CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Reducing MPI Communication Latency with FPGA-Based Hardware Compression

Accelerating Communication in Distributed Deep Learning Through Low-Latency Data Compression in HPC Clusters

Master's thesis in Computer science and engineering

ANIS BOURBIA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Reducing MPI Communication Latency with FPGA-Based Hardware Compression

Accelerating Communication in Distributed Deep Learning Through Low-Latency Data Compression in HPC Clusters

ANIS BOURBIA

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Reducing MPI Communication Latency with FPGA-Based Hardware Compression
Accelerating Communication in Distributed Deep Learning Through Low-Latency Data Compression in HPC Clusters
ANIS BOURBIA

Supervisor: Mateo Vázquez Maceiras, Computer Science and Engineering
Supervisor: Fareed Mohammad Qararyah, Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Reducing MPI Communication Latency with FPGA-Based Hardware Compression
Accelerating Communication in Distributed Deep Learning Through Low-Latency Data Compression in HPC Clusters
ANIS BOURBIA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

High-performance computing (HPC) clusters face significant communication overhead in distributed deep learning, where frequent data exchanges via the Message Passing Interface (MPI) can bottleneck overall training. This thesis explores an FPGA-based hardware compression approach to reduce MPI communication latency. We prototype integrating an FPGA compression module into the MPI stack, enabling on-the-fly compression of message payloads using fast lossless algorithms LZ4, Snappy, and Zstd. This hardware-accelerated compression offloads work from CPUs/GPUs and shrinks data volume before network transmission, thereby speeding up inter-node communication. In our evaluation, LZ4/Snappy/Zstd achieved compression ratios of 1.53x/1.51x/1.84x and reduced communication time by 34.6%, 33.8%, and 45.7%, yielding overall training speedups of 1.34x, 1.32x, and 1.50x, respectively. Experimental evaluation on representative deep learning workloads demonstrates up to a 1.50x improvement in end-to-end training time with the FPGA compression enabled. Among the tested compressors, Zstd achieved the highest compression ratio, translating to the greatest latency reduction and performance gain. These results highlight that FPGA-based compression can substantially improve throughput in distributed training by alleviating network delays, with negligible added overhead. The proposed method offers a practical path to accelerate HPC communications and scale deep learning workloads more efficiently.

Keywords: Computer, science, computer science, engineering, project, thesis, compression, FPGA, GPU, acceleration, HPC, DNN, lz4, zstd, snappy, lossless compression, MPI, networking, smart-NIC.

+

Acknowledgements

I would like to express my gratitude to my supervisors, Mateo Vázquez Maceiras and Fareed Mohammad Qararyah, for their guidance, constructive feedback and support throughout this project. I also thank my examiner, Prof. Pedro Petersen Moura Trancoso for his insightful critiques and for ensuring the quality of this work.

Anis Bourbia, Gothenburg, 2025-10-17

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

High-performance computing (HPC) clusters typically coordinate workloads through the Message Passing Interface (MPI), the *de facto* industry standard that defines the syntax and semantics for internode communication [1]. Modern HPC installations are increasingly employed to train deep learning models that contain billions or even trillions of parameters. While the computational power of GPUs has grown rapidly, the communication subsystems have struggled to keep pace, often turning inter-node data exchange into a critical bottleneck [2]. For example, in distributed training of large convolutional networks such as VGG19 consisting of 143 million parameters, communication overhead can account for up to 83% of the total training time, and even a 25 million-parameter model like ResNet50 can spend around 72% of its time in communication [2]. These figures underscore that simply scaling compute nodes is not sufficient, network latency and bandwidth limitations can severely throttle overall performance.

High-performance computing (HPC) clusters typically coordinate workloads through the Message Passing Interface (MPI), the de facto standard for inter-node communication [1]. Modern installations increasingly train deep learning models with billions or trillions of parameters. While GPU compute has grown rapidly, the communication subsystem has not always kept pace, so inter-node data exchange can become a bottleneck [2]. For example, in distributed training of large convolutional networks such as VGG19 (143M parameters), communication can account for up to 83% of training time; even ResNet50 (25M parameters) can spend around 72% in communication [2]. These figures underscore that simply scaling compute nodes is not sufficient with the large-messages typical of gradient exchanges, sustained bandwidth (throughput) is the dominant limiter, whereas message-startup latency mainly affects the first chunk and control/small messages. To avoid ambiguity, we use latency to mean message-startup latency, and throughput to mean the sustained data rate.

One promising direction to mitigate this gap is data compression. By reducing the volume of data transmitted, compression can alleviate bandwidth pressure and potentially reduce communication time. However, conventional compression techniques in a distributed training context face challenges. Software-based compression running on CPUs or GPUs introduces extra load, latency, which can negate the ben-

efits of sending smaller messages if the compression step cannot be overlapped with communication [2]. GPU-oriented solutions, such as NVIDIAs NCCL with hardware compression, have shown improvements in throughput but still compete for GPU resources that might otherwise be used for training computations [2] and would require the cluster to be equipped with Nvidia GPUs. Recent studies report that state-of-the-art GPU compression frameworks can accelerate deep learning training by up to 35.7% and reduce collective broadcast latency by as much as 80.9% [3], [4]. This motivates exploring other avenues of using dedicated hardware accelerators in parallel to GPUs for compression that operate in isolation to the main computation. In particular, Field Programmable Gate Arrays (FPGAs) have emerged as attractive candidates due to their ability to provide low latency, high throughput processing with custom logic tailored to specific tasks. In this work we therefore target throughput-bound gradient exchanges with FPGA-offloaded, streaming compression, while quantifying the changes to end-to-end traning time.

FPGAs offer deterministic latency and can exploit fine-grained parallelism, making them well-suited for streaming data applications. They have already been used in HPC for tasks such as network packet processing and encryption offload, delivering better realtime performance than software solutions in many cases [5], [6]. Moreover, preliminary work has demonstrated up to 3.9x, speed-ups for selected MPI collectives when offloaded to reconfigurable logic [7]. By dedicating an FPGA to handle compression of communication data, we can avoid overloading the CPUs or requiring vendor-specific GPUs to accomplish this task. Our approach proposes to integrate an FPGA-based compressor into the MPI communication pipeline, effectively compressing messages on the fly as they leave a node and decompressing as they arrive. This aims to reduce message sizes and thus transfer times without stalling the compute side, since the FPGA operates concurrently. In doing so, we dedicate hardware resources solely to compression, eliminating the contention for compute cycles that hampers traditional software solutions [6]. If successful, this could substantially shrink effective communication latency and improve overall scalability of distributed training.

1.2 Problem Statement

The efficiency of HPC clusters in domains like deep learning and scientific computing hinges on fast internode communication. Today's clusters rely on the MPI standard for coordinating work across nodes. Even with high-bandwidth interconnects such as InfiniBand and advanced networking hardware, communication overhead remains a principal bottleneck for many workloads [2]–[7]. As deep neural networks (DNNs) grow in size, the volume of data exchanged per training iteration also grows, often outstripping the improvements in network bandwidth. The result is that communication times can dominate, putting an upper bound on scaling efficiency. This thesis addresses bandwidth-dominated communication limits in distributed DNN training. High latency in exchanging model updates can slow down the iteration time, and insufficient effective bandwidth after accounting for protocol overheads and data sizes limits how quickly nodes can synchronize.

Prior research has explored various software-level and GPU-accelerated compression techniques to tackle these communication challenges. For instance, techniques that compress gradients or model parameters have been shown to speed up distributed training by reducing message sizes, albeit sometimes at the cost of losing some information accuracy [8]. It has been shown that error-bounded lossy compression can significantly improve collective communication performance in MPI, with recent frameworks demonstrating multifold speedups by tolerating slight inaccuracies in data [8], [9]. However, purely software or GPU-based compression approaches face tradeoffs. Software compression may not keep up with the high throughput required by fast networks and can introduce unacceptable delays. GPU-assisted compression integrates better with the training process, but as noted, it shares GPU resources with training tasks and can incur non-negligible overhead but the most significant argument against the proposed GPU compression solutions is the vendor-specific reliance.

In light of these challenges, this research investigates an alternative approach, an FPGA-based hardware compression solution dedicated to the MPI communication path. The problem we aim to solve is how to leverage FPGA acceleration to transparently compress and decompress data in MPI messages, thereby reducing the communication time between nodes without modifying the training algorithms themselves or incurring large overhead on the main processors. We specifically target the communication bottlenecks in distributed deep learning, where large volumes of floating-point tensor data e.g., gradients in FP32/FP16 or other parameter updates are exchanged frequently. The question is whether such an FPGA solution can increase throughput and reduce message-startup latency enough to yield a tangible reduction in end-to-end training time.

Unlike prior software-based MPI compression efforts or GPU offloading approaches, our contribution lies in three main aspects: (i) adapting vendor FPGA compression kernels to support streaming payloads rather than memory-mapped input, enabling them to realistically model MPI traffic; (ii) analyzing their potential impact on MPI communication through trace-driven simulation; and (iii) developing a reproducible workflow that combines real deep learning communication traces with hardware-characterized throughput models inside SimGrid.

1.3 Research Questions

The aim of this thesis is to design and evaluate an FPGA-based hardware compression mechanism to alleviate communication bottlenecks in distributed deep learning. We therefore pose the following research questions:

- RQ1: How can FPGA-based compression be integrated into the existing MPI communication stack?
- RQ2: What is the impact of the proposed solution on throughput and the scalability of distributed training?
- RQ3: How does FPGA-based compression compare to GPU-assisted and

software-based compression in terms of latency and throughput?

1.4 Ethical Considerations

This work follows standard principles of responsible research including honesty in reporting, transparency of methods, and reproducibility of results. We avoid selective reporting, and ensure that negative or neutral results are included when relevant. All claims are supported by measurable evidence, and limitations are stated explicitly.

To enable independent verification, we document software versions, configurations, and experimental procedures, and provide the artifacts needed to re-run our experiments [10]. External code and datasets are used in accordance with their respective licenses and cited appropriately. No proprietary information, non-disclosure agreements, or similar restrictions are involved in this work. There are no financial or personal conflicts of interest. Overall, the study is intended to be replicable, auditable, and useful to the community, aligning with institutional research ethics guidelines and good scientific practice.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2, Background, introduces the relevant scientific context and reviews related work in the field. We discuss distributed deep learning and the MPI communication model, highlighting why communication is a bottleneck, and we survey prior approaches to mitigate this bottleneck (including compression techniques). We also survey a number of compression algorithms and their characteristics in this chapter, explaining the rationale for the ones chosen in our hardware design. In chapter 3 we further look into the related work in the domain. Chapter 4, Methodology, describes the design and implementation of our proposed hardware compression solution. It details the FPGA development process, the system integration approach, the experimental environment, and the evaluation methodology including metrics. Chapter 5, Results, presents the performance outcomes of our FPGA-based compressor. We report on compression throughput, latency, and compression ratios obtained using real-world data, and we provide an analysis comparing our results to baseline software/GPU compression to interpret the benefits and potential trade-offs of the FPGA approach. Chapter 6, Conclusion, summarizes the findings of the thesis and discusses their implications. We reflect on the research questions, discuss to what extent they were answered, and outline directions for future work, such as dynamic reconfiguration and broader applicability of the approach.

2

Background

There is a wide consensus that communication remains the bottleneck in large computing clusters, particularly as the demand for HPC computing continues to grow. Over recent years, there has been significant effort to reduce this bottleneck, but it still persists, especially in inter-node communication. While intra-node communication has seen substantial advancements such as nodes featuring 8 or 16 GPUs interconnected using technologies like NVLink [11] these improvements amplify the need for efficient inter-node communication. The increased vertical scaling within nodes, enabled by tightly coupled GPUs, makes horizontal scaling between nodes even more critical in terms of communication efficiency. Indeed, numerous studies have observed that as compute power grows, network communication can become the dominant factor limiting performance [12]. Various network technologies (InfiniBand, high-speed Ethernet, etc.) deliver impressive raw bandwidth and low latency, yet the gap between node-local throughput and inter-node communication remains problematic. This gap motivates exploring new approaches to inter-node data exchange beyond conventional network upgrades.

2.1 Distributed Deep Learning (DDL)

Deep learning (DL) has revolutionized many fields, including computer vision, natural language processing, and scientific computing. As models continue to grow in size and complexity, training them on a single machine has become increasingly impractical. This has led to the widespread adoption of Distributed Deep Learning (DDL), where training is parallelized across multiple nodes, often using high-performance computing (HPC) clusters.

In DDL, model parameters such as weights and gradients are shared among multiple devices to ensure consistency. A commonly used approach for this is data parallelism, where each worker processes a different subset of the training data and computes gradients independently. These gradients must then be aggregated, typically by using a message passing communication protocol, where updated weights are communicated to all involved processes. This pattern introduces a heavy reliance on inter-node communication, which becomes the performance bottleneck at scale.

The frequency and volume of data exchanged during DDL make efficient communication essential. A single iteration of backpropagation can involve transmitting many of gigabytes of gradient data, and with state-of-the-art models exceeding trillions of

parameters, the overhead grows with model size. As highlighted in prior work [12], this communication overhead can dominate the training time, especially when the interconnect bandwidth is limited or the compute-to-communication ratio is low.

Compression is of particular interest because it can be orthogonal to other optimizations. However, many software-based approaches introduce latency due to CPU involvement and memory movement overheads. These delays can negate the benefits of reduced message sizes.

This motivates the exploration of hardware-accelerated compression, especially using FPGAs, as a way to compress communication payloads transparently without modifying the deep learning framework or model architecture. Hardware-based solutions promise low-latency, high-throughput compression with minimal CPU/GPU interference, potentially unlocking significant speedups in DDL workflows running on large-scale HPC clusters.

2.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to enable communication between processes in a distributed computing environment. It is widely adopted in HPC due to its efficiency, scalability, and compatibility across platforms. MPI allows for both point-to-point communication (e.g., `MPI_Send`, `MPI_Recv`) and collective communication operations (e.g., `MPI_Allreduce`, `MPI_Broadcast`) that involve groups of processes within a communicator.

In this work, we specifically target the Open MPI implementation of the MPI standard. Open MPI is a popular open-source implementation that supports a wide range of hardware architectures and interconnects, making it suitable for large-scale HPC and DDL workloads. When referring to MPI throughout this thesis, we refer to Open MPI unless stated otherwise.

MPI plays a critical role in distributed training by enabling synchronization and data exchange across compute nodes. Its design emphasizes low-latency and high-throughput communication, which are essential characteristics for modern distributed workloads. Among the various functions defined in the MPI standard, collective operations are of particular interest, as they encapsulate higher-level communication patterns frequently used in DDL. Collectives like `MPI_Allreduce` and `MPI_Broadcast` are performance-critical in synchronous training environments.

One of the most critical collectives in DDL is `MPI_Allreduce`, which performs a reduction (e.g., sum of gradients) followed by distributing the result to all processes. Standard algorithms for `MPI_Allreduce` use either tree-based methods (e.g., recursive doubling) or ring-based algorithms, each involving multiple communication steps that scale with the number of processes [13]. For example, a ring Allreduce breaks the data into p chunks (for p processes) and requires $2(p-1)$ message hops for each chunk, while a recursive doubling approach completes in $O(\log p)$ steps but with larger messages at each step. At large scale, these communication costs

can substantially degrade performance. To mitigate such overheads, modern HPC interconnects offer features for offloading collectives: for instance, Mellanoxs Scalable Hierarchical Aggregation and Reduction Protocol (SHArP) offloads reduction operations to the network fabric itself. SHArP-enabled switches can aggregate data as it flows through the network, reducing an Allreduces latency by performing the sum in-switch and cutting down communication steps required [14]. In a 128-node system, SHArP demonstrated over 2x speedup for small-message Allreduce by halving the number of network traversals [14]. However, even with advanced networking and collective offload, the fundamental challenge remains the volume of data to be exchanged. This is where data compression can complement MPIs built-in optimizations: by reducing message sizes, compression attacks the bottleneck from a different angle, potentially working synergistically with collective offloading and fast networks.

2.3 SimGrid

SimGrid is an open-source simulation framework for distributed systems, widely used in the parallel and distributed computing research community to model and study the behavior of large-scale distributed applications and platforms. SimGrid has evolved into a robust toolkit that enables researchers to simulate computing grids, clouds, HPC infrastructures, and peer-to-peer systems with a high degree of realism. A key strength of SimGrid is its emphasis on accuracy, scalability, and versatility in simulations. It provides validated models for core aspects of distributed execution including processor performance, network latency/bandwidth, and scheduling allowing users to obtain realistic performance predictions for algorithms under various scenarios. At the same time, SimGrid is designed to scale to thousands of nodes, making it suitable for simulating modern distributed environments. The framework exposes an API that lets users define application behavior and the platform configuration, after which SimGrid executes a synthetic run of the application on the virtual platform. Results can then be collected for analysis, providing insights into how a real deployment might perform. SimGrids popularity in academia stems from its balance of fidelity and ease-of-use, it offers sensible default models for common scenarios, yet it also allows customization or extension of models when more domain-specific accuracy is needed. In summary, SimGrid serves as a foundational tool in distributed systems research, enabling reproducible and controlled experimentation with complex distributed algorithms and applications [15].

2.4 Data Compression Techniques

Data compression refers to encoding information using fewer bits than the original representation. In the context of MPI communication, compression can reduce the amount of data that needs to be sent over the network, at the cost of some computation to perform compression and decompression. There are two broad categories of compression relevant to HPC data, lossless and lossy compression.

Lossless compression algorithms preserve data exactly, ensuring that the original data can be reconstructed bit-for-bit from the compressed form. Classic lossless methods include entropy coding such as Huffman coding and dictionary-based algorithms such as LZ77/LZ78. These methods work well when data has statistical redundancy. General-purpose compressors (e.g., Zlib, LZ4, Zstandard) have been used in HPC contexts to compress communication or I/O data streams. However, scientific data (like floating-point arrays of model parameters or gradients) often have high entropy and less obvious repetition patterns, making them challenging for traditional lossless compressors. Specialized lossless compressors for numeric data have been developed to address this. An example is FPC (Floating Point Compressor), which predicts each floating-point value from the previous one and compresses the unpredictable part using variable-length encoding [16]. FPC exploits the typical coherence between successive numerical values and has been shown to significantly outperform generic compressors on scientific floating-point streams, compressing data with low latency (on the order of nanoseconds per value in CPU implementations) [16]. While lossless compression ensures complete fidelity, the compression ratios achievable on gradient or weight data are often modest on the order of 1.5x or 2x because the data can appear nearly random to the compressor.

2.4.1 LZ4

LZ4 is a lossless compression algorithm based on the LZ77 family, designed with a primary focus on speed. It achieves very high throughput by using a fast hash table lookup for matching and encoding repeated substrings with minimal overhead. The compressed data format consists of sequences of literal bytes followed by back-reference tokens that encode the length of a match and the distance (offset) to a previous occurrence in a sliding window. This simple design allows LZ4 to trade off compression ratio for speed, it typically produces a moderate compression ratio usually in the order of 1.5x to 2.0x but can compress data much faster than traditional algorithms like DEFLATE. Notably, LZ4s decoder is extremely fast since decompression mainly involves copying referenced data from earlier in the stream. These characteristics have made LZ4 popular in systems where real-time compression is needed e.g. in-memory data processing, storage engines, as it provides an acceptable compression ratio while significantly reducing compression and decompression time [17].

2.4.2 Zstd

Zstandard often abbreviated Zstd is a modern general-purpose lossless compressor developed by Facebook, which combines LZ77-style string matching with advanced entropy coding techniques to achieve a balance of high compression ratio and speed. Like many compressors, Zstd uses a sliding window dictionary (LZ77) to find repeated substrings. However, it improves on older schemes by employing Finite State Entropy and Huffman coding in its entropy stage, allowing more efficient bit-level encoding of matches and literals. Zstd is designed to provide compression ratios comparable to or better than algorithms such as DEFLATE but with significantly

faster compression and decompression speeds. It offers a wide range of compression levels (1-22), enabling users to trade off compression time versus ratio. At its default levels, Zstd provides a midpoint trade-off: typically compressing data notably faster than zlib/DEFLATE for a similar or higher compression ratio. Higher levels use larger search windows and more thorough entropy coding to improve ratio at the cost of speed. Because of its versatile performance, Zstd has quickly become adopted in many storage and data processing systems as a drop-in replacement for older algorithms. It delivers fast real-time compression suitable for production workloads, while its decoder remains lightweight and fast due to the efficient coding and block structure [18].

2.4.3 Snappy Compression

Snappy is an LZ77-based compression algorithm developed by Google with the explicit goal of maximizing compression and decompression speeds rather than achieving the highest compression ratio. It forgoes complex entropy coding, which simplifies the format and reduces CPU usage at the expense of compression ratio. Snappy operates with a sliding window and emits sequences of literals and copies much like other LZ family algorithms, but its implementation is tuned for very fast string matching and minimal per-byte overhead. As a result, Snappy can compress data significantly faster than algorithms like zlib, albeit typically yielding lower compression ratios. One notable aspect of Snappy is its asymmetry, decompression is even faster than compression, making it attractive for read-heavy systems. In fact, Snappy was designed so that compressed data can be decoded at high speed with negligible latency, which is ideal for real-time applications. It also features a fixed, stateless compression scheme, simplifying its integration. Due to this design philosophy, Snappy is widely used in big-data frameworks (e.g. Hadoop, Spark, Kafka) and databases to reduce I/O volume without imposing significant CPU overhead. Overall, Snappy provides a pragmatic trade-off, favoring speed and simplicity, and is often chosen when throughput is more critical than maximum compression ratio [19].

2.4.4 Other algorithms

Classic compressors like DEFLATE and LZMA use more complex coding to squeeze out higher ratios, but at much greater cost. Deflate combines LZ77 with Huffman coding, producing smaller output than LZ4/Snappy on average, but its bit-level coding and tree updates make it slower and harder to pipeline. In our context, DEFLATE often cannot saturate hardware links without many cores. LZMA decompresses 2-3x slower than DEFLATE and its compression latency is prohibitive for streaming. This is why high-throughput systems prefer LZ4/Snappy/Zstd, they offer Gb/s real-time processing. In summary, DEFLATE/LZMA win on ratio but lose on throughput/latency. The chosen algorithms LZ4, Zstd, Snappy for our evaluation trade some ratio for orders-of-magnitude faster execution and simpler hardware logic, making them ideal for HPC communication and FPGA implementation[5], [20], [21].

2.4.5 Lossy

Lossy compression algorithms, on the other hand, allow some loss of information in exchange for much higher compression ratios. In HPC and deep learning, error-bounded lossy compressors have gained traction, as they can significantly shrink data size while guaranteeing that the distortion remains below a certain threshold. Two prominent compressors in this category are SZ and ZFP. SZ employs techniques like quantization, linear prediction, and Huffman coding to compress scientific data arrays, often achieving an order of magnitude reduction with errors below a user-specified tolerance [22]. ZFP is another lossy compressor designed for floating-point arrays, which uses a fixed-rate mechanism based on transforming blocks of data into a custom orthogonal basis and then truncating insignificant bits [23]. ZFP can operate in a mode where each block compresses to the same number of bits, useful for consistent throughput. These tools have shown compression ratios of 10x or more on real-world HPC datasets, far beyond what lossless methods can do. The downside is that the reconstructed data is an approximation of the original. In contexts like deep learning, a small amount of error in gradients or model weights may be tolerable, but this must be carefully evaluated to avoid adversely affecting convergence or accuracy.

Another class of compression appropriate to DDL is gradient compression techniques developed in the machine learning community. Rather than general-purpose bit-level compression, these methods reduce communication by compressing the values of gradients or parameters being exchanged [24]. Notable examples include quantization of gradients to lower precision and sparsification. QSGD (Quantized SGD) is one such approach that encodes gradients with low-bit precision stochastically, reducing communication bandwidth at the cost of added noise, while proving convergence under certain conditions [25]. Deep Gradient Compression (DGC) goes further by dropping (thresholding) small gradient values and only communicating the larger, more significant updates, coupled with momentum correction and local accumulation of dropped gradients to preserve convergence [26]. These techniques have demonstrated dramatic bandwidth reductions such as 270x to 600x in distributed training scenarios [26]. They are lossy in the sense that not every update is transmitted with full precision, but by focusing on the most important information, they manage to maintain model accuracy. Gradient compression approaches are typically implemented at the software algorithm level (within deep learning optimizers) rather than as a generic data compression step. Nevertheless, they highlight the opportunity and challenge of lossy compression where extremely high compression is possible if some loss of fidelity is permitted and properly managed.

In summary, a spectrum of compression methods exists for potentially mitigating communication bottlenecks, from fast lossless codecs that can be transparently inserted into MPI messaging, to aggressive lossy schemes that deeply integrate with the application semantics. For our purposes, the focus is on methods that can be applied in-line with MPI communication. This in our case is limited to lossless compression on the fly, but could potentially be extended to include lossy compression that respects application error bounds (for instance, compressing 32-bit floats to 16-bit with bias correction, or other quantization strategies) in future research.

The next section discusses how such compression techniques can be accelerated and integrated via hardware to further alleviate the overhead.

2.5 Hardware Acceleration for MPI Communication

While compression can reduce the volume of data to communicate, performing compression in software consumes computational resources and can introduce extra latency. Hardware acceleration aims to alleviate this by offloading the compression task and potentially other communication-related tasks to dedicated hardware, thus overlapping or eliminating the software overhead. There have been several efforts in this direction.

Software-based and GPU-based Approaches have demonstrated in research that even pure software solutions could improve MPI performance by compressing messages at runtime. As shown in [27] and [28] a modified MPI library performing on-the-fly compression achieved up to 98% and 87% execution time improvement respectively in communication-heavy applications, by drastically cutting message sizes. Their approach, however, also highlighted the trade-off, of the CPU spending additional cycles compressing and decompressing, which can interfere with the applications computation if not overlapped properly. Subsequent work [29] integrated multiple compression algorithms into an MPI library and evaluated them on real HPC applications, finding significant speedups in certain scenarios e.g., a 1.8× boost for a molecular dynamics code when using fast compression like LZ4. These software-based MPI compression frameworks underline both the potential gains and the overheads of compression.

More recently, attention has turned to using accelerators GPUs or programmable NICs to handle compression tasks. Modern HPC clusters often have GPUs primarily for computation, but GPUs can also be leveraged to compress data quickly using their high memory bandwidth and parallelism. For example, researchers have built MPI libraries that offload message compression to GPUs which is particularly beneficial when the data to send is already on the GPU, as in distributed deep learning. As demonstrated in [6] integrating compression into GPU-aware MPI collectives can substantially reduce end-to-end latency and even enable bandwidth-bound applications to scale better. In one design, each GPU compresses its outgoing data before invoking MPI, and the receiving GPU decompresses it, this is all done in parallel with computation. Their results showed improved performance for communication-heavy kernels and suggested that compression can be dynamically toggled on or off depending on message size and network conditions to maximize efficiency [6]. Another example is gZCCL, a framework that adds compression into NVIDIAs NCCL library for collectives, which manages to accelerate Allreduce operations on GPUs by carefully controlling the error from lossy compression so that it does not accumulate across iterations [25], [26]. GPU-offloading of compression, however, competes with the primary GPU tasks when the compression is executed as any other general purpose kernel on the GPU. If the GPU is busy with neural network operations,

dedicating GPU cores to compress communication data may still introduce some delay or contention. This motivates exploring dedicated hardware that can perform compression without stealing cycles from the CPU or GPU.

Smart NICs and In-Network Computing is another avenue of hardware acceleration is moving computation into the network interface or the network switches. Smart NICs (network interface cards with programmable compute, often using System-on-Chip or FPGA technology) enable customization of data processing as it enters or leaves the network. Off-loading compression to a Smart NIC means that as a message is being sent from a node, the NICs hardware can compress it on the fly, and similarly decompress on the receiving side, with minimal involvement from the host CPU. This effectively pipelines communication and compression. In [30] an example of this is presented with CEAZ, a co-designed adaptive lossy compression implemented on an FPGA-based SmartNIC, aimed at speeding up parallel I/O by compressing data as it travels through the NIC. Their compressor achieved over 2x higher throughput than prior FPGA compression designs and demonstrated significant improvements in end-to-end I/O bandwidth in a 128-node testbed, all while the host CPUs were free to continue computation. Although CEAZ targets storage I/O, the same concept can apply to MPI messages, a SmartNIC with an FPGA or ASIC compressor could reduce message sizes without CPU intervention. On the switch side, the aforementioned SHArP technology is an instance of in-network computing (performing reductions in switch ASICs) but does not compress data. One could envision similar in-network compression services in future HPC networks, where switches compress data on congested links to maximize effective bandwidth. Current InfiniBand and high-end Ethernet switches are starting to include programmable packet processing, extending this compression could be incorporated in those in the future.

FPGAs are particularly attractive for this purpose because they can be tailored to implement compression algorithms with deep pipelining and parallelism, achieving high throughput with minimal latency per byte processed. An FPGA-based compressor can be designed to run at line rate, meaning it could compress or decompress data on the fly as it streams through the network interface without introducing significant delay. Unlike CPU, which executes instructions sequentially, an FPGA can have dedicated hardware for each stage of the compression e.g., reading bytes, applying a predictor, coding the output all operating concurrently in a pipeline. This makes it feasible to integrate compression into the network path, for instance, placing an FPGA between the host and network that automatically compresses outgoing MPI messages and decompresses incoming ones. Some recent HPC systems have explored FPGAs for network processing.

2.5.1 NVIDIA Blackwell On-Chip Compression for HPC Communication

NVIDIA's Blackwell GPU architecture introduces dedicated on-chip hardware for data compression tasks most notably a Decompression Engine (DE) built into each GPU. This engine supports popular compression formats e.g. LZ4, Snappy, Deflate in hardware [31]. By offloading compression/decompression from the SM cores to

a specialized unit, Blackwell GPUs can handle data compression concurrently with normal kernel execution, avoiding interference with CUDA kernels. In practice, the Blackwell DE can stream compressed data directly over high-speed links (such as the 900 GB/s NVLink-C2C to an attached Grace CPU) and perform on-the-fly decompression as the data is transferred [31]. This includes fused copy-decompress operations that move data from CPU memory to GPU memory while decompressing in transit [32]. The result is that a GPU can, for example, receive compressed message data and automatically decompress it into HBM memory without burdening the CUDA cores. The main GPU compute kernels can proceed in parallel, with the only additional overhead being on the GPUs memory, since the compression work is handled by the dedicated engine [32].

This hardware-accelerated compression pipeline is particularly beneficial in HPC communication scenarios. Large message transfers can be compressed to reduce network payload, then decompressed by the GPU upon reception all without stalling ongoing computations. Prior to Blackwell, GPU-based compression was typically done via software libraries like NVIDIAs nvCOMP, which run compression kernels on the GPUs SMs [33]. While effective in boosting throughput related work have shown up to 35.7% faster training and 80.9% lower broadcast latency in distributed deep learning when using GPU compression [4], such an approach incurs overheads, the compression kernels compete with the main workload for GPU compute and memory bandwidth [33]. For instance, running an LZ4 or ZFP compression on the GPU consumes some fraction of the FLOPS and also streams data through the already busy memory hierarchy, which can slow down the primary computation. Blackwells on-chip engine eliminates compute-cycle contention by performing compression tasks in a separate pipeline, allowing CUDA cores and tensor units to focus solely on the application kernels [32], [33]. Moreover, by overlapping compression/decompression with communication, the effective latency of MPI transfers can be hidden or reduced e.g. the GPU can decompress incoming data while simultaneously computing on previously received chunks.

2.5.2 FPGA-Based Compression vs. GPU Compression Approaches

While NVIDIAs Blackwell GPUs embed fixed-function compression hardware, FPGA-based compression solutions offer a different set of advantages for HPC communication. FPGAs can be employed as dedicated compression accelerators sitting alongside or in place of GPUs in a cluster. Unlike a GPUs built-in engine which is limited to certain algorithms and available only on NVIDIA platforms, an FPGA can be reconfigured to implement virtually any compression scheme or precision-reduction technique required by the workload. This reconfigurability means HPC designers can tailor the compression logic to their data, for example, an FPGA design could include a custom floating-point compressor optimized for sparse scientific data or a domain-specific lossy compressor for simulation outputs. If a new compression algorithm emerges or if the communication pattern changes, the FPGAs logic can be updated to adapt which is something not possible with fixed GPU architecture. In practice, an FPGA placed on the network path can continuously compress and

decompress MPI messages in hardware, freeing both the CPU and GPU from these tasks. There is no contention with GPU cores or memory at all, since the compression is happening on a separate device dedicated to communication.

Another key strength of FPGAs is runtime adaptability. FPGA designs can be made parameterizable or even partially reconfigurable, allowing the compression behavior to adjust on the fly based on data characteristics or application requirements. For instance, an FPGA-based compressor might monitor the data streams entropy and switch between algorithms dynamically to maximize throughput. All of this can be done while maintaining line-rate processing of network data. By contrast, GPU compression techniques (even with hardware acceleration) are generally tied to a fixed algorithm after execution has started and a fixed hardware capability. In short, the FPGA approach is not limited to one vendors compression IP, it evolves with the state-of-the-art or can be customized to the needs of a specific HPC application.

Perhaps the most compelling advantage of FPGA-based compression is deployment flexibility in heterogeneous or non-GPU environments. In HPC clusters that do not use NVIDIA GPUs for example, those with AMD or Intel GPUs, or even CPU-only supercomputers, NVIDIAs on-chip compression engines are simply not available. FPGAs, however, are vendor-neutral accelerators that can be integrated into any system. They can sit on a PCIe card or as a SmartNIC in each node, compressing outgoing messages and decompressing incoming ones regardless of what CPU and GPU is in the node. This makes them viable for clusters like the Fujitsu Fugaku supercomputer which consists of Arm A64FX CPUs and no GPUs. In such a system, one could attach FPGA-based compression units to the nodes or interconnect. The FPGA on the SmartNIC compresses the data before it leaves the node, reducing network traffic, and the receiving side FPGA decompresses it all transparent to the CPUs. This kind of drop-in integration is possible in any HPC cluster. An FPGA module can work with AMD GPU nodes, Intel GPU nodes, or pure CPU nodes, providing similar benefits of reduced communication volume and latency.

In terms of performance, modern FPGAs can keep up with high-speed networks (100Gb/s and beyond) by leveraging parallelism. For instance, an FPGA design can dedicate many parallel engines for compression to handle large data streams, or even implement the compression across multiple FPGAs in a pipeline for extremely high throughput. A recent study [8] demonstrated that by using FPGAs for collective communication compression, its possible to significantly accelerate multi-node operations, their FPGA-based lossy compression achieved a 2–4× speedup in communication-bound scenarios. Another benefit is that offloading compression to FPGAs removes load from the CPUs, which can then devote more cycles to computation.

To summarize, FPGAs vs. GPU on-chip compression can be seen as a trade-off between integrated convenience and extensible flexibility. NVIDIAs Blackwell provides built-in compression that works out-of-the-box for GPU-to-GPU communications with very low overhead, but it is limited to NVIDIA-centric systems and fixed algorithms. FPGA-based approaches require additional hardware and design effort, but they unlock the ability to deploy compression anywhere in the system (network,

switch, or node), work with any vendors hardware, and evolve the compression strategy over time. In heterogeneous HPC clusters and next-generation supercomputers, FPGA compression engines can serve as a unifying solution to accelerate communication bridging the gap between ultra-fast node computations and comparatively slower network links by dramatically reducing data volume. Dedicated FPGA compression can offer the same level of parallelization as GPUs while avoiding data-movement latency between processing units and can be integrated into any cluster regardless of node architecture [8], [33].

3

Related Work

Research on accelerating MPI communications via compression has gathered pace in recent years, spanning GPUs, FPGAs, and SmartNICs. The driving motivation is that interconnect bandwidth lags behind compute speed, so reducing message sizes with compression can improve overall communication performance [28], [34]. Below, we summarize relevant works in this domain, emphasizing hardware-accelerated approaches to compressing MPI traffic.

Work on GPU-Accelerated Compression for MPI have been presented in [6] where they show the benefits of integrating GPU-based compression into MPI libraries. They modified the MVAPICH2 MPI library to compress messages on the fly in point-to-point transfers which yielded significant bandwidth savings. Subsequent works tackled this overhead by overlapping compression with communication and using faster compression kernels. In another paper it was demonstrated that an optimized GPU compression pipeline could reduce all-to-all latency by 87% in large-message scenarios [34]. They achieved this using a fixed-rate lossy compressor (ZFP) on NVIDIA GPUs, carefully scheduling compression on GPU streams to overlap with network transfer. The same research group further performed work connected to other collectives important in HPC and deep learning, Allgather and Reduce-Scatter [35], and Broadcast with GPU compression [4] for deep learning models. In the Allgather/Reduce-scatter work, they found a 31.7% speed up in DDL training by compressing gradients and model layers on the fly.

Large-scale deep learning training has become a prime beneficiary of compression in communication. Distributed training often spends significant time in MPI collectives like AllReduce to aggregate gradients. Multiple works have proposed lossy, error-bounded compression to accelerate this. We have also seen work on error bound lossy compression to accelerate MPI notably in [2], [9], who developed ZCCL (Zero-loss Compression Collective Library) to incorporate error-bounded lossy compression into MPI collectives. By using an optimized GPU-amenable compressor (fZ-light) and ensuring user-defined error bounds, they achieved 1.9x-8.9x speed-ups in collective communication across real scientific datasets. ZCCLs design emphasizes maintaining numerical fidelity while drastically cutting message sizes, demonstrating that even 8x faster communications are possible with lossy compression in exchange for minimal accuracy loss.

FPGAs have long been explored for accelerating computations, and recent efforts target MPI communication as well. In [8] a system where FPGAs perform on-the-

fly compression of collective communication messages was presented. Their work, investigates like us the potential speed-up of MPI using FPGA-based compression however they analyze lossy compression algorithms. Their work is motivated by the high bandwidth available on modern FPGA boards that often can't be fully utilized with naive data movement. By compressing data before sending between FPGAs, they ensure that both memory and network bandwidth are fully used on. They were able to show a 2.2x increase in total network throughput. As they use of a lossy method they target applications tolerant to a certain degree of error or approximation.

Another hardware acceleration path is using programmable network interfaces e.g. SmartNICs to handle compression tasks, thus offloading work from host CPUs. [36] introduced OmNICCL, which combines in-network computing with compression for sparse MPI AllReduce. In their design, SmartNICs (NVIDIA BlueField-2 DPUs) act as aggregation points that not only combine data from multiple nodes but also apply zero-overhead lossless compression on the fly using Direct Cache Access (DCA) techniques. By compressing zeros and sparseness out of gradients before aggregation, they effectively reduce network load without extra CPU/GPU involvement. OmNICCL delivered impressive results up to 7.24x faster than conventional AllReduce for dense data, and between 1.76x and 17.37x faster than state-of-the-art for sparse AllReduce.

In conclusion, the landscape of compression for MPI in HPC spans multiple hardware platforms. GPUs have demonstrated the ability to reduce communication times for both HPC simulations and AI training by performing compression in parallel with computation. FPGAs provide custom pipelines that can maximize network utilization through tailored compression, as seen in collective operations offloaded to FPGA clusters. SmartNICs/DPUs represent an emerging solution, moving compression into the network layer and freeing host resources. All these approaches share the goal of overcoming communication bottlenecks by trading some compute or precision for less data movement. The cited works confirm that substantial performance gains - often several-fold speed-ups are attainable. This related work provides a strong foundation and justification for the compression strategies discussed in the thesis.

4

Methodology

4.1 System Overview

This chapter begins with a high-level overview of the distributed deep learning (DDL) system and where our FPGA-based compression fits in the communication path. Figure 4.1 shows two representative nodes participating in a data-parallel training job. At each node, outgoing MPI payloads are compressed in hardware before traversing the network and decompressed on ingress, transparently to the application.

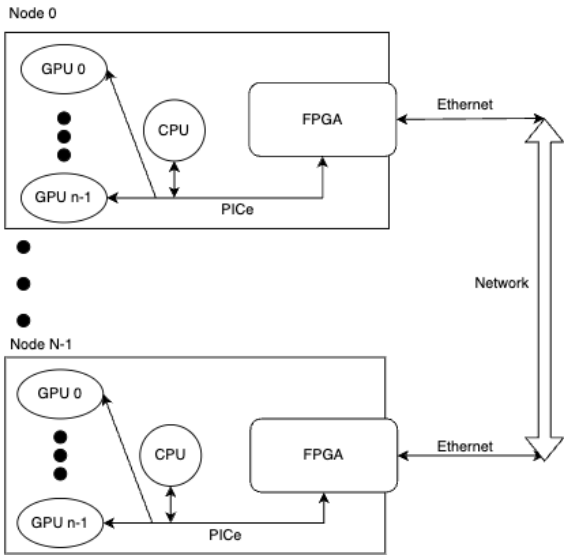

Figure 4.1: Conceptual architecture: FPGA-based compression/decompression modules are placed on the egress/ingress datapaths of each node, compressing MPI payloads before network transmission and restoring them on receipt.

4.1.1 Data Exchange in One Training Step

At a high level, one Stochastic gradient descent (SGD) step proceeds as follows, each process (MPI rank) runs the forward and backward passes and forms gradient buckets of 25 MiB. For a collective such as `MPI_Allreduce`, the bucket is streamed to the FPGA, framed into blocks, and compressed at a measured sustained throughput. The compressed data travels over the network and at the receiver, the FPGA decompresses the stream at at a measured sustained throughput and hands the bytes to

the MPI library. The collective finishes, the optimizer updates the model, and the next step begins. The steps explains the process each node traverses and identifies where compression latency is incurred and where it overlaps with communication.

4.1.2 Hardware Architecture

Figure 4.2 shows the hardware pipeline and how compression/decompression kernels are orchestrated. A packetizer converts host buffers into a byte stream with minimal in-band headers (block length and ID). Short, bounded queues are used to account for burstiness and maintain initiation interval of 1 cycle in steady state. Multiple compression and decompression kernels run in parallel to scale throughput. The egress path mirrors the ingress, a reassembler removes the headers and restores the payload byte-exactly using valid-byte masks.

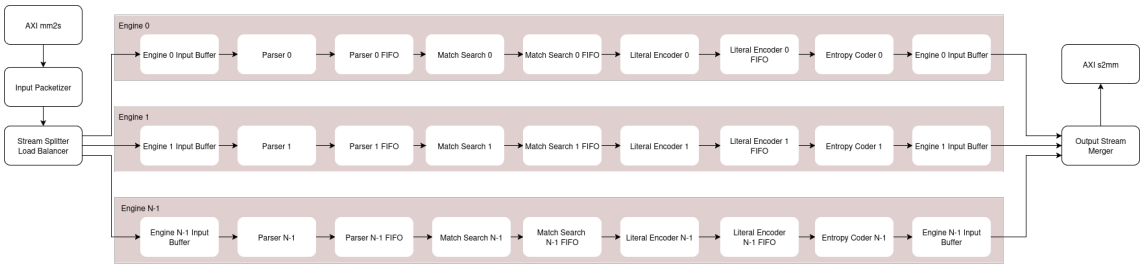

Figure 4.2: Streaming datapath with packetization, bounded queues, parallel compression engines, and symmetric decompression/reassembly on ingress.

4.2 MPI Collective Implementations

The performance impact of compression depends strongly on the collective algorithms used in the MPI runtime. All experiments in this work were conducted using Open MPI 5.0, compiled with default transport modules over TCP. The runtime internally selects among several well-established algorithms depending on message size and communicator configuration, consistent with the implementations described in Section 2.2.

4.3 FPGA Compression Module Development

The first phase of the methodology focused on designing a custom hardware module for real-time message compression on an FPGA. We designed a compressor based on generic IP-blocks provided by Xilinx Accelerated Libraries which we adapt to reduce message size on-the-fly, thereby potentially decreasing network transfer time. The compression algorithm used was selected for its balance of speed and compression ratio (as detailed in Chapter 2) and implemented in high-level synthesis to achieve low-latency operation. We targeted a Xilinx FPGA platform for implementation, using high-level synthesis tools to convert algorithmic C/C++ descriptions into RTL and then optimizing the design. Key design considerations included pipeline

parallelism to process data streams quickly and resource utilization ensuring the compressor fits within FPGA logic and memory constraints. The outcome of this phase was a synthesizable hardware compressor core capable of handling MPI message payloads in real time. The specific implementation details that were introduced to the IP blocks are explained in the following sections.

4.3.1 From memory-mapped to streaming

To support on-the-fly MPI traffic without staging data in external memory, we transform the data path into a continuous byte stream with a small control plane for configuration. Incoming bytes are grouped into blocks of configurable size B by a lightweight packetizer, which attaches a minimal in-band header (e.g., block length and an identifier id) ahead of each block. The stream carries explicit markers to delimit block boundaries and a per-word valid-byte mask so that the final, potentially partial, word in a block is handled precisely. When a codec exposes a native streaming interface, the packetizer connects to it directly; otherwise, a thin adapter bridges buffer-oriented and stream-oriented views so that the compression stage always sees uniform streaming of data. [10]

4.3.2 Pipelining, buffering, and reassembly

The datapath is organized as a concurrent pipeline comprising packetization, compression, decompression, and reassembly. Short, bounded queues between stages decouple burstiness and maintain a steady initiation rate while preventing head-of-line blocking. Queue depths are sized just large enough to absorb the compressors burst patterns without risking stalling. On egress, the reassembler removes the in-band header, restores the original payload bytes exactly, and emits a boundary marker at each block end. The valid-byte mask ensures that payloads not aligned to the data word size are reconstructed byte-accurately.

4.4 Hardware Emulation and Verification

We performed extensive testing via hardware emulation using Xilinx's toolchain. Hardware emulation in the Xilinx Vitis environment creates a cycle-approximate simulation of the FPGA design running on a virtual platform [37]. We emulate the Xilinx Alveo U200 acceleration card, typically used for data center acceleration use cases. This allowed us to verify functional correctness of the compression and decompression pipeline and to gather estimates of performance data in a controlled setting. Emulation executes the actual synthesized RTL of our compressor while modeling surrounding components. Because these models are not fully detailed, the emulated performance is not perfectly accurate. Additionally, the emulation environment runs orders of magnitude slower than real time, forcing us to use smaller message datasets for testing. Xilinx documentation notes that hardware emulation can be up to $10^6\times$ slower than actual FPGA execution and that it uses approximate models for external memory and interconnect, so performance results must be interpreted with a degree of uncertainty [37]. Even though our metrics were measured in emulation we

did verify a subset of our results correctness using an Ultrascale+ MPSoC FPGA. In practice, we hence treated emulation measurements as absolute values as we found negligible differences between emulation and hardware profiling. Furthermore the emulation was invaluable for debugging and ensuring that the integration with the MPI stack worked correctly before running on the physical testbed.

4.5 Performance Evaluation Methodology

After hardware verification of the FPGA-based compression kernel, we conducted a systematic evaluation of its impact on MPI-based communication throughput and application-level performance. The evaluation process comprises two distinct but dependent components, compression performance characterization, and system-level speed-up analysis. These components synergistically give an overview of the full system performance impact.

The compression performance evaluation quantifies the raw throughput and compression ratios achieved by the FPGA kernel. These results are used to parameterize the communication model and guide the configuration of the trace-driven simulations. We measure the maximum achievable throughput in GB/s for both compression and decompression, the compression ratio, and resource utilization on the FPGA (e.g., LUTs, BRAM, logic slices). These values serve as hardware-derived input parameters to the broader system-level model.

In parallel, the system speed-up evaluation assesses the overall improvement in training or MPI collective communication when compression is applied. This includes analyzing changes in communication time during synchronized collective operations and evaluating the end-to-end performance improvement in distributed deep learning workloads. We use a combination of trace-driven simulation using SimGrid, and microbenchmarking to compare execution times with and without hardware compression integrated.

By decoupling hardware-level and system-level evaluation, we can isolate compression-related bottlenecks, estimate upper bounds of acceleration. This methodology enables both a realistic and generalizable assessment of our FPGA-accelerated MPI compression solution.

4.6 Metrics and Experimental Setup

To quantify the benefits and overheads of the proposed compression mechanism, we define core performance metrics, describe the workloads and compression algorithms evaluated, and detail the hardware, interconnect, and simulation environment.

4.6.1 Performance Metrics

The principal metric is end-to-end speedup in distributed computation, specifically training time speedup for deep learning models:

$$S_{\text{speedup}} = \frac{T_{\text{baseline}}}{T_{\text{compressed}}}, \tag{4.1}$$

where T_{baseline} is the execution time with standard MPI communication and $T_{\text{compressed}}$ is the corresponding time with FPGA compression. Values $S_{\text{speedup}} > 1$ indicate a performance gain attributable to communication reduction.

We also report:

- **Compression throughput** (GB/s): sustained compression/decompression rate.
- **Compression ratio** (uncompressed/compressed): factor by which data size is reduced.

Throughput is measured under controlled conditions using datasets from training a language model as explained in 4.6.2. FPGA performance is benchmarked in hardware emulation (Vitis `hw_emu`), collected measured across end-to-end training.

4.6.2 Deep Learning Workload

We collect traces from training a three-layer LSTM language model on the WikiText-2 dataset for 20 epochs using PyTorchs distributed data parallel (DDP) mode. The model consists of an 830-dimensional embedding, two stacked LSTM layers with hidden size 1150, followed by a third LSTM with hidden size 650, and a final dense projection to the 50k-word vocabulary. This configuration, inspired by prior work on AWD-LSTM for language modeling [38], contains approximately 98.8 M parameters and performs ≈114 M floating-point operations per sequence time-step.

Gradients are bucketed (default 25 MiB per bucket), concentrating most communication into large messages. This allows us to study compression effectiveness in a environment where bandwidth is a clear bottleneck. Table 4.1 summarizes the model structure and computational load.

Table 4.1: Specification of the three-layer LSTM model used in our experiments.

Layer	Parameters	FLOPs/step
Embedding	41 714 140	negligible
LSTM1	9 112 600	$\approx 18.2 \times 10^6$
LSTM2	10 584 600	$\approx 21.2 \times 10^6$
LSTM3	4 682 600	$\approx 9.36 \times 10^6$
Output/Dense	32 717 958	$\approx 65.3 \times 10^6$
Total	**98 811 898**	$\approx$**114** $\times 10^6$

4.6.3 Kernel Deployment Strategy

Before conducting system-level simulations, we characterized the FPGA resource utilization and performance of the LZ4, Snappy, and Zstd kernels. These implementations are extended, streaming-capable versions of the AMD Xilinx Vitis Data Compression library, modified to sustain continuous input and output streams for MPI payloads. All designs were synthesized for the Xilinx Alveo U200 accelerator card and verified in hardware emulation mode.

To maximize throughput while respecting FPGA resource limits, we instantiated multiple parallel kernels per device. Because compression is slower than decompression, more compression engines were instantiated to balance the pipeline. For example, for LZ4, each compression engine sustains about 0.22 GB/s, while each decompression engine achieves 1.59 GB/s. We therefore deploy 32 compression and 4 decompression engines, yielding roughly symmetric aggregate throughput. Snappy follows the same configuration, while Zstd achieves balanced performance with 16 compression and 8 decompression kernels. This allocation strategy was chosen at design time based on resource availability and per-engine throughput, not dynamically adjusted at runtime.

4.6.4 MPI Collective Implementations

The performance impact of compression also depends on how collective communication is implemented in the MPI runtime. In this work, we use OpenMPI 5.0 with its default collective algorithms and tuning parameters. This configuration represents a realistic baseline for HPC and distributed training environments, ensuring that the observed effects of hardware-accelerated compression reflect those of a standard, unmodified MPI stack.

4.6.5 FPGA Platform

The hardware target is a Xilinx Alveo U200 accelerator card running at 275-300 MHz. Designs use streaming interfaces with bounded queues sized to prevent stalling. Compression and decompression delays per message are parameterized using throughput and latency values from hardware emulation.

4.6.6 Interconnect

We use a 25 Gbps Ethernet interconnect in the main experiments, representing a realistic configuration for many academic and enterprise clusters. We also run experiments up to 200 Gbps to confirm that observed trends hold at higher link speeds.

4.6.7 Simulation Environment (SimGrid)

The experimental workflow is trace-driven. MPI communication traces are collected by recompiling the MPI collectives (`Allreduce`, `Bcast`, `ReduceScatter`) with lightweight logging. This records message sizes, counts, and timestamps directly during execution, avoiding profiling overheads.

The traces are replayed in SimGrid with a modeled FPGA compression element in the communication path. Each MPI message payload is passed through the Vitis `hw_emu` flow to obtain cycle-approximate compression and decompression delays. The compressor is modeled as a fixed-rate streaming stage with negligible startup latency, consistent with the pipelined architecture.

All simulations and emulation runs were executed on a workstation equipped with an AMD Ryzen 9 7900 CPU, Nvidia GeForce RTX 4070, 64 GB DDR5 memory, and Debian 12 Linux kernel 6.8. This host setup was used consistently for all FPGA emulation and SimGrid trace-driven experiments, ensuring reproducibility. The SimGrid version employed was 4.0, compiled with MPI support, and all software dependencies and build scripts are archived in the projects public repository. [10]

4.6.8 Assumptions and Limitations

- Only application payloads are compressed; MPI headers/metadata are left uncompressed.
- PCIe transfer costs are assumed to overlap with compression in steady state.
- Simulation models throughput as a bounded-rate pipeline; fine-grained PCIe/network contention is not captured.

Our evaluation uses Vitis hardware emulation for the codec stage together with SimGrid for system timing. Hardware emulation is functionally accurate and performance-approximate for the kernel, but it is not cycle-accurate for the full system (e.g., host/PCIe effects). Similarly, SimGrid models network and I/O at an abstracted level to capture trends rather than packet-level timing. In practice, this makes per-block startup latencies for compression and decompression appear very small relative to MiB-scale message airtime. Because these values are not uniquely meaningful in isolation under our modeling assumptions, we do not report them as standalone results. Instead, their effect is reflected in the end-to-end timings used for the speed-up analysis.

4.7 Summary of Methodology

In summary, our research methodology comprised hardware development of a novel MPI message compression engine, its integration into a standard MPI communication pipeline, thorough verification via emulation, and comprehensive performance evaluation through both controlled benchmarks and trace-driven simulations. This multi-faceted approach ensured that we not only built a working prototype, but also gained insights into its behavior in realistic usage scenarios. By combining real hardware experiments with simulation, we accounted for both the measurable improvements on our test platform and the projected benefits in larger-scale deployments. In the next chapter, we present the results obtained from this methodology and discuss the effectiveness and limits of FPGA-based compression for MPI latency reduction. The complete source code, including the FPGA host integration, simulation scripts, and evaluation setup, is available in an open repository [10].

Figure 4.3 summarizes the end-to-end evaluation flow described in this chapter.

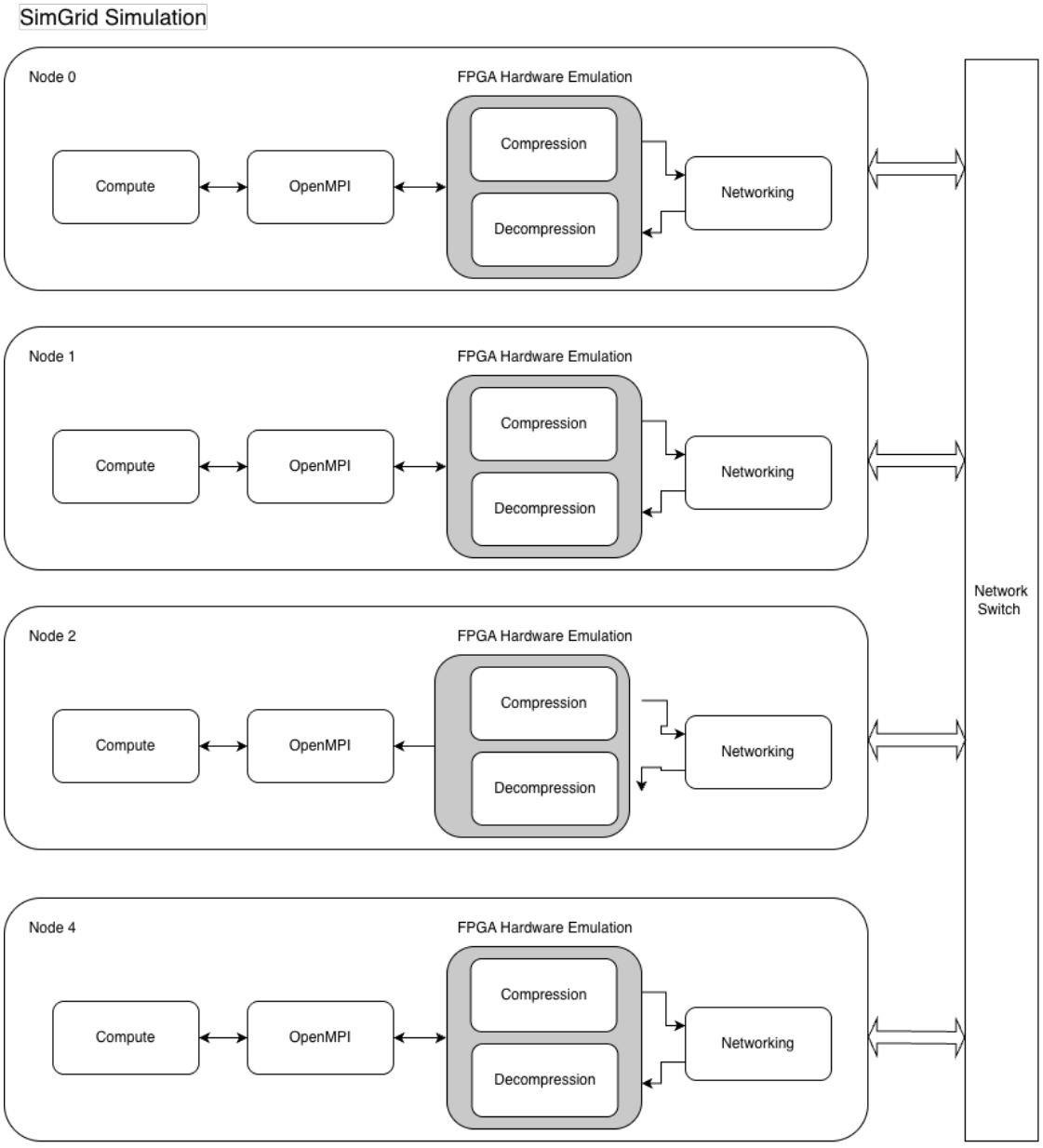

Figure 4.3: Full system simulation workflow. Trace-captured MPI collectives are replayed in SimGrid, which models the network fabric, NIC/PCIe data path, and a streaming pipeline for the codec stage. This combined setup yields per-collective communication times that are aggregated into end-to-end training timings for the speed-up analysis; results represent potential improvements under the stated assumptions.

5

Results

This chapter presents the performance results of the FPGA-emulated compression modules evaluated using real-world communication traces collected from distributed training of the three-layer LSTM language model described in Section 4.6. The evaluation examines whether hardware-accelerated compression can reduce the effective communication time in MPI collectives, thereby improving end-to-end training throughput in bandwidth-limited environments. Rather than increasing the raw network bandwidth, compression improves its effective utilization by transmitting fewer bytes per collective operation.

5.1 FPGA Resource Utilization

The FPGA implementation of each compression algorithm was first characterized in terms of resource usage per compression and decompression kernel. Table 5.1 and 5.2 summarizes the LUTs, BRAM, and URAM consumed by a single compress and decompress engine for LZ4, Snappy, and Zstd. For example, one LZ4 compression core occupies approximately 3K LUTs, 5 BRAM and 6 URAM, whereas an LZ4 decompression core uses only 11K LUTs, 15 BRAM, and 2 URAM, while a Snappy compress core uses 3K LUTs, 4 BRAM, and 6 URAM. The Zstd algorithm is the most resource-intensive overall, in our design, a single Zstd compression kernel used around 11K LUTs, 24 BRAM, and 10 URAM, while a single Zstd decompressor consumed 23K LUTs, 34 BRAM, and 3 URAM. These resource figures guided how many parallel kernels could be instantiated within the FPGA's limits. The relative compression ratios achieved are visualized in Figure 5.1

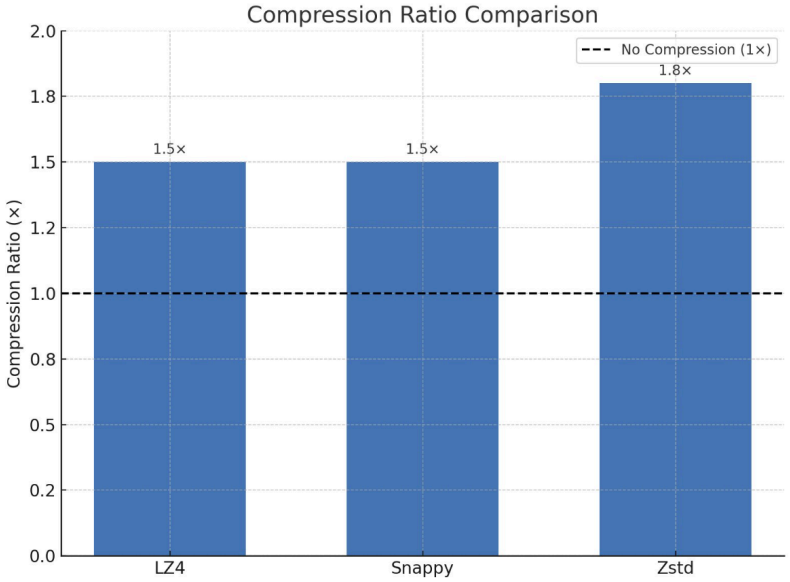

Figure 5.1: Comparison of compression ratios across algorithms. LZ4 and Snappy both achieve approximately 1.5× reduction, while Zstd reaches 1.8×, providing greater savings in transmitted data volume.

Table 5.1: Compression metrics and resource usage per kernel

Algorithm	**Compression Ratio**	**Throughput**	**FMax**	**LUT**	**BRAM**	**URAM**
LZ4	1.53	222 MB/s	300 MHz	3 K	5	6
Snappy	1.51	228 MB/s	300 MHz	3 K	4	6
Zstd	1.84	292.5 MB/s	275 MHz	11 K	24	10

Table 5.2: Decompression metrics and resource usage per kernel

Algorithm	**Throughput**	**FMax**	**LUT**	**BRAM**	**URAM**
LZ4	1.59 GB/s	292 MHz	11 K	15	2
Snappy	1.67 GB/s	300 MHz	12 K	15	2
Zstd	556.81 MB/s	240 MHz	23 K	34	3

Because the compression and decompression throughput per kernel differ, we instantiate different numbers of compression and decompression kernels to balance the pipeline throughput. In particular, we allocate more compression engines for the algorithms due to compression being is slower than decompression, to achieve symmetrical throughput. It should be noted that symmetrical throughput is not guaranteed to yield the best performance and depends on the communication pattern of the system, this is discussed more in the discussion chapter. For LZ4, a single compression engine can only sustain 0.222GB/s, whereas one decompression engine can processes 1.59GB/s. To balance throughput, we instantiate 32 compression engines and 4 decompression engine, matching the aggregate throughput to approximately 1.59,GB/s. This results in a maximum system throughput of 6.36GB/s. For

Snappy, which achieves 0.228GB/s compress and 1.67GB/s decompress throughput, we instantiate 32 compression engines and 4 decompression engine, matching a decompression throughput of approximately 6.68GB/s. Zstd requires more balanced deployment. With 0.2925GB/s compression and 0.55681GB/s decompression, we instantiate 16 compression engines and 8 decompression engine to approximate throughput symmetry yielding 4.45GB/s. The resulting aggregate throughput per algorithm is summarized in Figure 5.2.

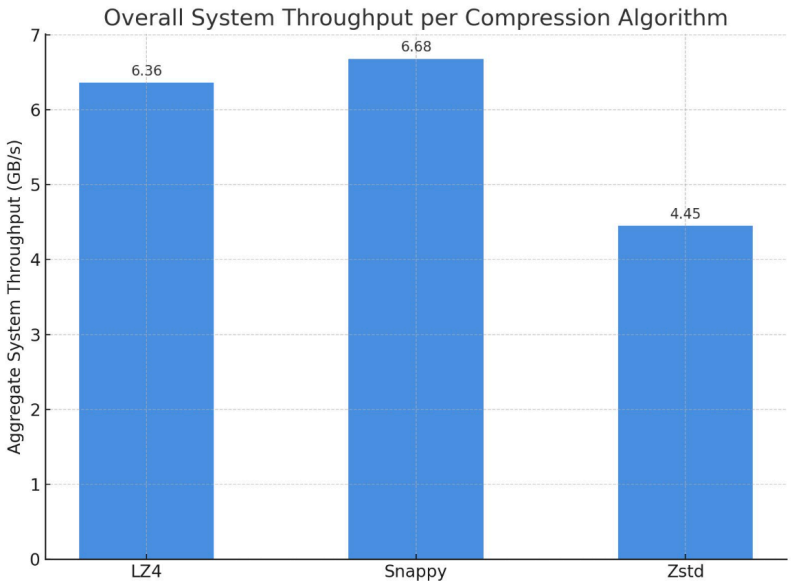

Figure 5.2: Aggregate system throughput per compression algorithm (GB/s) based on FPGA resource-balanced deployments of LZ4, Snappy, and Zstd.

5.2 System Simulations

To evaluate the impact of FPGA-based compression on the computational throughput and scalability of HPC clusters, we conducted a series of distributed training experiments using standard MPI communication. We compare baseline (uncompressed) training to the three algorithm variants where compression is applied to inter-node MPI payloads using LZ4, Snappy, and Zstd. Figures 5.45.7 present timeline traces for each configuration, showing the relationship between compute, communication, and idle phases during the training process. The overall execution time breakdown across algorithms is shown in Figure 5.3.

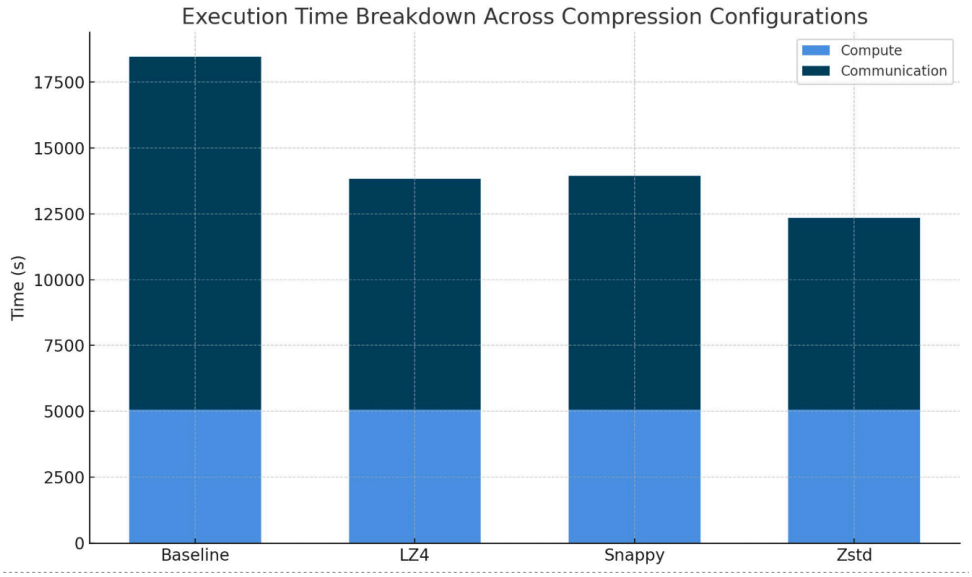

Figure 5.3: Execution time breakdown across compression configurations, showing the contribution of compute and communication time for the baseline, LZ4, Snappy, and Zstd.

The breakdown in Figure 5.3 is shown as separate compute and communication times for clarity. In reality, some overlap may occur in distributed training, but our trace-driven model assumes synchronous iteration boundaries-each step proceeds only after all ranks finish both compute and collective phases. This assumption reflects tightly coupled HPC workloads where synchronization dominates.

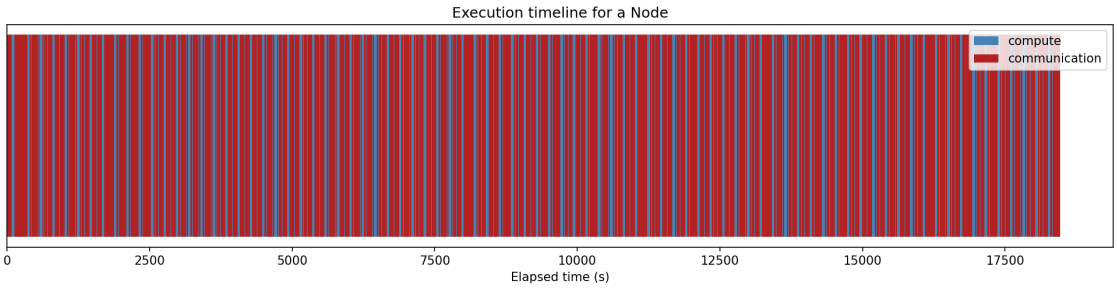

Figure 5.4: Timeline trace of the baseline training. This trace shows the communication latency and data exchange during the baseline training run without any compression.

The baseline trace illustrates the communication behavior of the training process without any compression applied. With a total of 337 iterations across 4 MPI ranks, the total compute time amounts to 5068.29 seconds, while communication dominates the timeline with a cumulative time of 13399.01 seconds. This disparity results in a communication-to-compute ratio of 2.64, emphasizing that communication is the primary bottleneck.

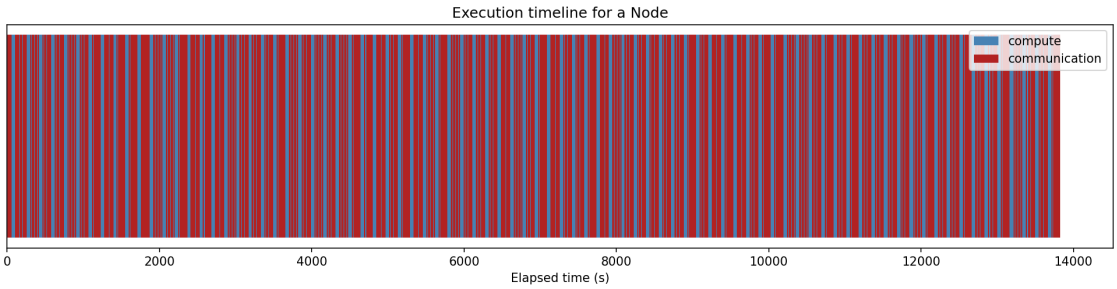

Figure 5.5: Timeline trace of the training using LZ4 compression. This trace shows the communication latency and data exchange after applying LZ4 compression to MPI payloads.

Applying LZ4 compression reduces the total communication time to 8757.52 seconds—an improvement of approximately 35% over the baseline. Notably, the compute time remains unchanged at 5068.29 seconds, confirming that the compression overhead is negligible relative to the compute load. The compression ratio achieved is 1.53×, leading to an overall speed-up of 1.34×. The timeline trace visually reflects this reduction in communication delays, with noticeably shorter idle periods between compute stages, indicating more efficient overlap between computation and data exchange.

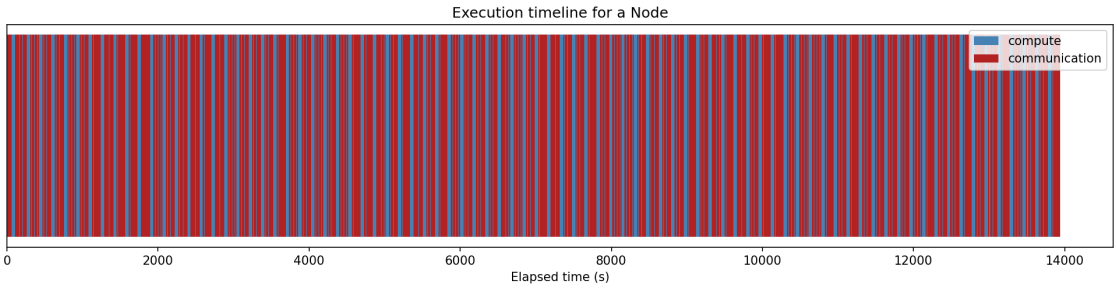

Figure 5.6: Timeline trace of the training using Snappy compression. This trace illustrates the communication latency and data exchange with Snappy compression applied to the MPI payloads.

With Snappy compression, the communication time decreases to 8873.52 seconds, slightly higher than LZ4 but still a 34% reduction relative to the baseline. The compression ratio of 1.51× closely mirrors LZ4s, and the resulting speed-up is 1.32×. As shown in the trace, the structure of computation remains unchanged, but communication intervals shrink modestly. Snappy offers comparable latency reduction to LZ4, with minor differences likely due to algorithm-specific encoding/decoding performance.

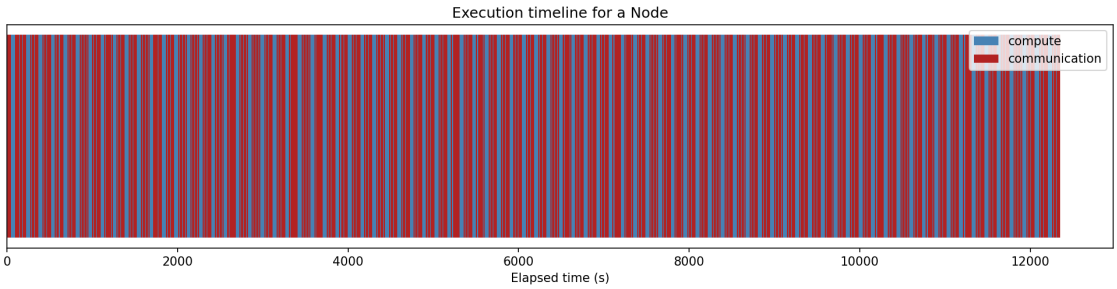

Figure 5.7: Timeline trace of the training using Zstd compression. This trace shows the communication latency and data exchange after applying Zstd compression to the MPI payloads.

Zstd compression demonstrates the highest efficiency among the evaluated algorithms, achieving a compression ratio of 1.84× and reducing communication time to 7282.07 seconds 46% reduction compared to the baseline. This leads to a speed-up of 1.50×, the highest among the compression techniques tested [Figure 5.8]. The trace reveals significantly shortened communication intervals and more compact synchronization phases. Despite Zstd's higher compression overhead, its superior compression ratio yields the most substantial end-to-end latency improvement, making it a strong candidate for environments where minimizing communication time is critical.

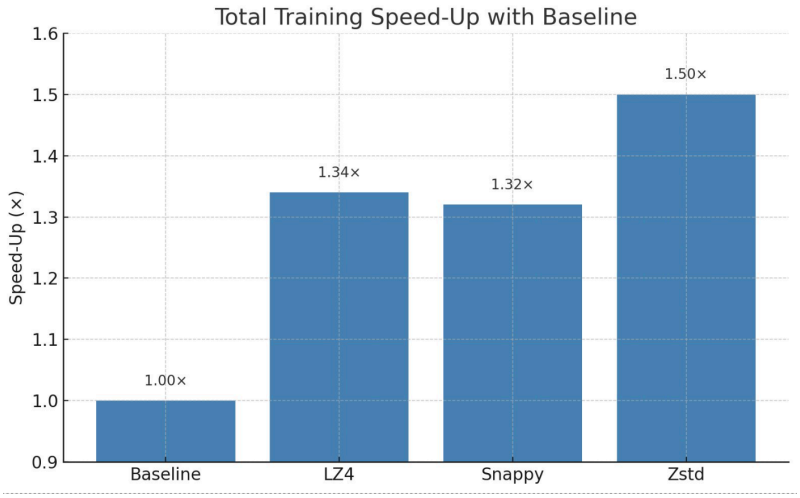

Figure 5.8: Overall training speed-up relative to the baseline across compression algorithms, highlighting Zstds maximum 1.50× improvement.

Although Zstd delivers lower raw compression throughput than very fast codecs like LZ4, it typically achieves a higher compression ratio. When the interconnect is the bottleneck, reducing the number of bytes in flight shortens airtime sufficiently to offset lower compressor bandwidth, improving end-to-end time. This effect is most visible when the link operates near saturation: the network time dominates and the higher ratio directly translates to shorter transfer time. To validate the hypothesis and contextualize our baseline, we include a sensitivity sweep at higher link rates

(25/100/200 Gbps), showing that as the link rate increases and the workload becomes less network-bound, the relative advantage of Zstd narrows while faster-but-lower-ratio codecs become superior.

In summary, FPGA-accelerated compression leads to substantial reductions in communication overhead, with negligible impact on compute time. All three evaluated algorithms LZ4, Snappy, and Zstd improve effective throughput by reducing the volume of data transmitted over the network. While compression does not inherently increase computation-communication overlap, it shortens the communication phase, thereby reducing idle time and improving overall resource utilization. These improvements scale effectively with cluster size, making hardware-based compression a practical and impactful enhancement to the MPI communication stack for HPC workloads.

5.2.1 When to prefer Zstd vs LZ4 vs Snappy

Our results indicate that no single lossless codec is universally optimal. Zstd tends to win for larger, compressible payloads in network-bound settings because its higher ratio reduces airtime. Conversely, for small or latency-sensitive messages, LZ4/Snappy can be preferable: their lower per-byte latency and decoder simplicity reduce end-to-end time despite lower ratio. In addition, we observed that for high-entropy tensors it may be beneficial to bypass compression, since in some cases the added overhead reduces throughput to the point where direct transmission is faster.

6

Conclusion

6.1 RQ1: How can FPGA compression be integrated into the existing MPI communication stack?

This work conceptually incorporated FPGA-based compression into the MPI communication pipeline and evaluated its impact through hardware emulation and trace-driven simulation, rather than a physical integration in a production system. The proposed integration approach envisions modifying the MPI send/receive routines to offload data to an FPGA for compression before transmission, and to decompress data on receipt using a similar hardware unit. In our methodology, this integration was modeled by inserting a compression and decompression stage into the simulated communication path, based on throughput and latency metrics obtained from hardware emulation. The design assumes a streaming interface and pipelined processing to overlap FPGA compression with data transfer. While no physical deployment was performed, the modeling captures key characteristics of such an integration, including latency overhead, kernel parallelism, and throughput asymmetry. This abstraction demonstrates that FPGA-based compression can, in principle, be encapsulated within the MPI transport layer transparently to applications. Accordingly, this evaluation provides strong evidence that such a solution is feasible and beneficial. We can hence conclude that FPGA-based compressors can be inserted into the MPI communication stack to reduce message size and communication latency, with minimal impact on application logic.

6.2 RQ2: How does the proposed solution affect the overall computational throughput and scalability of HPC clusters?

Offloading compression to FPGAs had a positive effect on overall throughput in our experiments, confirming that communication latency can be reduced by transmitting smaller, compressed messages. We observed training speed-ups of up to 1.50x in large-scale deep learning tasks, indicating a substantial reduction in communication time. By reducing message sizes, the network spent less time transferring data,

effectively increasing the systems communication efficiency. This benefit was particularly evident during bandwidth-intensive operations such as all-reduce gradient synchronization, where compression significantly reduced data volume. A notable topic which is discussed in the methodology is that perfectly symmetrical throughput between compression and decompression is not always optimal. While symmetry ensures balanced data flow in theory, the actual effectiveness depends on the communication pattern of the workload. Specifically, if a node predominantly sends data, higher compression throughput is more critical; if it predominantly receives data, then decompression speed becomes the limiting factor. In workloads with balanced send/receive operations, symmetry between compression and decompression becomes more important. In summary, hardware-accelerated compression improved throughput and scaled effectively in our evaluations. However, careful configuration and awareness of workload-specific communication patterns are essential to fully realize its benefits.

6.3 RQ:3 How does FPGA-based compression compare to GPU-assisted methods in terms of latency and performance?

Comparing our results to prior work involving GPU-assisted compression, FPGA-based solutions demonstrate competitive. For instance, in [35], the authors reported a 31.7% increase in training speed when integrating GPU-based compression techniques into their DDL pipeline. Our FPGA implementation surpasses this result across all evaluated algorithms, indicating that dedicated hardware pipelines tailored for communication-bound workloads may provide a more efficient compression-decompression path, particularly in latency-sensitive settings. However our training load which was ran across a 4 node cluster in simulation may have exhibited a different communication pattern that makes a direct comparison meaningless. In order to compare such figures one needs to evaluate both solutions in a equivalent environment performing the same workload.

Furthermore, when looking beyond DDL and into more general-purpose HPC and data-intensive applications, several studies have reported significant gains using GPU-based compression. For example, [27] and [28] demonstrated reductions in overall execution speed-up of up to 98% and 87%, respectively, by incorporating compression into workloads with high communication overhead. While these results are impressive, they often rely on broader algorithmic optimizations and integration with task scheduling and memory hierarchies, which may not be directly comparable to our use case focused solely on MPI-level communication compression in DDL.

In terms of broader trends, GPU-based lossy compression methods have achieved even greater speed-ups, ranging from 1.9x to as high as 8.9x in certain benchmarks. However, it is important to note that such approaches typically involve data approximation, precision loss, or model accuracy trade-offsnone of which are acceptable in our scenario where lossless communication fidelity is required.

The evaluation confirmed that using hardware compression in MPI can significantly reduce communication time for distributed deep learning. We measured notable speed-ups when compressing inter-node messages, which is a substantial improvement in an HPC setting. We also identified that the compression algorithm choice matters. In our experiments, all three codecs provided sustained compressor throughput above the line rate; therefore, end-to-end time was governed primarily by the compression ratio, and the higher-ratio codec Zstd delivered the best results. This does not imply that more aggressive codecs is always better universally the optimal point, it depends on link speed, message-size distribution, and symmetry/asymmetry between compression and decompression. At higher link rates e.g., 100–200Gb/s or for smaller, latency-sensitive messages, compressor/PCIe service time can surface as the bottleneck; in such scenarios, a lighter setting or bypass can outperform a heavier one.

6.4 Threats to Validity

We acknowledge several limitations and risks to validity in this study. Our performance results were obtained using a combination of hardware emulation and trace-driven simulation, rather than a full deployment on a production HPC cluster. While we carefully calibrated our FPGA emulator and network model, this approach can deviate from real-world conditions. For example, the simulation might not capture all sources of overhead such as PCIe transfer delays, cache effects, or interference on a busy network present in a physical cluster. Consequently, the absolute speed-ups reported may be optimistic.

The use of traces from specific deep learning workloads introduces potential bias. We gathered communication traces from representative training runs to model the MPI traffic, but these traces inevitably reflect assumptions about the workloads behavior e.g. message sizes, frequency, and timing. If those assumptions change for instance, with a different neural network model or a different batch size the effectiveness of compression could differ significantly. The trace-based evaluation cannot capture dynamic adaptations that would occur in a live environment. Third, there is a challenge in generalizing our findings beyond the tested workloads. Deep learning gradients and parameter tensors might compress reasonably well due to numeric value distributions or redundancy, but other HPC applications may not see the same benefit.

Our primary simulations use a 25 Gb/s interconnect to expose bandwidth saturation effects. While HPC clusters often offer much faster links, many production and research clusters still operate near or at our testbed speed. We mitigate external validity risks by confirming the overall trend at higher link rates 100 and 200Gb/s see Chapter 5, System Simulations, sensitivity sweep at 25, 100, and 200 Gb/s. Hence, our conclusions generalize to faster networks, with diminishing gains as bandwidth increases.

We evaluate on distributed deep learning data because it is communication-intensive and widely used. The resulting message patterns stress the network and make

compression trade-offs visible. This focus does not preclude applicability elsewhere (e.g., other collective-heavy MPI workloads), but generalization requires measuring the size/entropy profile of the target application and re-running our trace-driven pipeline. We therefore position this work as a generic method validated on ML workloads, with broader applicability to be quantified in future studies.

Prior research has noted that HPC data traffic is often too random for lossless compression to gain traction, and message size is not always the primary bottleneck. Our positive results on deep learning workloads may not directly translate to, say, molecular dynamics or finite element simulations where the data entropy is higher and inherent compressibility is lower. Therefore, the speed-ups we achieved are workload-dependent, and care should be taken before expecting similar gains in all scenarios. Lastly, the prototype nature of our implementation means that certain engineering considerations were not exhaustively addressed and could affect real-world adoption.

6.5 Conclusion

In conclusion, this thesis demonstrated that FPGA-based hardware compression can be a viable solution to reduce MPI communication latency in distributed deep learning. We showed how an FPGA can be integrated into the MPI stack to compress messages transparently, and we quantified the resulting performance benefits. The main contributions of this work are twofold. First, we developed and evaluated a conceptual integration of on-the-fly FPGA compression within a standard MPI communication model, using emulation and trace-driven simulation to assess its compatibility with HPC workloads and its potential to improve communication efficiency.

Figure 4.1 illustrates the conceptual system architecture evaluated in this thesis. Each node is equipped with an FPGA compression module positioned at the ingress and egress of the network interface. These modules handle compression and decompression transparently, enabling reduced data volumes over the interconnect without modifying application-level MPI logic. This design forms the basis of the integration strategy modeled in our simulation and emulation pipeline. Second, we provided an in-depth performance evaluation using state-of-the-art compression algorithms (LZ4, Snappy, Zstd), revealing that our approach can accelerate training by up to 50% and identifying Zstd as the most effective compressor in this context. We also contributed a comparison against GPU-based compression, highlighting the strengths of a dedicated FPGA strategy in terms of consistency and non-interference with computation. However, we recognize the limitations of our study, the results are based on emulation and specific workload traces, and the approachs efficacy may vary for other applications or at larger scales. These limitations delineate avenues for future work, such as deploying the solution on a real HPC cluster, exploring adaptive compression levels for different data types, and evaluating the approach on a broader array of workloads. Despite these caveats, the findings of this thesis validate the core idea that hardware-accelerated compression can alleviate network bottlenecks in distributed training, and they lay a foundation for further research

and development in efficient HPC communication optimization. The measured and simulated performance gains represent potential or upper-bound speed-ups achievable under the modeled assumptions. While the FPGA compression throughput and latency values are derived from real hardware emulation, system-level timing relies on SimGrid models that abstract network and PCIe details. Consequently, the 1.3-1.5x speed-ups should be interpreted as indicative of the attainable improvement once a full prototype is integrated in hardware, rather than as absolute values measured on a deployed cluster.

6.6 Future Work

Looking forward, there are several exciting directions in which this work can be extended and improved. Dynamic Algorithm Switching with Partial Reconfiguration: Although our results suggest limited benefits from switching compression algorithms on the fly, future research could explore scenarios with more diverse data patterns or specialized algorithms where dynamic switching might be beneficial. Using Dynamic Partial Reconfiguration on the FPGA, one could load an alternative compression core optimized for a certain data type or network condition at runtime. Investigating the feasibility of run-time algorithm adaptation for example, switching to a high-compression algorithm during bandwidth-constrained periods, or to a fast lightweight algorithm for latency-sensitive phases would provide deeper insight. Key challenges to address would include minimizing reconfiguration time and developing policies to decide when and what to swap. Even if the average improvement is small, dynamic schemes might prove beneficial in edge cases or future hardware with lower reconfiguration overhead.

Scaling to Multi-Node Clusters and Complex Benchmarks: A critical next step is to evaluate the FPGA compression mechanism in a larger-scale, real-world environment. This involves integrating the compression-enabled MPI into a multi-node HPC cluster or cloud setup and testing with full applications and standard benchmarks. For instance, running popular HPC benchmarks or applications (LINPACK, HPCG, etc.) with and without compression offload would help quantify benefits in practice. Similarly, implementing our compression approach in a distributed deep learning framework for example, using it to compress gradient exchanges in an all-reduce for ResNet-50 training across several nodes would demonstrate its impact on end-to-end training time. Such system-level evaluations would also uncover any issues with network interoperability, reliability under heavy load, or integration challenges with MPI implementations on high-speed networks. By studying multi-node performance and scaling behavior, we can verify that the advantages observed in our controlled tests hold at scale, and identify any bottlenecks or overheads that appear only in large deployments.

Extended Compression Strategies and Data Types: While our work focused on general-purpose lossless compression for typical numeric data, future research might look at custom or domain-specific compression schemes that could further enhance effectiveness. For instance, HPC applications dealing with matrices or scientific data could benefit from compression techniques that exploit domain knowledge such as

floating-point specific compression or error-bounded lossy compression implemented in hardware. Similarly, in DNN training, one could combine our lossless compression offload with existing techniques like quantization or sparsification of gradients for greater effect. Investigating how our FPGA compression module could be extended or tuned for such use-cases and how it impacts overall accuracy or fidelity when lossy methods are involved.

In summary, the foundation laid by this thesis opens up numerous possibilities for enhancing data movement in large-scale computing. Dynamic reconfiguration, larger-scale testing, and co-design with other system components are all natural next steps to push the boundaries of what FPGA-based compression can achieve. By pursuing these future directions, we can further bridge the gap between computation and communication speeds, and contribute to the development of more efficient and scalable high-performance computing infrastructures.

Bibliography

[1] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356176. [Online]. Available: https://doi.org/10.1145/3295500.3356176.

[2] J. Huang, S. Di, X. Yu, *et al.*, *Zccl: Significantly improving collective communication with error-bounded lossy compression*, 2025. arXiv: 2502.18554 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2502.18554.

[3] Q. Zhou, B. Ramesh, A. Shafi, M. Abduljabbar, H. Subramoni, and D. K. Panda, "Accelerating mpi allreduce communication with efficient gpu-based compression schemes on modern gpu clusters," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, Prometeus GmbH, 2024, pp. 1–12.

[4] Q. Zhou, Q. Anthony, A. Shafi, H. Subramoni, and D. K. D. Panda, "Accelerating broadcast communication with gpu compression for deep learning workloads," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, 2022, pp. 22–31.

[5] J. Chen, M. Daverveldt, and Z. Al-Ars, "Fpga acceleration of zstd compression algorithm," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 188–191. DOI: 10.1109/IPDPSW52791.2021.00035.

[6] Q. Zhou, C. Chu, N. S. Kumar, *et al.*, "Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 444–453. DOI: 10.1109/IPDPS49936.2021.00053.

[7] P. Haghi, A. Guo, Q. Xiong, *et al.*, "Fpgas in the network and novel communicator support accelerate mpi collectives," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2020, pp. 1–10.

[8] M. Koibuchi, Y. Ishida, S. Hirasawa, *et al.*, "Lossy Compressed Collective Inter-FPGA Communications," in *Proc. Intl Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*, 2024, pp. 162–172. DOI: 10.1145/3712031.3712328.

[9] J. Huang, S. Di, X. Yu, *et al.*, *Gzccl: Compression-accelerated collective communication framework for gpu clusters*, 2024. arXiv: 2308.05199 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2308.05199.

[10] A. Bourbia, *Master-thesis-reducing-mpi-communication-latency-with-fpga-based-hardware-compression*, 2025. [Online]. Available: https://github.com/EnisBourbia/Master-Thesis-Reducing-MPI-Communication-latency-with-FPGA-Based-Hardware-Compression.

[11] NVIDIA Corporation, *NVLink and NVSwitch*, https://www.nvidia.com/en-us/data-center/nvlink/, Accessed on May 10, 2025, 2014.

[12] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," 2018. arXiv: 1802.09941 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1802.09941.

[13] R. Rabenseifner and J. L. Träff, "More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 36–46, ISBN: 978-3-540-30218-6.

[14] R. L. Graham, D. Bureddy, P. Lui, *et al.*, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016, pp. 1–10. DOI: 10.1109/COMHPC.2016.006.

[15] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Lowering entry barriers to developing custom simulators of distributed applications and platforms with SimGrid," *Parallel Computing*, vol. 123, p. 103 125, 2025. DOI: 10.1016/j.parco.2025.103125.

[16] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," 1, vol. 58, 2009, pp. 18–31. DOI: 10.1109/TC.2008.131.

[17] W. Liu, F. Mei, C. Wang, M. O'Neill, and E. E. Swartzlander Jr., "Data Compression Device Based on Modified LZ4 Algorithm," *IEEE Transactions on Consumer Electronics*, vol. 64, no. 1, pp. 110–117, 2018. DOI: 10.1109/TCE.2018.2810480.

[18] O. Shadura, B. Bockelman, P. Canal, D. Piparo, and Z. Zhang, "ROOT I/O Compression Improvements for HEP Analysis," in *EPJ Web of Conferences (Proc. CHEP 2019)*, vol. 245, 2020, p. 02 017. DOI: 10.1051/epjconf/202024502017.

[19] L. Promberger, R. Schwemmer, and H. Fröning, "Characterization of data compression across cpu platforms and accelerators," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 20, e6465, 2023. DOI: https://doi.org/10.1002/cpe.6465. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6465. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6465.

[20] T. Chen, S. Song, and Z. Wang, "A high-throughput hardware accelerator for lempel-ziv 4 compression algorithm," 2024. arXiv: 2409.12433 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2409.12433.

[21] K. Chasapis, M. F. Dolz, M. Kuhn, and T. Ludwig, "Evaluating power-performace benefits of data compression in hpc storage servers," in *The Fourth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)*, Jan. 2014, pp. 29–34, ISBN: 9781612083322.

[22] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[23] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014. DOI: 10.1109/TVCG.2014.2346458.

[24] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, *Deep gradient compression: Reducing the communication bandwidth for distributed training*, 2020. arXiv: 1712.01887 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1712.01887.

[25] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," 2017. arXiv: 1610.02132 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1610.02132.

[26] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2020. arXiv: 1712.01887 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1712.01887.

[27] J. Ke, M. Burtscher, and E. Speight, "Runtime compression of mpi messanes to improve the performance and scalability of parallel applications," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 2004, pp. 59–59. DOI: 10.1109/SC.2004.52.

[28] Q. Zhou, P. Kousha, Q. Anthony, *et al.*, "Accelerating mpi all-to-all communication with online compression on modern gpu clusters," in *High Performance Computing: 37th International Conference, ISC High Performance 2022, Hamburg, Germany, May 29 June 2, 2022, Proceedings*, Hamburg, Germany: Springer-Verlag, 2022, pp. 3–25, ISBN: 978-3-031-07311-3. DOI: 10.1007/978-3-031-07312-0_1. [Online]. Available: https://doi.org/10.1007/978-3-031-07312-0_1.

[29] B. Dickov, M. Pericàs, G. Houzeaux, N. Navarro, and E. Ayguadé, "Assessing the impact of network compression on molecular dynamics and finite element methods," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012, pp. 588–597. DOI: 10.1109/HPCC.2012.85.

[30] C. Zhang, S. Jin, T. Geng, J. Tian, A. Li, and D. Tao, "Ceaz: Accelerating parallel i/o via hardware-algorithm co-designed adaptive lossy compression," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ACM, Jun. 2022, pp. 1–13. DOI: 10.1145/3524059.3532362. [Online]. Available: http://dx.doi.org/10.1145/3524059.3532362.

[31] NVIDIA Corporation, *NVIDIA Blackwell Architecture Whitepaper (Datasheet)*, https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf, 2024.

[32] NVIDIA Corporation, *Accelerating Lossless GPU Compression with nvCOMP (v4.2)*, https://developer.nvidia.com/nvcomp, Accessed May 15, 2025.

[33] C. Penaranda, C. Reano, and F. Silla, "Hybrid-smash: A heterogeneous cpu-gpu compression library," English, *IEEE Access*, vol. 12, pp. 32 706–32 723,

Mar. 2024, Publisher Copyright: ĺ 2013 IEEE., ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3371253.

[34] J. Huang, S. Di, X. Yu, *et al.*, *An optimized error-controlled mpi collective framework integrated with lossy compression*, 2024. arXiv: 2304.03890 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2304.03890.

[35] Q. Zhou, Q. Anthony, L. Xu, *et al.*, "Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 134–144. DOI: 10.1109/IPDPS54959.2023.00023.

[36] T. Gu, J. Fei, and M. Canini, "Omniccl: Zero-cost sparse allreduce with direct cache access and smartnics," in *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*, ser. NAIC '24, Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 75–83, ISBN: 9798400707131. DOI: 10.1145/3672198.3673804. [Online]. Available: https://doi.org/10.1145/3672198.3673804.

[37] AMD Xilinx, *Vitis Application Acceleration Development User Guide (UG1393)*, Version 2022.2, https://docs.amd.com/v/u/en-US/ug1393-vitis-application-acceleration, AMD Xilinx, San Jose, CA, Nov. 2022.

[38] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," 2017. arXiv: 1708.02182 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1708.02182.