

FPGA-based SmartNIC for Distributed Machine Learning

Master's Thesis Nr. 483

Hongyu Hè

hongyu.he@inf.ethz.ch

Systems Group
Department of Computer Science
ETH Zürich

Supervisors:
Wenqi Jiang
Prof. Dr. Gustavo Alonso

April 29, 2024

Acknowledgements

I am deeply grateful to my advisor, Prof. Dr. Gustavo Alonso, and my mentor, Wenqi Jiang, for their invaluable guidance and support throughout this journey. Their mentorship has been instrumental in shaping my academic path, and I am immensely thankful for their expertise and support throughout. I would also like to extend my gratitude to the Systems Group at ETH for providing such a supportive and stimulating research environment. Interacting with brilliant researchers from around the world has greatly enriched my learning experience. Last but certainly not least, I am indebted to my mom and my grandparents for their unwavering support. Despite not being able to be with them in person for the past six years, their encouragement and love have been a constant source of strength.

Abstract

Over the past decade, there has been an exponential growth in both the size of AI models and the volume of training data. This surge has made distributed training imperative, where workers synchronize in lockstep by transmitting large gradients across the network. However, as compute power outpaces network bandwidth, communication overhead has emerged as a critical bottleneck in distributed training. In response to this challenge, various gradient compression methods have been proposed to reduce such communication overhead.

However, there are four main challenges that hinder the practical adoption of existing gradient compression methods. Firstly, existing GPU-based gradient compression methods contend precious resources with gradient computation, completely blocking the backward pass. As a result, they eliminate the opportunity for computation-communication overlap that is a crucial technique for speeding up distributed training. Secondly, the interval of time for gradient compression methods to process large amounts of gradient data is at the subsecond level, demanding extremely high throughput. However, previous methods often fail to meet such stringent performance requirements, leading to poor scalability and the waste of network bandwidth. Thirdly, most of existing methods are only compatible with certain cluster topologies, restricting their applicability and usability. Lastly, while the effectiveness of gradient compression depends on the workloads, existing methods are static. Consequently, they tend to perform well only for specific models and downstream tasks, while suffering substantial performance loss for other workloads.

To address these limitations, we introduce gCow, a hardware accelerator that can be integrated into FPGA-based SmartNICs attached directly to GPUs for on-the-wire gradient compression. gCow is platform- and topology-agnostic, designed to necessitate minimal application-level modifications from users and compatible with any collective communication primitives. We implement gCow as an open-source, easy-to-use high-level synthesis library. gCow can preserve full model quality on CIFAR10 and induces only 2-3% accuracy loss on ImageNet, while reducing network communication overhead by 50×. By exploiting both global dataflow pipelining and local block-level parallelism, gCow can saturate 100 Gb network bandwidth, achieving performance comparable to and even better than existing RTL counterparts. Furthermore, it empowers users with flexible tuning options to tailor the compression algorithm to specific user workloads, striking a desired balance between model quality and training speedup.

Contents

List of Figures

Chapter 1

Introduction

Machine learning (ML) has seamlessly integrated into our daily lives, revolutionizing virtually every sector of the industry. Its widespread success can be primarily attributed to two pivotal factors: the exponential growth in model size and the increasing amounts of training data. However, the continuous scaling in these two dimensions faces mounting challenges. With the rapid stagnation of Moore's Law post the end of Dennard Scaling nearly a decade ago, increasing the size of ML models and the amount of training data has become ever more challenging.

1.1 Motivation

The proliferation of ML models has witnessed an unprecedented surge in scale, with models now reaching trillions of parameters [1, 2, 3]. The Neural Scaling Laws [4, 5, 6] project an ongoing trajectory of increase in models size, which so far positively correlates with performance on downstream tasks, reflecting an incessant pursuit of higher model quality. However, the improvement of (fast) GPU memory capacity is unable to match this exponential increase. For instance, a model just over 30 billion parameters requires more memory to train than a single GPU can currently provide [7]. Moreover, ML training requires staggering amounts of data, for example, GPT-4 [2] was trained on four trillion tokens. Training such a model on a single device would have taken hundreds of years. Consequently, distributed ML training methods (§2.1) have seen widespread adoption, often involving hundreds or even thousands of devices working in lockstep.

Unfortunately, scaling ML training over a large compute cluster remains difficult due to many challenges, chief among which is the communication overhead between devices. Specifically, for synchronous data parallel training, the amount of data needed to be transmitted between two devices at *each* iteration is at least equal to the model size[1], which is typically on the scale of gigabytes. Moreover,

[1] Sometimes, other parameters from the optimizer state, such as the velocity values associated

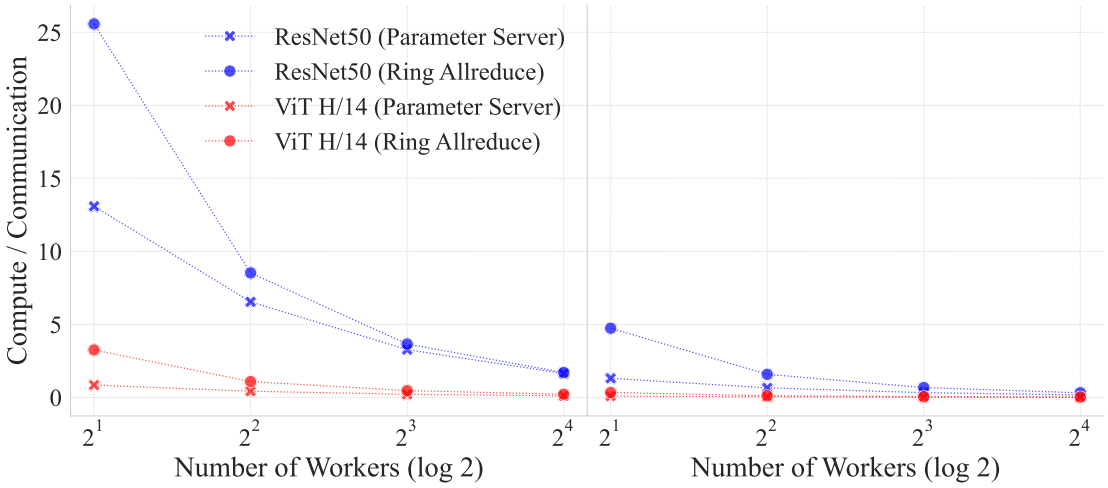


Figure 1.1: Compute to communication ratio for scaling the training of ResNet50 and ViT H/14 on NVIDIA GeForce RTX 3090 GPUs with 100 Gb/s (left) and 10 Gb/s (right) network bandwidths. The lower the ratio is, the more communication overhead the training process incurs.

such communication would have to happen multiple times per second. For example, training a relatively small model like ResNet50 (97.5 MB) two iterations per second on a 128-node cluster using Parameter Server (PS) [8] would require a network bandwidth of 24.38 GB/s. This requirement is even larger than what a high-end Mellanox Infiniband ConnectX-5 can provide, a 12.5 GB/s bandwidth. Other topologies such as Ring Allreduce (RA) [9] require less bandwidth but require more network hops.

Figure 1.1 shows the compute-to-communication ratio of training ResNet50 and ViT H/14 (2.5 GB) using PS and RA with different network bandwidth on A100 GPUs. Communication quickly dominates iteration time in both cases as the number of workers increases. This phenomenon is especially pronounced when the network bandwidth is scarce. Furthermore, ViT can result in more than 10× lower training efficiency compared to ResNet50 due to heavier communication costs even with large network bandwidth. This result indicates the mounting pressure on network communication as ML models become ever larger.

As a result, many techniques have been proposed to accelerate distributed ML training (§2.2), such as fine-grained pipelining [10, 11], model compression [12, 13, 14], and gradient compression [15, 16, 17]. This work focuses on gradient compression, which reduces the amount of network traffic by applying lossy compression on gradients before they are transmitted. Nevertheless, most of those methods are composable, e.g., applying both gradient compression and

with each model parameter (for computing gradient momentum), also need to be synchronized.

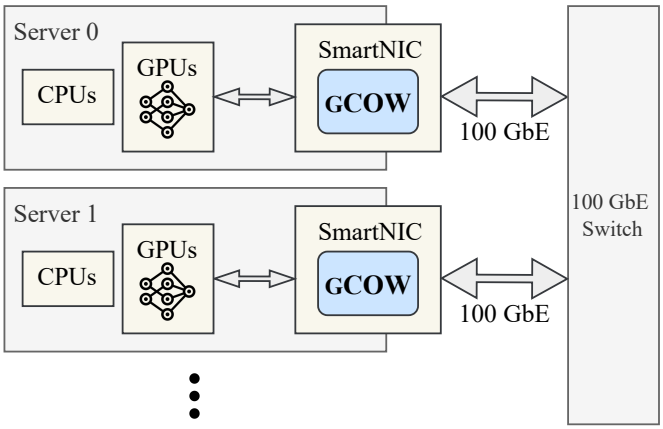


Figure 1.2: System overview of GCow.

fine-grained pipelining at the same time.

1.2 Gaps

Over the past years, many gradient compression methods have been proposed (e.g., [18, 19, 16, 15, 20]). They can achieve compression ratios over 100×, thereby reducing the network traffic and in turn, speeding up distributed training substantially. However, four major gaps (**G**s) in existing methods need to be filled in order to facilitate practical adoption of gradient compression methods:

G1: Existing GPU-based gradient compression methods contend precious resources with gradient computation, completely blocking the backward pass [21], and therefore, making overlapping gradient communication with compute infeasible.

G2: The interval of time for gradient compression methods to process large amounts of gradient data is at the sub-second level, demanding extremely high throughput. However, previous methods typically fail to meet such stringent performance requirements [21], leading to poor scalability and the waste of network bandwidth.

G3: Most of existing methods are compatible with certain cluster topologies (e.g., centralized parameter server or decentralized all reduce), restricting their applicability and usability.

G4: While the effectiveness of gradient compression depends on the workloads, existing methods are static. Consequently, they tend to perform well only for specific models and downstream tasks [18, 16], while suffering substantial performance degradation for other workloads.

1.3 Contributions

To address these limitations, we propose GCoW, a hardware accelerator that can be integrated with FPGA-based SmartNICs attached directly to GPUs for gradient Compression On the Wire (Fig. 1.2). Specifically, we make the following main contributions:

C1. We explore the use and characterize the effect of a new lossy compression method ZFP [22] in the context of gradient compression.

C2. We implement GCoW on as an open-source[2], easy-to-use high-level synthesis library for FPGAs, designed to necessitate minimal application-level modifications from users and allowing the overlap between gradient computation with communication (**G1**).

C3. By leveraging both a global dataflow pipeline and local parallelism, GCoW can saturate 100Gb network bandwidth, achieving throughput comparable to existing register transfer level (RTL) counterparts [23, 24, 25, 26](**G2**).

C4. GCoW is platform- and topology-agnostic, compatible with any collective communication primitives (**G3**).

C5. GCoW empowers users with several tuning options to tailor the compression algorithm to specific workloads, striking a desired balance between model quality and training speedup (**G4**).

1.4 Scope

As a proof-of-concept, we outline the scope of this work to ensure timeliness and feasibility. Firstly, we defer the implementation of GCoW's software driver suite for future work. While this task requires substantial engineering effort, it offers comparatively fewer research insights. Consequently, we postpone end-to-end testing at this stage. Nevertheless, we underscore the importance of showcasing end-to-end performance through validated performance models parameterized by real system measurements. Secondly, this thesis concentrates on investigating the feasibility of utilizing the ZFP CODEC for bump-in-the-wire (BITW) gradient compression, with the goal of optimizing throughput to achieve line rate. Therefore, while creating the necessary interfaces to the SmartNIC, we defer the integration with specific types of devices for future work, since such an integration would also entail developing handshaking protocols and traffic classification within the SmartNIC. Furthermore, although GCoW can potentially be applied to various other applications such as model-parallel paradigms, asynchronous

[2] GCoW is available at https://github.com/fpgasystems/gcow

training, or even additional forms of model compression, we focus solely on exploring its application in gradient compression within a data-parallel scheme in this study. We leave the exploration of other potential use cases for future research endeavors.

Chapter 2

Background

In this section, we briefly introduce some background knowledge as well as related work.

2.1 Distributed ML Training

Parallel Training Schemes. There are broadly two categories of distributed training schemes, namely, data and model parallelisms (Fig. 2.1). Each of the two training schemes can be either synchronous or asynchronous. Although here we focus on synchronous data parallel training only, the compression technique studied in this work can be applied to other schemes as well.

There are two commonly used cluster topologies in data parallelism: centralized Parameter Server (PS) and decentralized Allreduce. In PS, the training dataset is split into shards distributed equally among N workers. The entire ML model of size P is replicated N times. Each of the N workers will hold one such replica of size P. The workers in the cluster operate in lock steps — at each iteration, each worker trains their local replica on one batch of their local data shards and computes local gradients during backward pass. Instead of updating the local replicas using the computed gradients as what single-node training would do, the workers will first *push* their local gradients of size P to the head node for synchronization. Upon receiving the gradients from all workers ($N \cdot P$ gradient values in total), the head node will aggregate the gradient values and then *broadcast* the averaged gradients to all N workers. After updating their local model replicas with the received gradients from the head node, workers will proceed to the next training step together.

Unlike PS, Allreduce architectures do not have a centralized head node aggregating the gradient values at each iteration. Similar to PS, each worker holds one of the N shards of the training dataset and trains their local replica of the model on their local shards in lockstep. The main difference rests in the synchronization step — workers synchronize their gradient values using the *allreduce* collective operation instead of relying on centralized head nodes. The *allreduce* collective

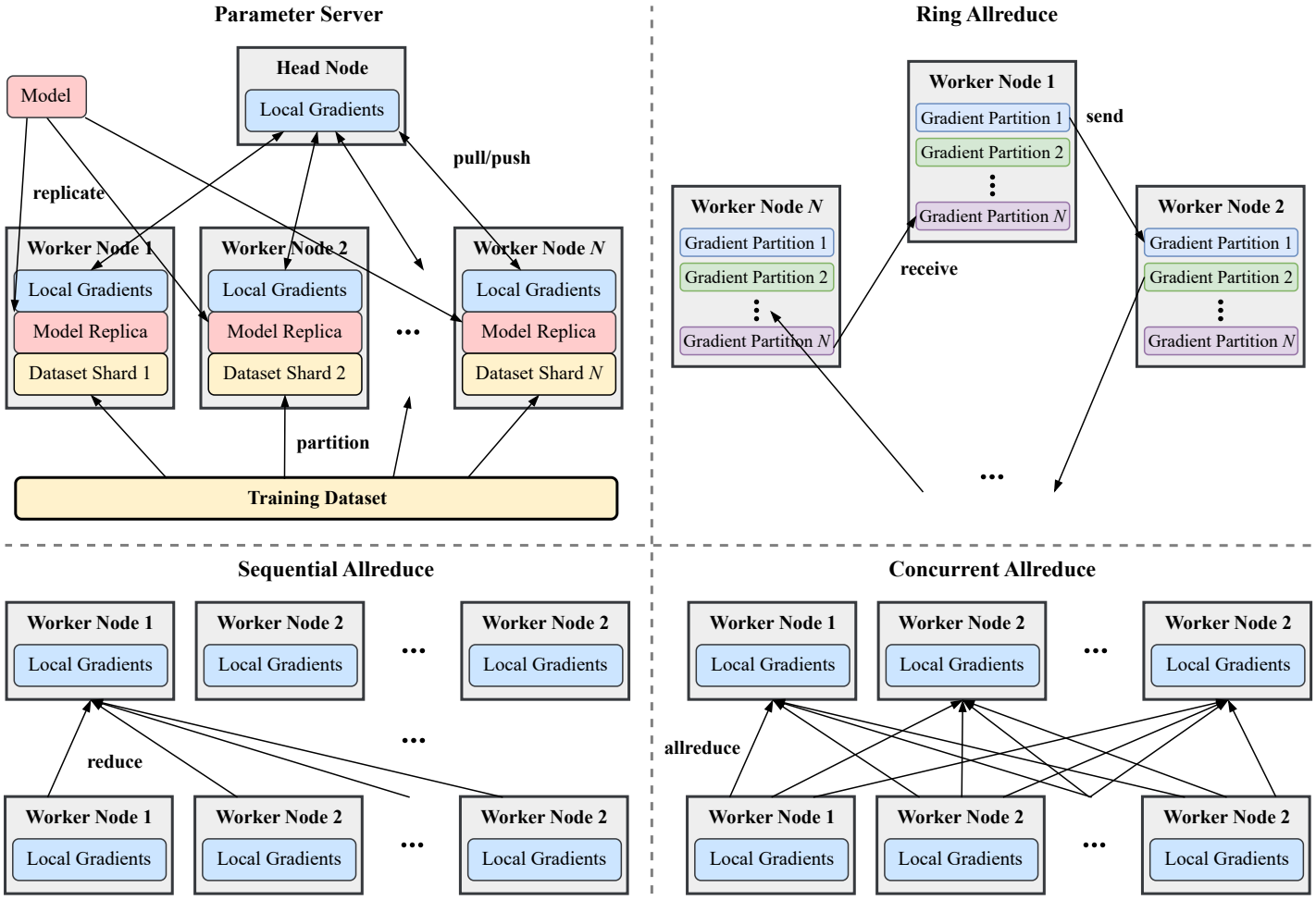


Figure 2.1: Common data parallel training schemes.

operation can be achieved with different communication patterns, chief among which are Sequential, Concurrent, and Ring Allreduce.

Sequential Allreduce (SA) synchronizes gradients by first executing a *gather* collective operation on each worker *sequentially*, pulling $(N-1) \cdot P$ gradients from all other workers, and then each worker averages the gathered gradients to update their local replicas. Instead of using a *gather* operation, Concurrent Allreduce (CA) conducts an *allgather* collective operation, in which all workers pull $(N-1) \cdot P$ gradients from all other workers at the same time, demanding $N\times$ more bandwidth than that of SA.

Ring Allreduce (RA) operates rather differently — each worker first splits their local gradients into N buckets. Next, the worker of rank i will send its i-th bucket of size (P/N) to the worker of $(i+1)$ and receive the $(i-1)$-th

Data Parallel Training Scheme	Compute Time	Peak Network Bandwidth
Parameter Server	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(N)$
Concurrent Allreduce	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(N^2)$
Ring Allreduce	$\mathcal{O}(N)$	$\boldsymbol{\mathcal{O}(1)}$
Sequential Allreduce	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Table 2.1: Costs of common data parallel training schemes.

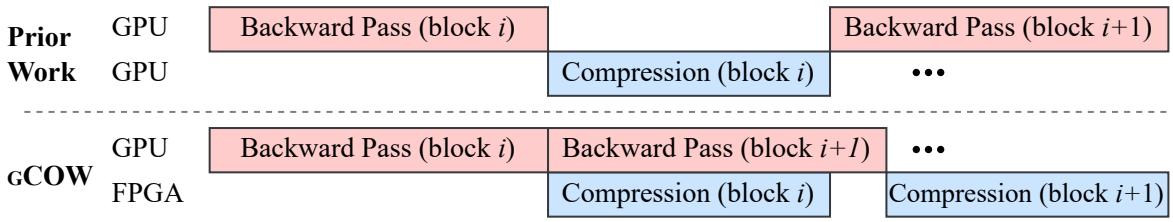


Figure 2.2: Overlapping gradient computation and gradient compression.

gradient bucket of the worker of rank $(i-1)$. One training step requires $(N-1)$ such communication rounds to synchronize gradient update among N workers. Table 2.1 summarizes the costs of the abovementioned data parallel training schemes. In Section 4.4, we study how gCow can potentially benefit the first three schemes.

Overlapping Communication with Compute. One crucial technique commonly employed to reduce the cost of gradient computation is overlapping gradient communication and its computation (Fig. 2.2). This technique can hide part of the communication overhead behind the backward pass. To leverage this method, the ML model is first split into several gradient buckets. In PyTorch, for example, this splitting is a greedy process — users specify the desired bucket size, and the framework will then greedily pack as many layers as it can meet the size limit.[1] Then, during the backward pass of gradient bucket i, the framework synchronizes the gradient values of bucket $(i-1)$. Prior work [21, 27, 28] has shown that overlapping the two can reduce training iteration time by almost 50%. Unfortunately, existing GPU-based work does not allow such an overlap, since gradient compression would contend with gradient computation for resources [21].

[1] https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

2.2 Accelerating Distributed ML Training

Many methods have been proposed to accelerate distributed ML training. Chief among those are fine-grained pipelining in model parallel training, model compression, and gradient compression. Fine-grained pipelining methods (e.g., [10, 11]) split one batch of training data into multiple training micro-batches and schedule pipeline-paralleled model layers to train on different micro-batches in a way that minimizes device idle time (i.e., the "bubbles" in training pipeline). These methods can either be synchronous or asynchronous and are complementary to model/gradient compression methods. Model compression methods aim to reduce the model size by quantizing (e.g., [29, 17]), pruning (e.g., [18, 20]) or sparsifying (e.g., [30, 31]) model parameters. These methods can greatly reduce model size, thereby reducing the amount of traffic for updating models on workers. However, since such methods directly operate on the models, they typically incur heavy degradation in model quality [17, 30, 18]. Gradient compression methods, on the other hand, reduce the amount of network traffic during the synchronization phase by directly compressing transmitted gradients among model replicas.

2.3 Gradient Compression Methods

Existing gradient compression methods are mainly GPU-based and can be broadly categorized into two types, namely, gradient pruning and numerical quantization. Typical gradient pruning methods include techniques such as gradient sparsification/selection [18, 32, 20], which selectively transmit subsets of gradients (e.g., only 0.1% of all values), and low-rank approximation [19] replacing gradient matrices with low-rank approximates. On the other hand, gradient quantization methods employ numerical methods to quantize individual gradient values, reducing the amount of storage required for each gradient.

Numerical quantization discretizes the real line into buckets. All numbers that fall into a single bucket will be represented by a single number, e.g., the average value of the bucket. The main difference between various quantization methods primarily lies in the size distribution of the buckets. For instance, the integer types in IEEE 754 standard splits the real line into equidistant buckets. One can quantize 32-bit floating point gradients by directly casting them to 8-bit or even 4-bit integers. A more refined approach would be using ML-specific data types such as bfloat16 [33]. These data types typically prioritize numerical range over precision, allocating more bits to exponent and reducing the number of bits assigned to mantissa. To achieve much higher compression ratios, many methods have been proposed to more aggressively reduce the number of bits used to represent gradient values, such as 1-bit [16] and ternary [17] data types.

Unfortunately, several limitations in existing methods hinder their practical

adoption. Firstly, GPU-based gradient compression methods contend precious resources with gradient computation, completely blocking the backward pass [21]. Secondly, the interval of time for gradient compression methods to process large amounts of gradient data is at the sub-second level, demanding extremely high throughput. However, previous methods typically fail to meet such stringent performance requirements [21], leading to poor scalability and the waste of network bandwidth. Thirdly, most of existing methods are compatible with certain cluster topologies (e.g., centralized parameter server or decentralized all reduce), restricting their applicability and usability. Lastly, while the effectiveness of gradient compression depends on the workloads, existing methods are static. Consequently, they tend to perform well only for specific models and downstream tasks, while suffering substantial performance degradation for other workloads.

2.4 ZFP Lossy Compression

In this work, we explore a new compression and decompression algorithm (CODEC), ZFP [22], in the context of gradient compression. This CODEC specializes in compressing floating-point and integer arrays and facilitates high-throughput I/O random access operations. ZFP features the ability to efficiently represent multi-dimensional numerical data in memory. This feature is particularly valuable for various computational tasks, such as differential equation solvers, data analysis, and visualization, offering substantial reductions in memory consumption.

2.4.1 Key Features of ZFP

We choose ZFP as the CODEC of gCoW for the following reasons. Firstly, this CODEC is commutative, meaning that the order in which it is applied to batches of input does not affect the final result. This feature is crucial for gCoW to be compatible with any cluster topologies that users may prefer (**C4**). Secondly, ZFP is highly versatile, offering both a lossless configuration and several lossy modes (§2.4.3). In contrast to existing methods that are mostly static, this flexibility is crucial to allowing users to adapt gCoW to their specific model-task combinations (**C5**), trading off the loss in model quality and training speedup at well. Thirdly, the CODEC was originally designed to mitigate data movement (e.g., to and from disk, across the internet, between compute nodes, and even through the memory hierarchy) for high-precision numerical use cases such as physical simulations, scientific observations, and experiments. As a result, it can achieve large compression ratios (5×–200× under the lossy modes and 1.5×–4× in the lossless configuration), while providing certain error bounds [34]. Therefore, theoretically speaking, the CODEC is able to substantially reduce the amount of transmitted gradient data without sacrificing too much ML model quality.

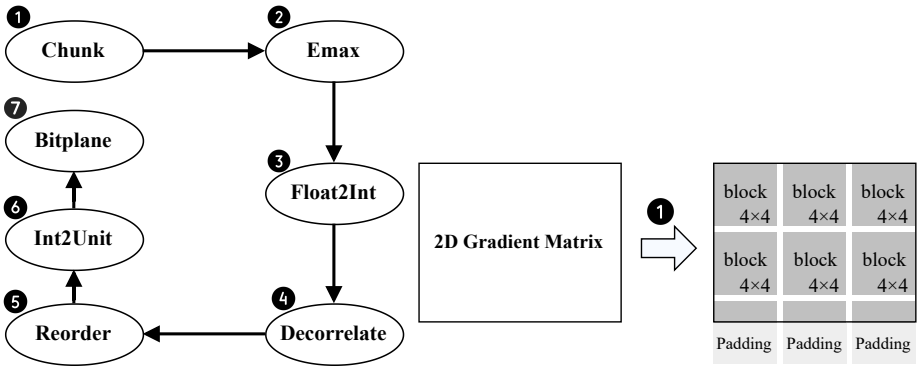
2.4.2 CODEC Pipeline

While providing several appealing features (§2.4.1), ZFP is more complex than existing gradient compression methods. Its CODEC pipeline consists of six stages (Fig. 2.3a).

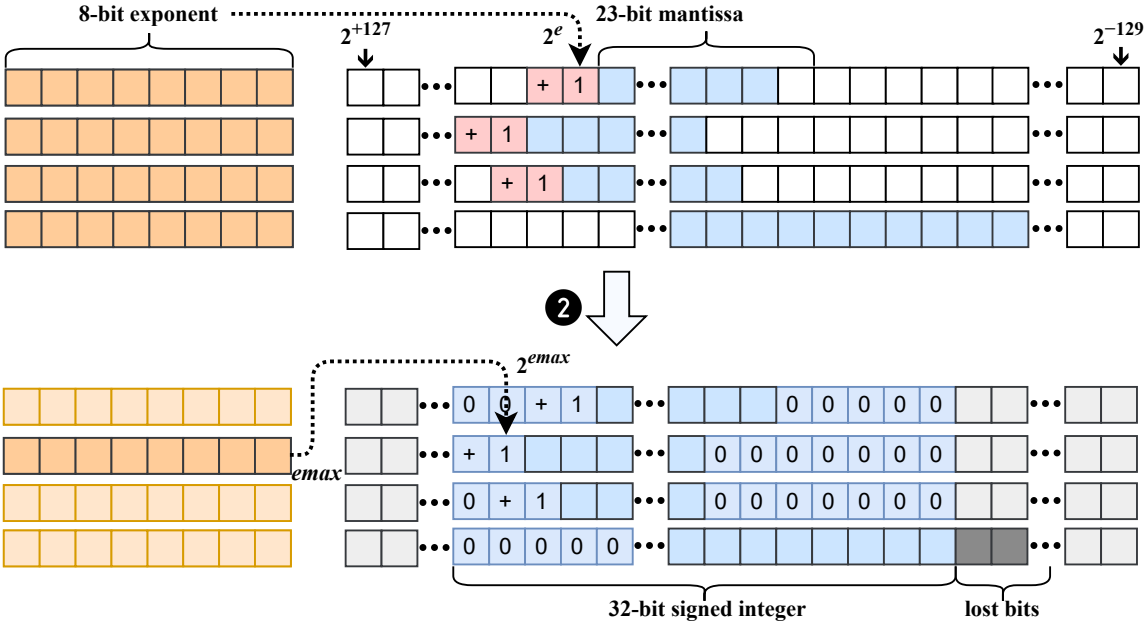
In the first stage `Chunk` (❶), data of $d \in (0, 4]$ dimensions is chunked into 4^d blocks. For example, a 2D matrix will result in a number of 4×4 blocks. In case an axis of the matrix is not divisible by 4, zero paddings are added (Fig. 2.3b). These blocks are the units of operations on which the CODEC will be applied *independently*. In other words, each block goes through the following pipeline stages separately. In the second stage `Emax` (❷), block floating point transform is applied to each block (Fig. 2.3c). In this stage, the maximum exponent of a block is taken as the common exponent of all the values therein and is plainly encoded (8 bits for single precision numbers and 11 bits for double precision).

In the third stage `Float2Int` (❸), the mantissas of all the numbers are shifted based on the common exponent and represented as 31-bit or 63-bit signed integers. Note that, when the range of a number is large, the remaining bits can be truncated. In addition, if all the values in the block are zero or less than the specified tolerance under the "fixed-accuracy" mode (§2.4.3), then a single zero-bit is encoded to indicate that this is effectively an "empty" block that can be directly expanded to zeros during decoding. Consequently, this is the first stage that can cause precision loss. Nevertheless, this data format prioritizes numerical ranges over precision, which adheres to existing the objectives of many existing ML data formats such as bfloat16 [33].

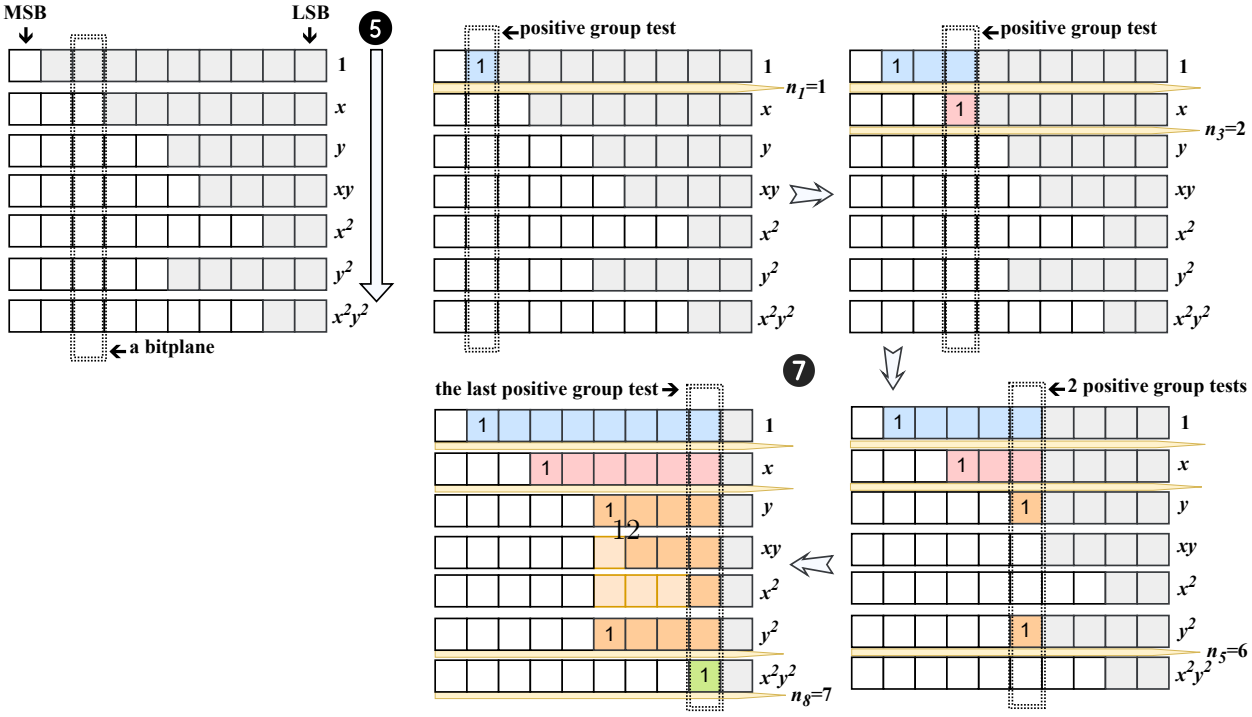
The `Decorrelate` stage (❹) is responsible for applying a near-orthogonal transform, similar to the discrete cosine transform (DCT II) used by the JPEG CODEC, on the block of signed integers, the mantissas, produced by the previous stage. This process is also known as "decorrelation" or "whitening," which aims to remove correlations between the features or components of the data. This step is commonly used in various CODECs, helping to reduce redundancy and improve the efficiency of compression algorithms, such as those based on entropy coding or transform coding. The decorrelation transform employed by ZFP is more efficient than most alternatives such as DCT II and discrete wavelet transform (DWT), since it can be implemented with a simple lifting scheme that consists only of shift and add operations [22]. Next, the `Reorder` stage (❺) is responsible for ordering the mantissas based on their frequencies in ascending order. Such an ordering is similar to the entropy encoding used in the CODEC of JPEG, in which the pixel blocks are encoded from the upper-left corner to the lower right corner in a zigzag manner. Intuitively, the coefficients of lower frequencies are listed first, since they tend to have larger magnitudes (Fig. 2.3d left). In a spatial sense, the `Reorder` stage will also order a 2D mantissa block with coordinates (i, j) first by $(i + j)$ then by $(i^2 + j^2)$.



(a) CODEC pipeline. (b) `Chunk` stage.



(c) `Emax` stage.



(d) `Reorder` and `Bitplane` stages.

Figure 2.3: Inner working of the implemented numerical CODEC from ZFP.

The sixth stage, denoted as the `Int2Uint` phase (❻), transforms the mantissas from two's complement signed integers into their corresponding negabinary (base negative two) representation. This conversion process involves performing one addition and one bitwise exclusive `OR` operation per integer. As negabinary lacks a dedicated sign bit, the resulting integers are subsequently treated as unsigned. In contrast to sign-magnitude representations, in negabinary, the leftmost one-bit simultaneously signifies the sign and approximate magnitude of a number. Additionally, unlike two's complement, negabinary representation results in leading zeros for numbers with small magnitudes, irrespective of their sign, which simplifies the encoding process.

In the last stage, `Bitplane` (❼), is an embedded coding procedure that exploits the fact that the mantissa values in negabinary form produced by the previous stage tend to have many leading zeros, which need to be explicitly encoded. Algorithm 1 specifies the details of the algorithm. This embedded coding algorithm encodes the mantissas from the bitplane of MSBs to that of the LSBs, using a technique called Group Test [35], a technique initially devised to check groups of consolidated blood samples from large populations for infectious diseases.

In the context of ZFP, the CODEC conducts group tests to check whether a group of mantissas is greater than a significance threshold, instead of testing every single mantissa one at a time. This process effectively translates to testing whether the bits higher than the i-th bit of all the values in a group contain at lease a one-bit (Fig. 2.3d right). Each group test splits the mantissas into two groups, one in which all values are assumed to be having at least one non-zero bit above the i-th bitplane and another group in which this property does not hold. For each positive group test, a one-bit is encoded, and negative tests are flagged by zeros. Consequently, the encodings contain both the flag bits of the group tests and the actual data. Such group testing and splitting is done recursively as groups become smaller and smaller until all values are either significant or insignificant. In the case of the "Fixed-Rate Mode" (§2.4.3), a bit budget is specified to stop the encoding/group testing once a certain number of bits have been encoded. Therefore, this stage is also a potential source of precision loss.

To achieve the recursive group testing, the CODEC algorithm employs a parameter n. It is accumulated from the encoding of the bitplanes of the MSBs to that of the LSBs (line 12 and 17). Specifically, when encoding the i-th bitplane, the i-th bits of the first n mantissas are estimated to be significant and are encoded verbatim (line 8), while the bits of the rest mantissas are not and in turn, needed to be further group-tested and potentially split into subgroups.

2.4.3 CODEC Modes

The *lossy* CODEC of ZFP [22] has three configurations, namely, fixed-rate, fixed-accuracy, and fixed-precision modes.

Fixed-rate mode. In this mode, each block of 4^d values are encoded using a fixed number of bits (*maxbits* in Algorithm 1). The encoding process stops as this threshold is reached. Since each block is an independent encoding unit, this mode effectively limits the *average* number of bits, the "rate", used for encoding each number in a block, i.e., rate = maxbits/4^b. For example, when rate is 8, (128+8) bits will be used in total to encode a 2D block of 16 numbers, each of which takes 8 bits on average. Note that, since a block at least requires 1 bit to indicate whether the numbers in the block are all zero or not, it takes at least 9 bits to encode a block of single precision numbers and 12 bits for double precision numbers of any supported dimensions.

Fixed-rate mode is essential for facilitating random access to encoding blocks and is the primary mode utilized in implementing ZFP's compressed arrays. It also guarantees a predictable memory or storage footprint. However, it typically yields lower accuracy per bit compared to the variable-rate but fixed-precision and fixed-accuracy modes.

Fixed-precision mode. This mode does not promise to use the same number of bits to encode every block. However, the number of encoded bitplanes, the precision, is guaranteed and specified by *maxprec* in Algorithm 1. This mode is preferred when relative error is more important than absolute error.

Fixed-accuracy mode. In this mode, all bit planes are encoded up to a minimum bit plane number. However, it is important to note that the actual minimum bit plane depends on the dimensionality d of the input. Variance may occur since the inverse transform during decoding entails range expansion, which varies based on the number of dimensions. This mode is controlled by a "tolerance" parameter, which should be interpreted as the base-2 logarithm of an absolute error tolerance. Specifically, for an uncompressed value, x, and a reconstructed value, $\hat{x}$, the absolute difference $\mid x - \hat{x} \mid$ is guaranteed to be at most $2\times$ the tolerance value. It is worth mentioning that achieving error tolerances smaller than machine epsilon relative to the largest value within a block is infeasible. However, this error tolerance can be conservatively set to ensure it is respected even for worst-case inputs (although it may not always be optimally tight), particularly for 3D and 4D arrays. Similar to the fixed-precision mode, the number of bits used per block may vary and is determined by the data itself. Setting the tolerance to 0 enables near-lossless compression.

Notably, fixed-accuracy mode offers the highest quality, measured in terms of absolute error, for a given compression rate, making it preferable when random access to encoding blocks is not required.

2.4.4 Challenges of ZFP

While ZFP offers a range of powerful features such as versatile configurations, high compression ratios, and error bounds under its lossy modes, it comes with several critical limitations as well. Firstly, it employs a bit-level compression CODEC, resulting in significantly lower throughput compared to other byte-level CODECs and compression methods. Additionally, ZFP is inherently sequential, both in terms of its staged global pipeline and individual stages. For instance, the `Bitplane` stage (❼) cannot be parallelized since the bitplanes must be encoded one after another in order. Similarly, encodings must be emitted in the order of their corresponding blocks, necessitating the ordering of stages involving I/O (e.g., ❶ and ❼).

Consequently, while there have been several attempts to implement ZFP in hardware (e.g., [24, 25, 36, 23]), they all require modifying the CODEC to make it more parallelizable, and thereby increasing achievable throughput. There is one hardware implementation [26] that adheres to the original CODEC, but it was based on a low-end device and demonstrated insufficient throughput for our purpose. Additionally, all existing works are implemented in RTL code, rendering them difficult to extend and integrate with other system components. Furthermore, although ZFP has demonstrated superior performance in scientific applications such as physical simulation and compressing massive amount of collected observation data, its applicability in gradient compression has yet to be systematically studied.

To address the challenges associated with employing the ZFP CODEC for line-rate gradient compression, we first characterize its performance in gradient compression under different configurations (§3.1) and then build GCoW (§3.2 and §3.3), which achieves high throughput (§4) while preserving *all* original features that ZFP has to offer.

Algorithm 1: Embedded Coding.

Input: Encoding stream s
Input: Array of mantissas in negabinary form *ublock*
Input: Number of mantissa values each block (= #bits/bitplane) *num*
Input: Bits per integer *intbits*
Input: Target precision *maxprec*
Input: Bit budget *maxbits*
Output: Number of encoded bits

```
   /* Index of the least significant bitplane to be encoded.   */
```
1 $kmin \leftarrow intbits > maxprec \; ? \; intbits - maxprec \; : \; 0$;
2 $bits \leftarrow maxbits$;
3 $i, k, m, n \leftarrow 0$;
```
   /* Encode one bitplane at a time.                           */
```
4 **for** $k \leftarrow intbits$ **downto** $kmin$ **do**
5 $\quad x \leftarrow 0$;
```
   /* Transpose values to bitplanes.                           */
```
6 $\quad$ **for** $i \leftarrow 0$ **to** $(num - 1)$ **do**
7 $\quad\quad x \leftarrow x + ((ublock[i] \gg k) \;\&\; 1) \ll i$;
```
   /* Encode first n bits of bit plane verbatim.               */
```
8 $\quad m \leftarrow \min(n, bits)$;
9 $\quad bits \leftarrow bits - m$;
10 $\quad$ Write m lower bits of x to s;
11 $\quad x \leftarrow x \gg m$;
12 $\quad$ **while** $bits > 0$ and $n{+}{+} < num$ **do**
13 $\quad\quad bits \leftarrow bits - 1$;
14 $\quad\quad bit \leftarrow x > 0 \; ? \; 1 \; : \; 0$;
15 $\quad\quad$ Write *bit* to s;
16 $\quad\quad$ **if** $bit \neq 0$ **then**
```
      /* Possitive group test.                                 */
```
17 $\quad\quad\quad$ **while** $bits > 0$ and $n{+}{+} < num$ **do**
18 $\quad\quad\quad\quad bits \leftarrow bits - 1$;
19 $\quad\quad\quad\quad bit \leftarrow$ LSB of x;
20 $\quad\quad\quad\quad$ Write *bit* to s;
21 $\quad\quad\quad\quad$ **if** $bit \neq 0$ **then**
```
         /* After encoding a 1, break out for another
            group test.                                       */
```
22 $\quad\quad\quad\quad\quad$ **break**;
23 $\quad\quad\quad\quad x \leftarrow x \gg 1$;
24 $\quad\quad$ **else**
```
      /* Negative group test:  only zero-bits left.           */
```
25 $\quad\quad\quad$ **break**;
26 **return** $maxbits - bits$;

Chapter 3

System Design of gCow

In this chapter, we describe the system design of gCow. We first present the characterization of ZFP in the context of ML gradient compression (§3.1). Then, we show how we use the Kahn Process Network (KPN) programming paradigm to construct a global dataflow pipeline among the CODEC stages (§3.2). To further boost the throughput of gCow, we exploit block-level parallelism within each processing stage (§3.3), making the dataflow system multi-rate. Finally, we briefly touch upon other optimizations and the integration interface to existing 100 Gb/s SmartNICs (§3.4).

3.1 Characterization of ZFP

In Section 2.4.3, we introduced the three lossy modes of the ZFP CODEC, each of which offers distinct features. Consequently, it is reasonable to expect that these compression modes entail different tradeoffs in the context of gradient compression.

3.1.1 Compression Ratio

As the effectiveness of compression CODECs varies depending on the workload, we begin by examining the compression ratio of ZFP under the three lossy modes in the context of gradient compression. To conduct this analysis, we extract the gradient values of ResNet50 at 10 random training steps and compress them using the ZFP CODEC.

The tuning spaces of the three modes are large, and enumerating all possible settings is infeasible. For instance, the accuracy parameter of the fixed-accuracy mode can range from anywhere between 0 and 1e-15. Therefore, we opt for four discrete configurations representing increasing levels of compression strength: high, medium, low, and minimum. Specifically, for the fixed-precision and fixed-rate modes, we explore four values: {4, 8, 16, 32} for both the precision and accuracy parameters, respectively. For the fixed-accuracy mode, we set

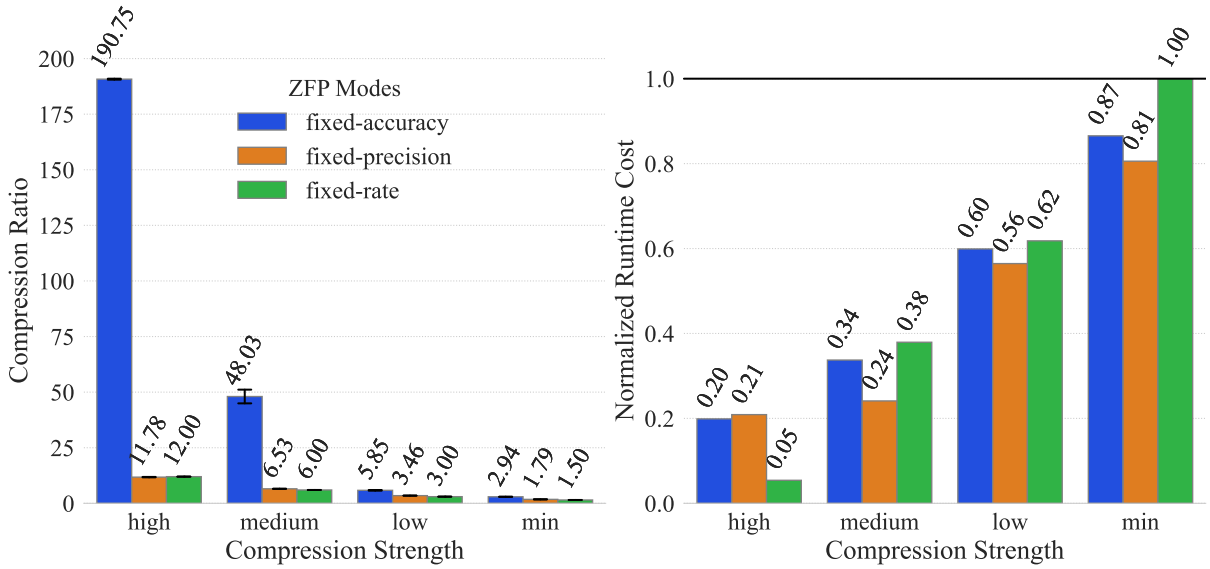


Figure 3.1: (Left) Compression ratio of the three lossy modes of the implemented CODEC from ZFP at four levels of compression strengths on gradients extracted from ResNet50. (Right) Normalized runtime cost of the corresponding compression modes.

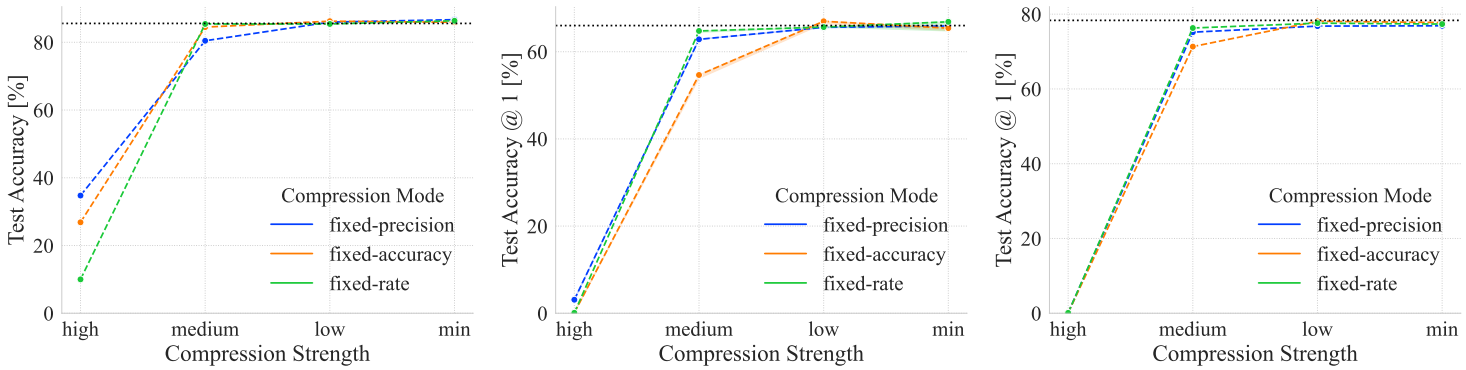
the accuracy target to {0.1, 1e-3, 1e-6, 1e-9} for the four levels of compression strengths. These values were selected to yield roughly similar runtimes for the three modes at the same compression levels. It is important to note that *lower* compression strength leads to *longer* CODEC processing times, as more information needs to be preserved during compression (Figure 3.1 right). Hence, there exists an inherent tradeoff between information loss and CODEC runtime.

Furthermore, the fixed-accuracy model generally yields the highest compression ratio at roughly the same runtime cost (Figure 3.1 left). Its advantage is increasingly pronounced as the compression strength becomes higher. The compression ratios of the other two modes are mostly on par across the four compression levels.

3.1.2 Workload-dependency of Compression Modes

To explore the effectiveness of the three lossy modes on different workloads[1], we vary both the ML model and training datasets. Specifically, we use two models of different sizes and architectures, namely, ResNet50 (97.5 MB) based on convolution and attention-based ViT H/14 (2.5 GB) [37]. Note that we use the software implementation of the ZFP CODEC for the following experiments

[1]Workload here refers to a specific combination of ML model and training dataset.



(a) ResNet50 performance on CIFAR10 with gradient compression using the ZFP CODEC.

(b) ResNet50 performance on ImageNet with gradient compression using the ZFP CODEC.

(c) ViT H/16 performance on ImageNet with gradient compression using the ZFP CODEC.

Figure 3.2: Characterization of ZFP CODEC in ML gradient compression. Dotted lines indicate test performance without gradient compression.

since gCow currently does not have a driver, so end-to-end testing is infeasible at the moment (§1.4). However, the CODEC implemented in gCow has been carefully validated against the ZFP software implementation[2], guaranteeing their consistent output given the same input. Therefore, the following characterization results also apply to our hardware implementation.

In Figure 3.2a, we train ResNet50 on CIFAR10 dataset [38] for 200 epochs and evaluate the model performance with the ZFP CODEC under the three compression modes. As compression strength reduces, the corresponding model quality improves, and we can achieve the original accuracy at medium compression strength under the fixed-accuracy and fixed-precision modes. These two modes can achieve around 50× and 7× communication reduction respectively at a medium compression level (Fig. 3.1). This result not only serves as a validation of the applicability of the ZFP CODEC in the context of gradient compression but also demonstrates its promising performance on par with prior works [16, 18, 19, 20]. This result also implies that the three compression modes demonstrate different tradeoffs at each compression level.

The model used in Figure 3.2b is still ResNet50, but it is trained on ImageNet [39], one of the largest and most comprehensive image datasets. The model is trained under each compression mode for 50 epochs. Firstly, the three compression modes demonstrate a rather different tradeoff space compared to that of CIFAR10. For instance, the worst-performing mode at the medium compression level changed from the fixed-precision mode to fixed-accuracy. Nevertheless, at the low compression level, accuracy is able to match the model accuracy of

[2] https://github.com/LLNL/zfp (commit: fa1014f5e)

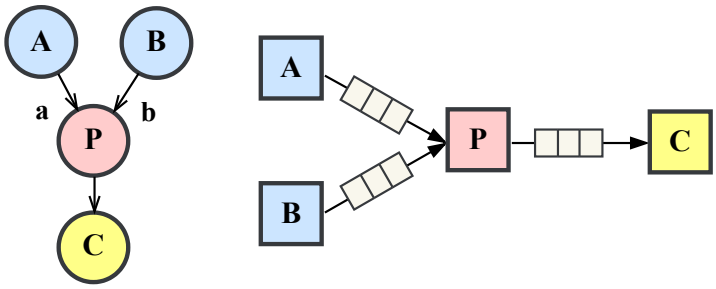


Figure 3.3: Example Kahn Process Network.

without applying gradient compression. Furthermore, another important observation is that high compression strength no longer works for ImageNet, which is a much larger and harder downstream task compared to CIFAR10. This result illustrates that the tradeoff space induced by the three compression modes is task-dependent.

In Figure 3.2c, the model been trained is changed from ResNet50 to a much larger one, ViT H/14. Similar to the result shown in Figure 3.2b, high compression strength will cause the training process to diverge, leading to non-usable models. The overall accuracy pattern resembles that of ResNet50 in that, by slightly lifting the compression strength, model accuracy can reach the no-compression level very quickly. For example, with a median compression strength (which can reduce the communication overhead by around 50×), the model only demonstrates approximately 3% performance degradation. Moverover, compared to the result from ResNet50, ViT seems to be less sensitive and therefore more robust to changes in gradient compression strength, which might be a result of larger model capacity. Thus, the ML model quality induced by the different ZFP modes is also model-dependent.

Given the above characterization, we can conclude that (1) the ZFP CODEC has competitive performance in the context of gradient compression compared to state-of-the-art methods, and (2) the three compression modes are workload-dependent and induce different tradeoff spaces as workload varies. Hence, we decide to support all three compression modes instead of only a subset of them in building GCoW.

3.2 Global Dataflow Pipeline

The staged CODEC of ZFP (Fig. 2.3a) greatly resembles a Kahn Process Network (KPN) [40]. KPNs are the foundation of dataflow computation theory, where sequential processes represented by nodes running concurrently that communicate through *single-producer*, *single-consumer* first in first out (FIFO) channels represented by edges (Fig. 3.3). These communication channels are assumed to be

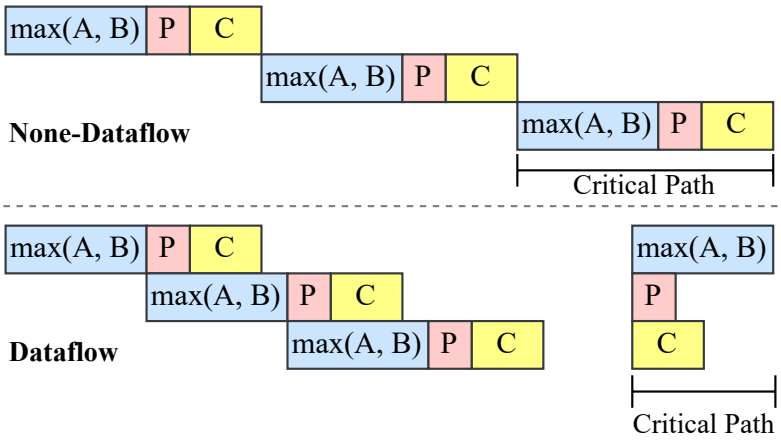


Figure 3.4: Dataflow can reduce the critical path of a computation graph.

infinite in size, thereby implying that read operations could be blocking but writes are always non-blocking. The pieces of data passed from one stage to the next are "tokens," and once all required input tokens arrive, a node becomes "fireable." All nodes in the network are always in either of the two following states: (1) waiting for required input tokens, or (2) running its computation to produce tokens for the subsequent node(s).

Dataflow networks are a simpler and more practical implementation of KPNs. They are mostly the same but have two notable differences. Firstly, unlike infinite communication channels from the theoretical model of KPNs, nodes exchange data through single-directional, *finite* first-in-first-out (FIFO) queues in dataflow networks. Consequently, both read and write operations can be blocking. Moreover, the behaviors of nodes are required to be *deterministic* — the number of tokens produced and consumed are supposed to be verified at compile time, mitigating issues such as deadlock. Not only does dataflow provide a deterministic concurrency model based on message-passing and shared-nothing principles to flexibly model parallel, iterative applications, but it also reduces the critical path of the computation graph of the target application (Fig. 3.4).

To leverage the dataflow paradigm, we model the stages of the ZFP CODEC as nodes in a computation graph, each operating independently and concurrently (Fig. 3.5). We merge the `Reorder` and the `Int2Uint` stages from the original CODEC (Fig. 2.3a) into one stage, and `Float2Int` is renamed to `Cast` for brevity. The tokens communicated between these nodes are the 4^d blocks, treated as individual operation units. Our dataflow system is implemented using Vitis HLS, connecting nodes via a non-blocking `stream` interface. This global pipeline improves the overall system throughput by approximately 5× (Fig. 4.1). However, this design alone is insufficient for GCOW to achieve the desired performance, as the system throughput remains at the same level as the software

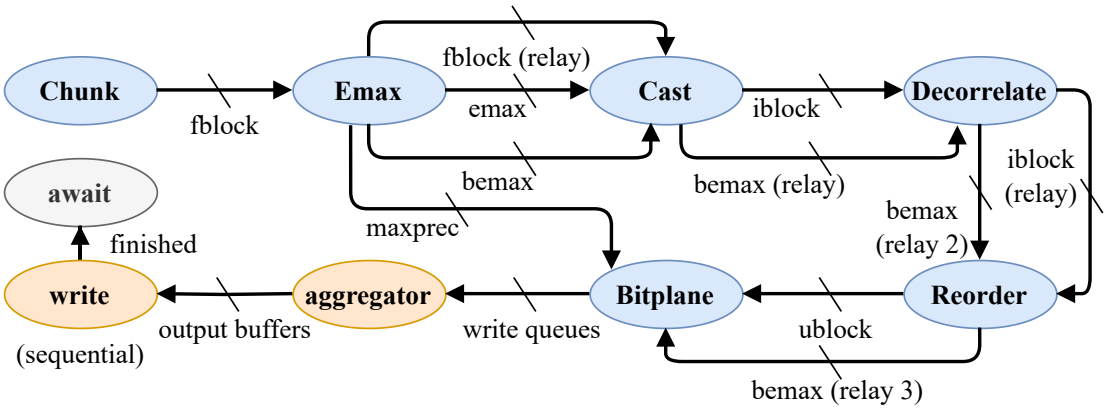


Figure 3.5: Global dataflow pipeline of GCOW. The values on the edges are dataflow tokens passed among nodes. Blue nodes are the CODEC stages, and those in orange and gray are I/O and auxiliary stages, respectively.

implementation. Given that the critical path of a dataflow system is determined by the slowest node, we must identify the bottleneck stages in the CODEC and exploit parallelism at the block level to further enhance throughput.

3.3 Local Block-level Parallelization

In the CODEC pipeline, each 4^d block undergoes independent compression. As a result, multiple blocks can be processed simultaneously without compromising numerical precision. To enable this parallel operation, we can create K parallel processing elements (PEs) within every dataflow node and increase the bandwidth of all FIFO queues by a factor of K. This global strategy can ideally boost overall system throughput by a factor of K. However, deploying this approach for the CODEC implementation proves infeasible due to its complexity — it quickly depletes FPGA routing resources. For instance, setting $K = 64$ causes routing failures attributed to congestion on an AMD Alveo U250 FPGA [41], a high-end data center accelerator card.

To address the resource constraints, we implement a *multi-rate dataflow pipeline* in which nodes possess varying levels of parallelism and, in turn, different token processing rates. Specifically, we adjust K based on the stage — introducing K_i parallel PEs within the node of the i-th CODEC stage. This optimization presents two primary challenges. Firstly, determining K_i necessitates a per-stage throughput analysis. Secondly, varying processing rates among dataflow nodes can easily lead to deadlocks, overflowing or exhausting the communication channels between stages. To prevent deadlocks, we carefully tune the depths of FIFO queues between nodes to accommodate accumulated tokens, which could otherwise fill up buffers and block the entire dataflow system. The current solution

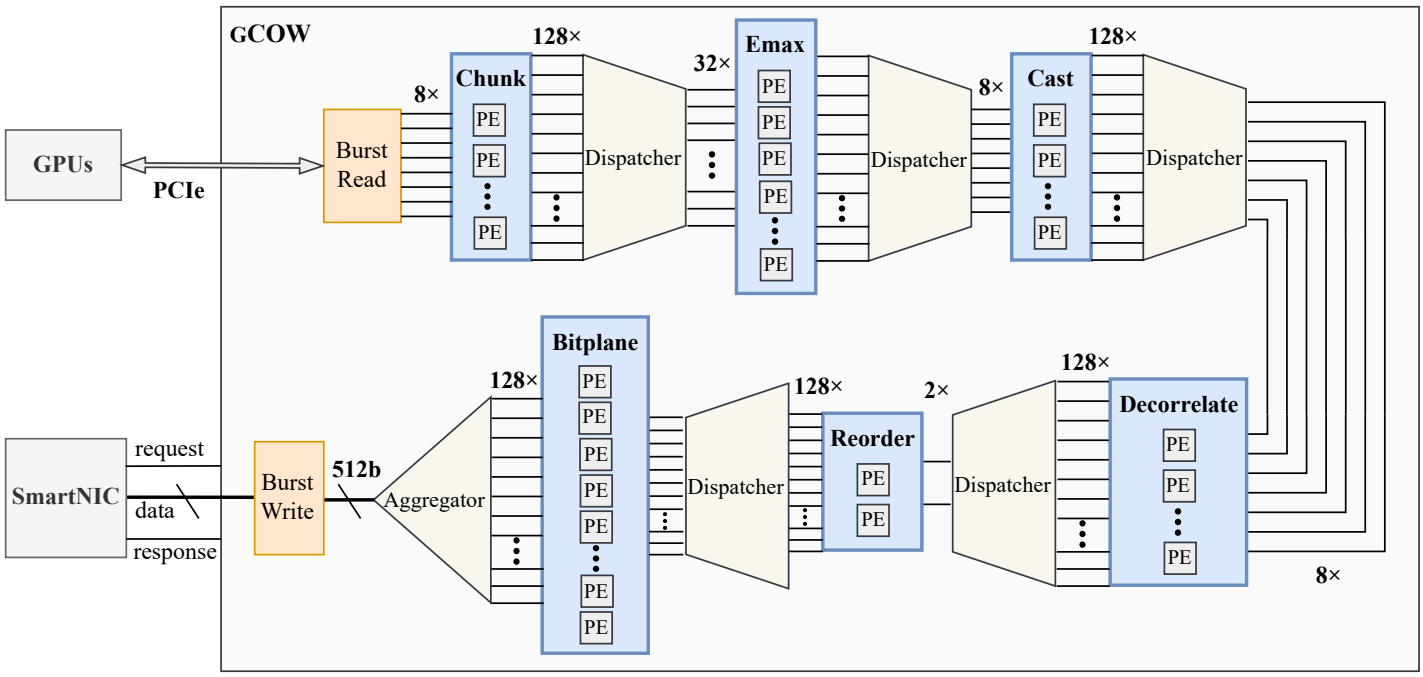


Figure 3.6: Overview of GCOW implemented as a multi-rate dataflow system.

involves conducting performance analyses for each CODEC stage (Fig. 4.2). Since the overall system performance is dictated by the worst-performing stage, we select K_i to ensure the i-th CODEC stage meets its throughput target without utilizing excessive resources that could otherwise further improve performance. Based on K_i and the processing rate of the corresponding stage, we determine optimal FIFO buffer sizes between stages. This approach allows for more efficient resource allocation to CODEC stages that appear as bottlenecks in the dataflow network, enabling them to achieve higher degrees of block-level parallelism, while mitigating the constraint on routing resources.

3.4 Other Optimizations and Interface to SmartNICs

In addition to the global dataflow network and local block-level parallelism, several other optimizations have been implemented.

Concurrent Memory Access. Block RAMs (BRAMs) have limited numbers of available memory ports, supporting only one or two concurrent reads and writes, which becomes a bottleneck for achieving high block-level parallelism. Rather than utilizing block BRAMs for the parallel FIFO buffers between dataflow nodes, we employ distributed LUT RAM. In this way, multiple PEs can

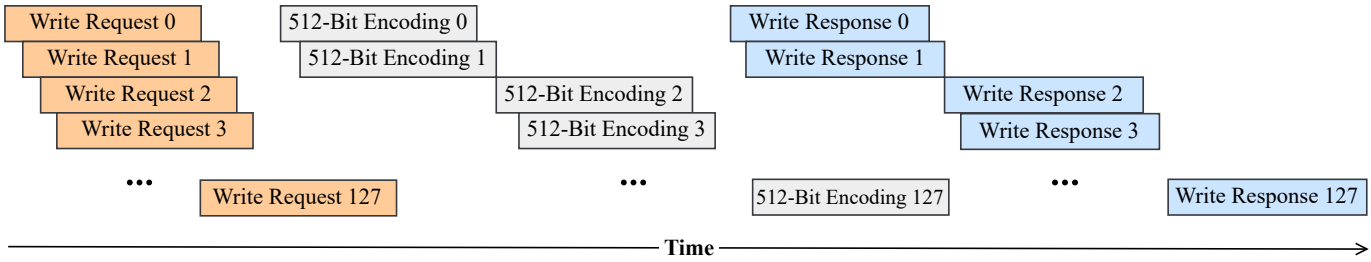


Figure 3.7: Pipelined burst memory writes, assuming two memory write ports are available.

read from and write to the queues concurrently, attaining desired parallelism.

Burst Memory I/O. One caveat with Vitis Shell is that unaligned memory access could significantly reduce achievable memory bandwidth and in turn, prevent GCow from reaching line-rate performance. In fact, sequential accesses to unaligned memory addresses could limit the memory bandwidth to as low as approximately 25 Gb/s. To avoid this issue and maximize throughput, GCow writes encodings to memory in bursts of 128 512-bit write operations[3]. A burst of such writes is pipelined (Fig. 3.7), which can provide a theoretical throughput of $(512 \times 250 \text{ MHz})/1 \text{ second} \approx 15$ GB/s with a DDR memory. This design not only makes sure that the memory accesses are of multiple of 64 bytes but also let the `Burst Write` component (Fig. 3.6) expose a 512-bit FIFO stream interface, making it compatible with existing FPGA-based SmartNICs and easier to pipeline the packet transactions [42]. The `request` and `response` wires are for handshakes with the SmartNIC to send TCP/IP packets through the 512-bit `data` bus.

[3]512 bits is the maximum kernel interface bitwidth in Vitis HLS.

Chapter 4

Evaluation of gCow

This chapter presents a comprehensive evaluation of gCow, focusing on addressing the following key questions:

(1) How much speedup can the global dataflow design offer? (§4.1)

(2) How much speedup can block-level parallelism offer? (§4.2)

(3) How does gCow compare to other ZFP hardware accelerators and existing GPU-based gradient compressors in terms of throughput? (§4.3)

(4) How much performance benefits can gCow provide for scaling out of distributed ML training with different cluster topologies? (§4.4)

In addressing the above evaluation questions, we describe analyses and experimental results across the specified sections to illustrate the effectiveness and comparative performance of gCow within the context of ZFP hardware acceleration and distributed ML training.

4.1 Speedup from Dataflow

To analyze the benefits of the dataflow network, we conduct evaluations comparing two versions of gCow: one implemented without a dataflow scheme and another rewritten using a dataflow paradigm. For our evaluations, we utilize exponentially distributed synthetic workloads devoid of zero values to maintain a conservative assessment. Figure 4.1 shows that employing dataflow yields approximately a 5× speedup over the naive implementation, achieving a throughput around 0.05 GB/s. This speedup is attributed to the global dataflow pipeline, which substantially reduces the critical path (Fig. 3.6).

However, despite this performance improvement, this level of throughput does not yet provide a competitive edge over the original ZFP CODEC, which is a single-core software implementation of the ZFP CODEC running on Intel Xeon

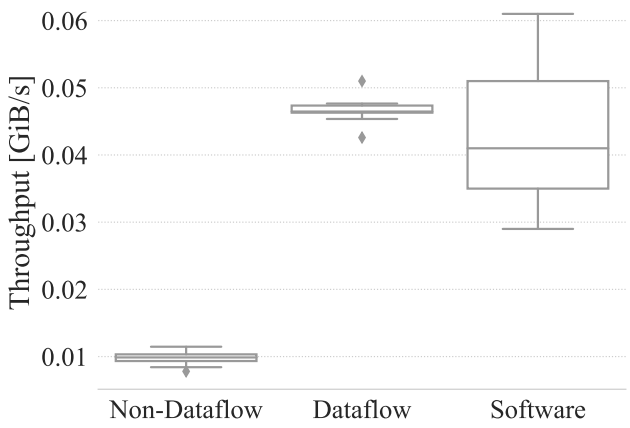


Figure 4.1: Dataflow design provides around 5× speedups over the non-dataflow version.

Gold 6248 CPU @ 2.50 GHz. While demonstrating a larger variation in performance, the software version can still outperform the naive dataflow implementation.

4.2 Speedup from Block-level Parallelism

Given that every 2^d block of d-dimensional gradient values progresses through the CODEC pipeline independently (§2.4.2), these blocks serve as the smallest operation elements and parallel units. To improve throughput, we aim to leverage block-level parallelism within each dataflow node. However, uniformly increasing parallelism across all CODEC stages can quickly deplete the routing resources of a high-end FPGA card (§3.3), causing prohibitive congestion.

To efficiently allocate limited onboard resources, we first identify bottleneck stages by measuring the throughput of each CODEC stage individually. Figure 4.2 illustrates that three stages emerge as major bottlenecks: `Emax`, `Cast`, and `Bitplane`. Consequently, we allocate more resources to these bottleneck stages to achieve higher degrees of parallelism compared to other stages that can reach the target throughput with fewer parallel PEs. To do so, we transition gCow into a multi-rate dataflow network in which the i-th node contains K_i PEs based on corresponding performance analysis. This approach balances resource allocation across stages, while ensuring all nodes in the network achieve the desired throughput. Notably, certain CODEC stages (e.g., `Reorder`) are more amenable to parallelization than others (e.g., `Bitplane`). While all other stages can reach the performance target with the help of block-level parallelism, even with 128 parallel PEs, the final throughput of `Bitplane` is still slightly below the target throughput as it is ultimately constrained by heavy I/O operations. Therefore,

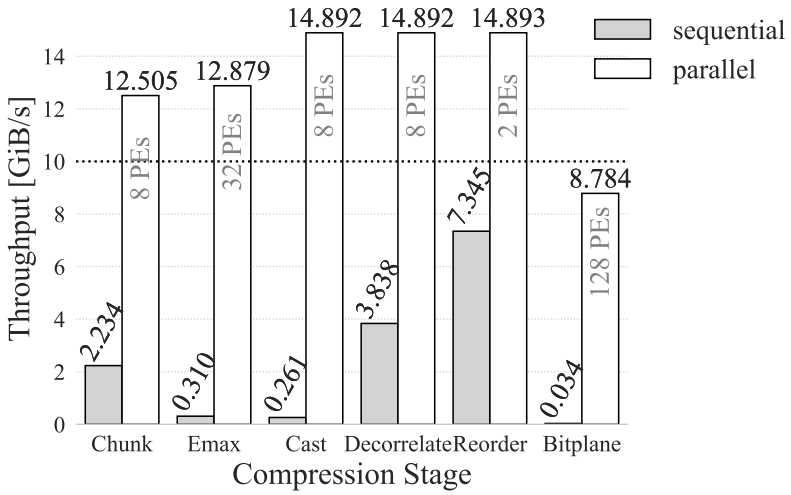


Figure 4.2: Speedup provided by using varying number of parallel PEs within each dataflow node. The dotted line indicates the target throughput, i.e., the 100 Gb/s SmartNIC bandwidth.

Resource	Usage (util%)
LUT	380,591 (19.6%)
Register	358,933 (10.4%)
BRAM	228.5 (8.5%)
DSP	441 (3.6%)
Power (W)	18.285

Table 4.1: Resource utilization and power of gCow.

two CODEC stages achieving the same level of throughput may require significantly different numbers of PEs. Table 4.1 lists the power and resources used by the final implementation of gCow on an AMD Alveo U250 data center card [41].

4.3 Comparing with Other ZFP Accelerators

This section compares gCow with existing ZFP-based hardware implementations and GPU-based gradient compressors. Firstly, we compare gCow with five existing hardware accelerators for ZFP-based CODECs: ZFP-V [25], sZFP [23], DE-ZFP [36], ZHW [26], and ZFPe [24] in Figure 4.3. Our evaluation dataset and clock frequency are aligned with most of these works for a more intuitive comparison, considering the challenges arising from variations in FPGA devices used by these accelerators. Specifically, gCow is implemented on an AMD Alveo U250 data center card, clocked at 250 MHz. This frequency is used by most of the referenced works. We evaluate gCow on the SDRBench lossy compres-

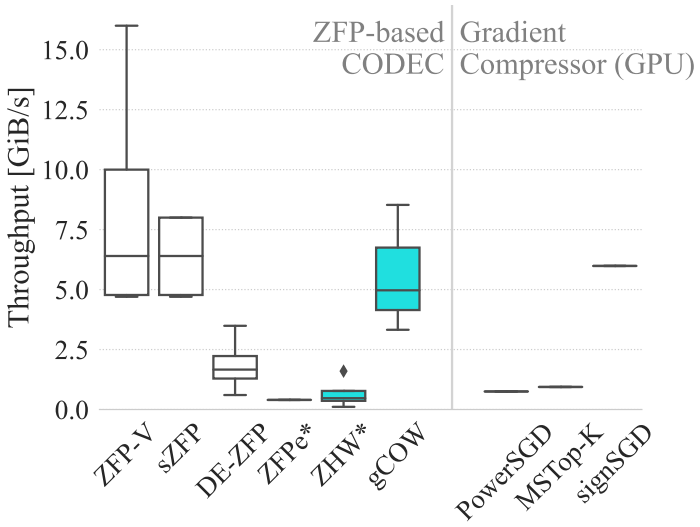
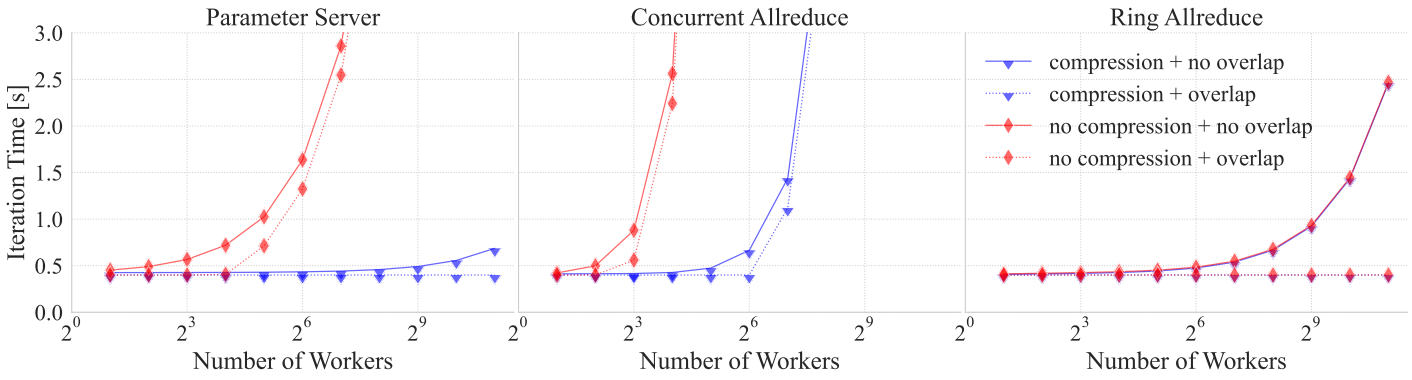


Figure 4.3: Throughput comparison with existing hardware accelerators of ZFP-based CODEC on the SDRBench lossy compression dataset [43]. Blue color indicates that the corresponding implementation preserves all features that the original ZFP CODEC has to offer. The GPU-based gradient compressors on the right part are evaluated with ResNet50's training.

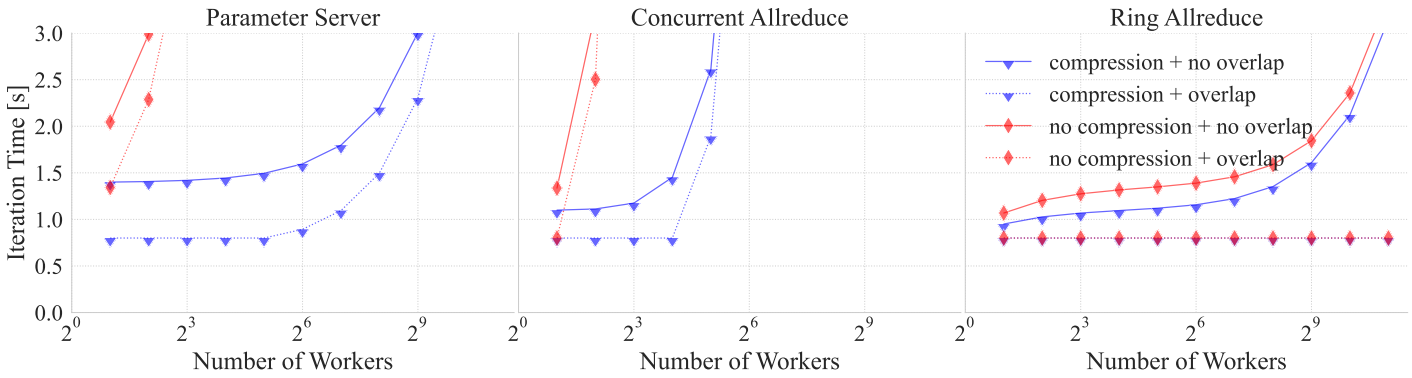
sion dataset [43] for scientific computing, which is also the most commonly used workload among these accelerators for evaluation. Notably, among the five accelerators, only ZHW and gCow preserve all ZFP features (§2.4.1), whereas others have modified the ZFP CODEC to enhance throughput. For instance, DE-ZFP substitutes the `Bitplane` stage with a dictionary encoding algorithm to boost throughput at the cost of lower compression rate. Furthermore, gCow stands out as the sole HLS implementation, offering superior user-friendliness and customization compared to RTL-based implementations of other accelerators. Despite being implemented in HLS, gCow achieves an average throughput of around 7 GiB/s, which is comparable to ZFP-V and sZFP, outperforming DE-ZFP, ZHW, and ZFPe.

In the right part of Figure 4.3, we compare gCow with GPU-based gradient compressors —PowerSGD [19], MSTop-K [18, 20], and signSGD [44] — using their reported throughput values during the training of ResNet50. gCow exhibits significantly higher throughput than PowerSGD and MSTop-K, and matches the performance level of signSGD. Note that signSGD's simplicity in algorithm comes at the expense of heavier model performance loss.

These comparisons highlight gCow's competitive performance against both existing accelerators of ZFP-based CODEC and GPU-based gradient compressors, positioning it as a promising solution for efficient and scalable machine learning model training.



(a) Scaling the training of ResNet50.



(b) Scaling the training of ViT H/14.

Figure 4.4: Scaling ML training with GCoW in simulation. `(no) compression`: whether or not to use GCoW to compress gradient values for worker synchronization. `(no) overlap`: whether or not to overlap gradient computation with communication.

Parameter	Value
Local batch size	256
ResNet50	
Compute time	300 ms
# gradient buckets	4
ViT H/14	
Compute time	800 ms
# gradient buckets	11
GCoW	
Compression ratio	50
Throughput	10 GiB/s

Table 4.2: Simulation configuration parameterized by the measurements from executions on NVIDIA A100 GPU and previous evaluation of GCoW.

4.4 Scaling out ML Training with GCoW

In this section, we explore the potential benefits of using GCoW to scale out ML training to thousands of nodes using simulation. To this end, we employ a validated performance model [21] parameterized by the measurements that we obtained from real-world executions on NVIDIA A100 GPUs.

In Figure 4.4, we vary the number of workers from 2 to 1024 and plot the duration per training iteration. In these experiments, we focus on the impact of network traffic going through the NIC and ignore the intra-node communication among GPUs within the same worker. We employ two models of drastically different sizes and architectures, namely ResNet50 and ViT H/14. Their runtime was measure on NVIDIA A100 GPUs on the ImageNet dataset [39] with a batch size of 256 for both models. The compression ratio of GCoW is set to 50 (Fig. 3.1). Detailed simulation parameters are summarized in Table 4.2.

Firstly, both the small (ResNet50) and large (ViT H/14) models can greatly benefit from GCoW when scaled out to large numbers of workers using cluster topologies that are constrained by network bandwidth, i.e., PS and CA (Table 2.1). Specifically, for PS, GCoW can provide 7.5× and 14.8× speedup on average for ResNet50 and ViT H/14, respectively. For CA, the speedups are 18× and 22×. Hence, the higher bandwidth the training scheme requires, the more speedup GCoW can provide. However, in the case of RA, whose bandwidth requirement scales linearly with the number of worker nodes in the cluster, the speedup is less significant — GCoW provides no speedup for ResNet50 and 1.2× speedups on average for ViT H/14.

Furthermore, overlapping gradient communication and computation has demonstrated its paramount role. Specifically, GCoW can provide an *additional* 1.3× speedup for ResNet50 across the three training schemes, compared to without

the overlap. Notably, gCow can make PS scale even better with the help of the overlap using PS than RA. The benefit of such compute-communication overlap is similarly pronounced for lager models like ViT H/14, offering an additional 1.6× speedup on average. However, as model size increases, gCow can no longer bridge the performance gap between the training schemes that are constrained by bandwidth (PS and CA) and those are not (RA). In other words, choosing a training scheme whose bandwidth requirement scales linearly with the size of the cluster is still the most important consideration in training models of hundreds of millions of parameters and beyond, even when gradient compression methods are employed.

Chapter 5

Future Work and Conclusion

Future Work. While gCow provides a rich set of configurations (§3.1) that allow the users to more flexibly navigate the tradeoff space, balancing speedup and model quality, as well as adapting the CODEC to specific workloads, this freedom can also lead to the difficulty of choosing an appropriate configuration given a model-dataset combination. Therefore, an automatic tuning method for the desired configuration of the CODEC is imperative. Secondly, one major limitation of gCow is its high resource requirements (Table 4.1). Although resources are allocated efficiently based on the degrees of required parallelism of each CODEC stage in the dataflow network (§4.2 and §3.3), gCow still requires almost 10× more FPGA resources compared to existing accelerators of ZFP-based CODEC [26, 36]. Furthermore, although this work focuses on data parallel training (§2.1), gCow is versatile and agnostic to the workloads as well as the training platform. For example, prior work [45] has shown that various model parallelisms possess different characteristics when it comes to gradient compression. Thus, other potential applications of gCow in for example, tensor and pipeline parallel training should be further explored. Additionally, this work does not consider other commonly used techniques such as momentum SGD with error feedback and correction [15, 46, 16], local gradient accumulation [15, 47], and adding warmup period at the start of training [15, 48, 49, 19]. These techniques could further improve the performance of gCow. Moreover, although integrating SmartNIC with GPUs have been proposed, existing work is limited in usability and compatibility with many mainstream devices [42]. As a result, improving SmartNIC-GPU integration is an important step forward. Last but not least, gCow currently applies the same CODEC configuration to all layers of the model. Varying compression configuration for different layers or model components is a promising future direction.

Conclusion. This work presents gCow, a novel framework enabling gradient compression on FPGA-based SmartNICs for distributed ML training. We are the first to demonstrate the effectiveness of the ZFP CODEC as a gradient compressor. Our characterizations show that ZFP achieves competitive performance

compared to existing methods, balancing compression ratio and accuracy loss effectively. Specifically, gCow can preserve full model accuracy on CIFAR10 and incurs only a 2–3% top-1 accuracy degradation on ImageNet, while reducing network communication overhead by 50×. Moreover, gCow inherits all the versatile features of the ZFP CODEC, allowing users to adapt the CODEC to various ML model types and downstream tasks. Additionally, implemented in HLS, gCow provides superior extensibility and usability compared to other RTL counterparts. By leveraging a global dataflow network and local block-level parallelism, gCow achieves near line-rate throughput of around 7 GiB/s on average. This performance is comparable to or exceeds that of existing hardware accelerators for ZFP-based CODECs and GPU-based gradient compressors.

Through a validated performance model for distributed ML training, we demonstrate that, when scaling out to large numbers of workers using data-parallel paradigms, gCow can accelerate the training of ResNet50 and ViT H/14 by 7.5–18× and 14.8–22×, respectively. Furthermore, unlike existing GPU-based gradient compressors, gCow allows for overlapping gradient communication with its computation during the backward pass, resulting in additional speedups of 1.3× and 1.6× on average for ResNet50 and ViT H/14, respectively. Despite several remaining opportunities and challenges, gCow demonstrates promising potential as a solution to achieving line-rate gradient compression on FPGA-based SmartNICs for distributed ML training.

References

[1] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[4] Y. Bahri, E. Dyer, J. Kaplan, J. Lee, and U. Sharma, "Explaining neural scaling laws," *arXiv preprint arXiv:2102.06701*, 2021.

[5] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[6] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[7] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 583–598.

[9] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[10] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[11] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[12] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[13] J. Kim, S. Park, and N. Kwak, "Paraphrasing complex network: Network compression via factor transfer," *Advances in neural information processing systems*, vol. 31, 2018.

[14] Y. Idelbayev and M. A. Carreira-Perpinán, "A flexible, extensible software framework for model compression based on the lc algorithm," *arXiv preprint arXiv:2005.07786*, 2020.

[15] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[16] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth annual conference of the international speech communication association*, 2014.

[17] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *Advances in neural information processing systems*, vol. 30, 2017.

[18] S. Shi, X. Zhou, S. Song, X. Wang, Z. Zhu, X. Huang, X. Jiang, F. Zhou, Z. Guo, L. Xie *et al.*, "Towards scalable distributed training of deep learning on public cloud clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 401–412, 2021.

[19] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[20] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 2238–2247.

[21] S. Agarwal, H. Wang, S. Venkataraman, and D. Papailiopoulos, "On the utility of gradient compression in distributed training systems," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 652–672, 2022.

[22] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[23] G. Sun, S. Kang, and S.-W. Jun, "Burstz: a bandwidth-efficient scientific computing accelerator platform for large-scale data," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[24] S.-M. Lim and S.-W. Jun, "Mobilenets can be lossily compressed: Neural network compression for embedded accelerators," *Electronics*, vol. 11, no. 6, p. 858, 2022.

[25] G. Sun and S.-W. Jun, "Zfp-v: Hardware-optimized lossy floating point compression," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 117–125.

[26] M. Barrow, Z. Wu, S. Lloyd, M. Gokhale, H. Patel, and P. Lindstrom, "Zhw: A numerical codec for big data scientific computation," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–9.

[27] P. Damania, S. Li, A. Desmaison, A. Azzolini, B. Vaughan, E. Yang, G. Chanan, G. J. Chen, H. Jia, H. Huang *et al.*, "Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[28] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch fsdp: experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[29] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[30] H. Wang, S. Sievert, S. Liu, Z. Charles, D. Papailiopoulos, and S. Wright, "Atomo: Communication-efficient learning via atomic sparsification," *Advances in neural information processing systems*, vol. 31, 2018.

[31] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[32] X. Ye, P. Dai, J. Luo, X. Guo, Y. Qi, J. Yang, and Y. Chen, "Accelerating cnn training by pruning activation gradients," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*. Springer, 2020, pp. 322–338.

[33] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[34] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, "Error analysis of zfp compression for floating-point data," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1867–A1898, 2019.

[35] E. S. Hong and R. E. Ladner, "Group testing for image compression," *IEEE Transactions On image processing*, vol. 11, no. 8, pp. 901–911, 2002.

[36] M. Habboush, A. H. El-Maleh, M. E. Elrabaa, and S. AlSaleh, "De-zfp: An fpga implementation of a modified zfp compression/decompression algorithm," *Microprocessors and Microsystems*, vol. 90, p. 104453, 2022.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[38] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, no. 7, pp. 1–9, 2010.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[40] K. Gilles, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, no. 471-475, pp. 15–28, 1974.

[41] AMD, "Alveo u250 data center accelerator card," 2024. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

[42] Z. Wang, H. Huang, J. Zhang, F. Wu, and G. Alonso, "{FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 967–986.

[43] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "Sdrbench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE international conference on big data (Big Data)*. IEEE, 2020, pp. 2716–2724.

[44] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signsgd: Compressed optimisation for non-convex problems," in *International Conference on Machine Learning*. PMLR, 2018, pp. 560–569.

[45] S. Bian, D. Li, H. Wang, E. P. Xing, and S. Venkataraman, "Does compressing activations help model parallel training?" *arXiv preprint arXiv:2301.02654*, 2023.

[46] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi, "Error feedback fixes signsgd and other gradient compression schemes," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3252–3261.

[47] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: scalable, low-cost training of massive deep learning models," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 472–487.

[48] J. Ma and D. Yarats, "On the adequacy of untuned warmup for adaptive optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 8828–8836.

[49] A. Gotmare, N. S. Keskar, C. Xiong, and R. Socher, "A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation," *arXiv preprint arXiv:1810.13243*, 2018.