Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning

Jiawei Fei* Chen-Yu Ho* Atal Narayan Sahu Marco Canini Amedeo Sapio†
KAUST †*Intel*

Abstract

Efficient collective communication is crucial to parallel-computing applications such as distributed training of large-scale recommendation systems and natural language processing models. Existing collective communication libraries focus on optimizing operations for dense inputs, resulting in transmissions of many zeros when inputs are sparse. This counters current trends that see increasing data sparsity in large models.

We propose OmniReduce, an efficient streaming aggregation system that exploits sparsity to maximize effective bandwidth use by sending only non-zero data blocks. We demonstrate that this idea is beneficial and accelerates distributed training by up to 8.2×. Even at 100 Gbps, we assess that OmniReduce delivers 1.2–2.6× better performance for network-bottlenecked DNNs.

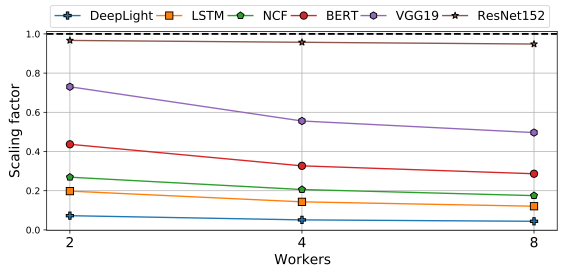


Figure 1: Scalability of six DDL workloads (cf. Table 1) as the number of workers increases. The y-axis shows the scaling factor defined as in [58]: $\frac{T_N}{NT}$ where T is the single GPU throughput and T_N is the measured throughput for a cluster with N workers. Linear scalability requires the scaling factor to be 1 for any N. The experimental setup is in §6.

1 Introduction

Collective communication routines (or simply, collectives) are a core building block of parallel-computing applications. Collectives are commonly used to combine data among multiple processes performing operations in parallel. Achieving high-performance collective communication is paramount in virtually every scenario where an unfavorable computation to communication ratio restricts the ability to scale the workload efficiently.

One such scenario – also the focus of this paper – is distributed deep learning (DDL), which is now in widespread use to reduce the training time of large deep neural networks (DNNs) by parallelizing training over a large number of GPUs. The most common DDL approach is data-parallel training via stochastic gradient descent (SGD) [29]. Distributed SGD is a parallel, iterative workload with two steps: (1) every worker trains a local copy of the model by processing in parallel a different subset (mini-batch) of the training data; (2) all workers combine the results of their computation (local gradient) to produce an average gradient that is applied to the model, prior to the next iteration.[1]

When distributing SGD on many workers, we can either keep the per-iteration total mini-batch constant (strong scaling) or linearly increase the mini-batch size with the number of workers (weak scaling). In the former case, the computation time decreases while the per-worker gradient size stays constant; therefore training easily becomes communication-bound. In the latter case, the computation to communication ratio (ideally) remains constant. However, in reality, communication time increases with the number of workers [31, 42, 54]. Moreover, a large mini-batch size can degrade training quality [27]. To enable better scaling, we aim to *decrease communication overheads by optimizing collective communication*. Figure 1 shows that these overheads are substantial in many DNN workloads, especially for large models where there exists a significant gap between the

*Equal contribution.

[1] Other DDL approaches are model-parallel, pipeline-parallel, and asynchronous data-parallel. These are not as common and out of scope.

measured performance and ideal linear scaling.

Moreover, the size of new DNN models is increasing at a faster pace than hardware compute capacity [22]. Therefore efficient communication is becoming even more crucial. For instance, in a short span of 2.5 years, model size has grown by over $1,000\times$ from $\sim$100M weights in 2018 for ELMo [43], GPT, BERT to $\sim$100B for OpenAI GPT-3 [6] (May 2020). In contrast, the current best-in-class NVIDIA A100 GPU (May 2020) is advertised [40] as up to $10\times$ and $20\times$ faster for floating-point and mixed-precision calculations, respectively, than its V100 predecessor, released in Dec. 2017.

Indeed, the fact that communication is the main performance bottleneck in DDL is well-known [28], and many works [10, 31, 33, 48, 49, 55] proposed various optimizations to achieve high-bandwidth collective communication specialized for DDL. Besides, a recent body of work, primarily within the ML community, developed gradient compression methods [1, 2, 36, 53, 56] to reduce communication time by decreasing the amount of communication, albeit at the cost of reduced training quality due to the lossy nature of compression.

However, these works have failed to observe that, along with the fast-paced increase in model size, gradient sparsity (i.e., the proportion of zero elements in the gradient vector) follows a similar trend. Table 1 shows that gradient sparsity exceeds 94% for the two largest DNN workloads in our study. In fact, sparse gradient vectors (i.e., with sparsity above 50%) are typical for DNNs with a large proportion of embedding weights.[2] This characteristic spans a broad range of DL tasks.

Most existing collective libraries – including DDL-specialized ones like NCCL [39] and Gloo [18] – have no native support for sparse data. These libraries assume dense input data and make inefficient use of precious network bandwidth to transmit large volumes of zeros. This is also a limitation for gradient compression methods because their implementations generally first gather the sparse data into a dense-like format (which has overheads) before invoking a collective routine [57] (§2).

Our key innovation is the design of efficient collective operations for sparse data. We present OmniReduce, a streaming aggregation system designed to maximize the efficient use of bandwidth, and serve as a drop-in replacement for the traditional collective libraries. OmniReduce exploits the sparsity of input data to reduce the amount of communication. As shown in the last column of Table 1, OmniReduce moves up to two orders of magnitude less data by leveraging an aggregator component that determines the non-zero data at each worker in a streaming look-ahead fashion. OmniReduce splits input data into blocks where a block is either a split of contiguous values within an input vector in a dense format or a list of key-value pairs representing non-zero values. OmniReduce achieves high performance through fine-grained parallelization across blocks and pipelining to saturate network bandwidth. OmniReduce leverages fine-grained control of the network to design a self-clocked, bandwidth-optimal protocol and an application-aware failure recovery mechanism to recover from packet losses. The block-oriented approach, fine-grained parallelism and built-in flow control afford us the opportunity to implement the aggregator in-network using modern programmable switching ASICs.

[2]Embedding layers are used to process high-dimensional and typically sparse data. Typically, updates to embedding weights are sparse as only a few embedding vectors from a huge dictionary are used in one batch, and only these vectors have non-zero gradients in the batch.

OmniReduce achieves the following goals:

- **High performance and scalability.** Algorithmically, computational and space complexity do not depend on the number of nodes, while aggregation latency is masked with pipelining. This allows OmniReduce to scale better than previous approaches fundamentally.
- **Data-format universality.** The acceleration is proportional to the sparsity of input data. At the same time, OmniReduce does not require data to be sparse to provide benefits. In the limit, when data is dense, OmniReduce is comparable to bandwidth-optimal dense AllReduce.
- **Flexibility.** OmniReduce's streaming aggregation algorithms admit a variety of instantiations. Sparse input data can be in a block-based dense or sparse (key-value) format without requiring a new API. The aggregator component can run on dedicated server resources (cheaper than worker nodes equipped with GPUs), can run co-located on worker nodes, or with the aid of network switches, as an in-network aggregation component similarly to Mellanox SHARP [49] and SwitchML [48].

To the best of our knowledge, OmniReduce is the first system that realizes all of the above goals at once. SparCML [45] is a collective library for sparse data; however, it requires very high sparsity to achieve performance benefits over dense AllReduce (their results, which we confirm (§6) show benefits when sparsity $> 94\%$). Parallax [30] is a parameter-server architecture specialized for sparse data but requires runtime profiling. Unlike OmniReduce, both of these approaches require input data in the sparse format. We believe that OmniReduce is a general approach and could benefit other applications like data-parallel analytics and sparse matrix multiplication.

We make the following contributions:

- We present the design (§3) and implementation (§5) of OmniReduce, an efficient streaming aggregation system for sparse-native collective communication. Via performance modeling, we demonstrate the theoretical advantages of OmniReduce over standard approaches.
- We introduce block gradient sparsification (§4), a gradient compression technique that works by sampling gra-

Model	Task	Dataset	Batch size	Dense weights	Embedding weights	Gradient sparsity	OmniReduce comm.
DeepLight [13]	Click-through Rate Prediction	Criteo 1TB [11]	2^{11}	1.8 MB	2.26 GB	99.73% (50 epochs)	**16 MB (0.7%)**
LSTM [25]	Language Modeling	GBW [8]	128	74 MB	1.52 GB	94.50% (50 epochs)	**90 MB (5.5%)**
BERT [14]	Question Answering	SQuAD [44]	4	1 GB	284 MB	9.31% (1 epoch *)	**1.13 GB (88%)**
NCF [21]	Recommendation	ML-20mx4x16 [19]	2^{20}	0.4 MB	679 MB	84.6% (30 epochs)	**280 MB (41%)**
VGG19 [50]	Image Classification	ImageNet-1K [46]	64	548 MB	–	32.0% (1 epoch *)	**547 MB (100%)**
ResNet152 [20]	Image Classification	ImageNet-1K [46]	64	230 MB	–	21.6% (1 epoch *)	**230 MB (100%)**

Table 1: Characteristics of benchmark DNN workloads. The table separates model size as dense and embedding weights (which are the weights in embedding layers of a DNN). The gradient has the same size as the model and the table lists its sparsity averaged over a longitudinal analysis of several epochs (* refers to a pre-trained model). The last column details the average per-worker communication by using OmniReduce shown as volume (and % of the otherwise dense communication).

dients' blocks of contiguous elements. We prove convergence and demonstrate empirically that our block-based sparsification techniques can sparsify data to obtain training speedup with negligible degradation in model performance.

• We quantify the performance benefits of OmniReduce using six popular DNN workloads (§6). In end-to-end settings, OmniReduce speeds up training throughput by up to 8.2× at 10 Gbps, 3.5× at 25 Gbps compared to standard ring AllReduce. We quantify the expected speedup at 100 Gbps as being up to 2.6×. We also use benchmarks to compare to state-of-the-art solutions and show that OmniReduce outperforms them by 1.3–5.0×.

2 Background

Collective communication routines. The Message Passing Interface (MPI) [16] standard defines a set of communication protocols for point-to-point and collective routines. In DDL, three collectives are typically used:

• *Broadcast* distributes data from one process to all other processes. This is often used to sync model state among workers, e.g., when reading from a model checkpoint.

• *AllReduce* combines data collected from all processes into a global result by a chosen operator. AllReduce is the most frequently used collective operation in DDL workloads to aggregate gradients by summation.

• *AllGather* collects data from all processes and stores the collected data on all processes. AllGather is useful when the reduction operation is not an associative, point-wise operation. Some gradient compressors use it [57].

We refer to the datatype of collectives' input and output data as a tensor (i.e., a multi-dimensional matrix). Let n be the input size, and c_v be the number of bytes needed to represent a non-zero input value. m is the number of non-zero values.

Tensor data format. The elements of a *dense tensor* are consecutively stored like an array in memory. It is often beneficial or sometimes necessary (due to insufficient memory capacity) to use specialized data structures to store *sparse tensors*. For example, coordinate lists (COO) store a list of non-zero values and a list of the corresponding indices. Dictionary of keys (DOK) stores a dictionary that maps indices to non-zero elements. Although some ML toolkits support sparse tensors (typically in COO format), state-of-the-art collective libraries like NCCL and Gloo operate only with dense tensors (even though the underlying data may be sparse).

2.1 Related work

A strawman solution to collective communication with sparse tensors is AllGather-based sparse AllReduce (AGsparse), as implemented by PyTorch. AGsparse invokes AllGather twice to collect the values and indices of a sparse tensor and then makes a local reduction at every process. Because AllGather needs to allocate an intermediate buffer proportional to the number of processes, AGsparse increments the memory footprint despite sparse data. Further, AGsparse has poor scalability (analyzed in §3.4) as it implicitly assumes no overlap of non-zero indices and is viable iff $m \leq \rho = \frac{nc_v}{c_i + c_v}$, where c_i is the number of bytes needed to store an index (i.e., sparsity above 50% assuming $c_v = c_i$).

SparCML [45] is a set of collectives for arbitrary sparse input data designed for DDL. SparCML uses a latency-bandwidth cost model to characterize different cases and trade-offs between small vs. large messages and whether the output remains sparse or becomes dense (adapting to $m > \rho$), delineating two scenarios: static and dynamic sparse AllReduce (SSAR/DSAR). Data representation in SSAR is always in the sparse format. When the amount of data is small, latency dominates the bandwidth term, thus a latency-optimal recursive doubling algorithm is used. With large data, SSAR_Split_allgather is a two-phase algorithm that optimizes AGsparse by (1) splitting the input into N partitions, one per process, each processed via an AGsparse-like approach to gather data at each designated process and (2) a gathering phase that uses a concatenating AllGather to collect reduced sparse data at all processes. In DASR, DSAR_Split_allgather starts with sparse representation and switches to dense representation during the reduction operation once the condition $m > \rho$ is detected.

In AGsparse and SparCML, communication and reduction occur separately and serially. Instead, OmniReduce performs communication and reduction in parallel by streaming data via the aggregator. This enables OmniReduce to make full use of network bandwidth, while bandwidth is wasted when conducting local reductions in AGsparse and SparCML. OmniReduce supports dense inputs without the format conversion overheads paid by AGsparse and SparCML (§6.1). OmniReduce can reduce communication volume by adopting a block-based format because it does not need to transfer indices.

Parallax [30] devises a hybrid DDL system that has a runtime sparsity monitor and uses a cost model to partition the model weights between a parameter server (PS) architecture for sparse data and traditional AllReduce for dense data. OmniReduce neither requires prior knowledge nor introduces runtime profiling.

Gradient compression. Orthogonal to efficient sparse collective communication, a recent body of work proposes to reduce the amount of communication via gradient compression. As there is a vast literature on the topic, we refer to a recent survey [57] for a more comprehensive discussion. And while techniques abound [1, 2, 36, 53, 56], we distinguish two main approaches: *sparsification* – which sends a subset of elements – and *quantization* – which reduces the element bit-width. Gradient compression is typically lossy and, as a result, can impact the resulted model quality; however, the drop in accuracy is usually small, and one can regulate the compression level to navigate the trade-off. These techniques and OmniReduce are complementary: on the one hand, gradient compression helps to sparsify data in a principled manner; on the other hand, OmniReduce accelerates collective communication of sparse data (allowing for a less aggressive compression level for a given communication budget). We defer other related work to §8.

3 OmniReduce Design

To minimize AllReduce latency, the core idea of OmniReduce is to partition an input tensor G into blocks of tensor elements and transmit only non-zero blocks (i.e., blocks with at least one non-zero value). OmniReduce consists of *worker* and *aggregator* components. The aggregator coordinates workers, instructing them which block to send next. For scalability, the aggregator executes over one or more nodes; in the latter case, each node owns a disjoint shard of blocks. Each aggregator node has a pool of slots, and each slot aggregates a block-sized set of tensor elements. Workers are responsible for detecting and sending non-zero blocks.

Depending on the application, the block format is either dense (i.e., a contiguous subset of G) or sparse (i.e., a list of key-value pairs). We first consider the dense format and then generalize it to the sparse format.

OmniReduce fundamentally improves AllReduce performance thanks to the following two design principles:

• **Fine-grained parallelism and data pipelining.** With each block being independent of any other block, aggregation can be easily parallelized. This enables tightly coupled workers to stream data as a form of a latency-masking pipeline to saturate the aggregator's processing rate, which also serves as a flow control function, yielding a self-clocked protocol similar to other streaming aggregation approaches [48, 49].

• **Coordinated aggregation.** Coordination is key to sending only the non-zero data. The aggregator globally determines the positions of non-zero values among workers in a look-ahead fashion based on the next position metadata efficiently available at the workers (which communicate it to the aggregator). This component differentiates OmniReduce from any related work.

3.1 Basic solution

We first introduce the more straightforward scenario of a lossless network with guaranteed packet delivery. We relax this assumption in the next section. Figure 2 illustrates this scenario with an example.

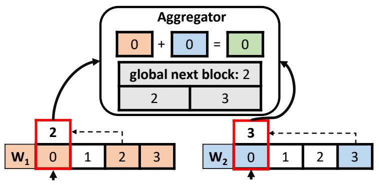
Algorithm 1 illustrates the basic OmniReduce algorithm for dense tensors. A dense tensor consists of a list of values partitioned into blocks. Every block has a size of *bs*. For ease of description, and without loss of generality, we assume that the tensor size is a multiple of *bs*, and the pool size is 1 (i.e., the aggregator has a single slot). We assume the reduction operation is sum ($+$). Other commutative reduction operations are analogous.

What the Worker does: Every worker initially sets *next* as the offset of the next non-zero block after the first block and records it locally. The worker then sends a packet p containing the first block and *next* (Figure 2a).

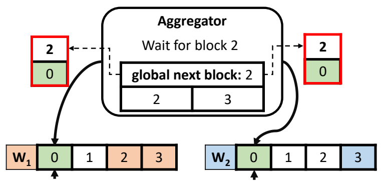
Then, each worker enters a loop where it awaits the aggregator's response. Upon receiving a packet, the worker obtains: (1) the aggregated block data (*p.data*) along with its respective number (*p.block*), and (2) the next block requested (*p.next*) by the aggregator (Figure 2b).

The worker stores the aggregated block into the local tensor G; then, the worker checks whether its next non-zero block corresponds to the aggregator's request. If so, the worker updates *next* with the subsequent non-zero block and sends the requested block to the aggregator (W_1 in Figure 2c). Otherwise the worker awaits a further packet (W_2 in Figure 2c). The loop repeats (Figure 2c-2e) and ends once the aggregator signals that reduction is complete by requesting ∞ as the *next* block (Figure 2f).

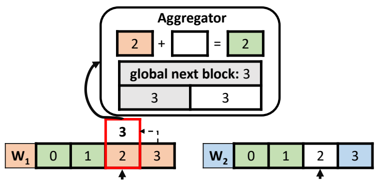
What the Aggregator does: The aggregator does not only aggregate blocks but also keeps track of each worker's next non-zero block. This state is updated whenever a worker sends a packet and enables the ag-



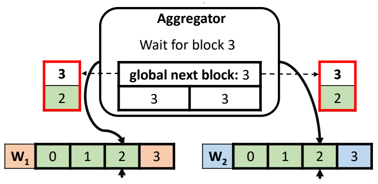
(a) W_1 and W_2 send first block (0).



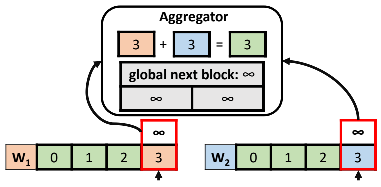
(b) Agg. returns block 0 and asks block 2.



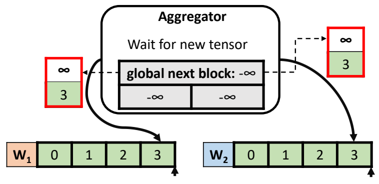
(c) Only W_1 sends non-zero block 2.



(d) Agg. returns block 2 and asks block 3.



(e) W_1 and W_2 send non-zero block 3.



(f) Agg. returns and signals the end (∞).

Figure 2: After every worker sends its first block, the aggregator maintains a view of the global next block necessary for aggregation. Workers only transmit non-zero blocks when the block number matches the requested global next block and inform the aggregator of their next non-zero block. Cumulatively, this ensures that aggregation completes once all non-zero block are transmitted; zero blocks are not transmitted. Legends: 🟧 and 🟦 are non-zero blocks; 🟩 are aggregated blocks; ☐ are zero blocks.

gregator to know the global next non-zero block number. This information is piggybacked into a packet that the aggregator multicasts to the worker with the aggregated data once it determines that a slot is complete. To determine so, the aggregator compares the packet's block number ($p.block$) with the minimum of next non-zero blocks across all workers $\min(next)$. Note that $next$ is already updated to reflect $p.next$. If $p.block$ is less than $\min(next)$, then the current packet is the last one for this slot. The aggregator then crafts a response packet for $p.block$ with the aggregated data in $slot$, resets the state ($next$ and $slot$), and multicasts that packet to the workers.

Fine-grained parallelism. It is easy to observe that the above aggregation logic, while tightly coupling workers at a particular slot, can be parallelized across slots. In the limit, each slot is an independent unit of aggregation. In practice, available network bandwidth limits the number of slots that can be addressed in parallel before the aggregator responds.

OmniReduce exploits this kind of fine-grained parallelism to achieve a form of pipelining that improves performance. The aggregator maintains a *pool of slots* addressable by an index (carried in each packet). Workers apply the logic of Algorithm 1 for S independent aggregation streams (or threads), each of which addresses a separate slot while proceeding at the same rate. As packets are serialized over the network, this architecture can be viewed as pipeline-based processing of one slot per time unit. The available network bandwidth dictates the pipeline depth that is necessary to avoid processing stalls. We present this complete architecture with the following algorithm that also addresses packet loss recovery.

How sparsity affects performance: Since workers only send non-zero blocks, a crucial performance factor is the tensor's block sparsity, which is the proportion of all-zero blocks in the tensor. In turn, block sparsity is determined by not only the tensor itself but also the block size. In general, smaller block size increases block sparsity, but it also decreases bandwidth utilization efficiency due to packets carrying a smaller payload.

We analyze the effects of block size on the performance of OmniReduce in the evaluation section. Therein we empirically find (§6.3) that a block size of 256 elements is the best choice in our setting.

Another factor influencing the performance of OmniReduce in practice might be the cost of finding the next non-zero block. But we find that checking all values in one block has a negligible overhead when this operation is done in parallel on the GPU, as we implement it (§5).

3.2 Packet loss recovery

We now extend our design to support packet retransmission to account for lossy network environments. First, we revisit Algorithm 1 and see how it would fail in the presence of packet loss. A packet loss in the upward path from worker to aggregator prevents the aggregator from completing block aggregation. Whereas, the loss of one of the result packets sent to the workers on the downward path (aggregator to worker) not only keeps a worker from obtaining the aggregated block, but may also stop the worker from sending the next block, and halt the entire aggregation.

To tolerate packet loss, we include acknowledgment packets and use a timer mechanism to detect losses. Further, the aggregator keeps two versions of its per-slot state. The revised algorithm is listed in Algorithm 2. Note that this description includes the pool of S slots,

Algorithm 1: OmniReduce block aggregation

```
1  At Worker:
2      $p.next, next \leftarrow$ index of first non-zero block past block 0
3      $p.block \leftarrow 0$
4      $p.wid \leftarrow$ worker ID
5      $p.data \leftarrow G[0 : bs]$
6      send $p$ to $agg$
7      repeat upon receive p(data, block, next, wid)
8          $G[p.block : p.block + bs] \leftarrow p.data$
9          if $p.next = next$ then
10             $p.data \leftarrow G[next : next + bs]$
11             $p.block \leftarrow next$
12             $p.next, next \leftarrow$ next non-zero block index or else $\infty$
13             $p.wid \leftarrow$ worker ID
14             send $p$ to $agg$
15     until $p.next = \infty$
16 At Aggregator:
17     $slot[bs] := \{0\}$
18     $next[N] := \{-\infty\}$
19     forever upon receive p(data, block, next, wid)
20         $slot \leftarrow slot + p.data$ // reduction operation
21         $next[p.wid] \leftarrow p.next$
22         if $p.block < \min(next)$ then
23             $p.data \leftarrow slot$
24             $p.next \leftarrow \min(next)$
25             $slot[bs] := \{0\}$
26             if $\min(next) = \infty$ then $next[N] := \{-\infty\}$
27             send $p$ to all workers
```

one per stream used by independent worker threads.

Every time the worker receives a packet, it responds to the aggregator for the requested block. But when the aggregator requests a block that the worker would not send (a zero block), the worker only sends an ack. packet with no payload. The worker associates a timer to every transmitted packet; if the timer fires, the worker assumes packet loss and retransmits it. The aggregator has a *count* of aggregated packets; a result packet is sent only once the *count* reaches the number of workers N. To avoid (incorrectly) aggregating duplicate transmissions, the aggregator maintains a boolean vector *seen* that tracks which worker's packet has been processed.

Put together, the approach above ensures that single-sided timers are sufficient to recover from packet loss, regardless of whether a loss occurs on the upward or downward path. However, the aggregator must be able to retransmit a dropped result packet to worker i even after a different worker j has already sent its next non-zero block addressing the same slot. This requires two versions of each *slot* that are used in alternate phases. When the worker receives the resulting packet from the aggregator, it changes the *slot* version by flipping *p.ver* before sending the next block to the aggregator. Each version of a *slot* gets reused only when it is certain that all workers have received the aggregated result in that *slot*. This happens when all workers have sent their blocks to the other version of that *slot*, signaling that all workers have moved forward.

3.3 Extension to sparse block format

OmniReduce's block aggregation approach generalizes to sparse tensors (e.g., in COO format). We briefly discuss this extension, which is listed in Algorithm 3 (for ease of presentation, we do not show stream parallelism and packet loss recovery). In this case, the input tensor is the pair K, V where K is the list of keys (or indices) and V is the list of the corresponding values. The worker sends a packet with a block of *bs* key-value pairs along with the *nextkey* to indicate the key of the next non-zero value. The aggregator keeps track of *nextkey* for every worker, attaching the minimum next key it needs to receive from any worker when sending back a result packet. Only when a worker receives a *p.nextkey* matching its next non-zero value will it send another block to the aggregator. The aggregator internally uses a hashtable or a similar keyed-memory abstraction to carry out aggregation based on key-value pairs.

While we present the above approach for completeness, we do not investigate its practical realization and leave that to future work. This is because our real-world applications only use dense tensors, to begin with, and format conversion entails non-negligible overheads (§6.1). That being said, as this approach only transmits non-zero values, it could be more advantageous than the dense block format when the block has more than $\frac{bs \cdot c_v}{c_i + c_v}$ zero values within it. We observe that in our settings, the dense block format maintains high sparsity for a range of practical block sizes (§6.3).

3.4 Performance analysis

We analyze the theoretical benefits of OmniReduce following the modeling approach of Patarasuk et al. [41]. We use a performance model to compare OmniReduce versus ring AllReduce, which is bandwidth optimal [41] and versus AGsparse AllReduce. As the primary interest is the dominating communication time, our analysis ignores the unitary local reduction time in the model below since pipelining could mask much of this latency term.

Ring AllReduce is a widely-adopted AllReduce algorithm and is the default algorithm for Gloo and NCCL. Consider N workers and that each worker has full-duplex

Algorithm 2: Block aggregation w/ loss recovery

```
At Worker:
    $p.next, next \leftarrow$ first non-zero block past block 0
    $p.block, p.ver \leftarrow 0$
    $p.stream \leftarrow$ stream/thread ID $s$
    $p.wid \leftarrow$ worker ID
    $p.data \leftarrow G_s[0 : bs]$
    send $p$ to $agg$; start_timer($p$)
    repeat upon receive p(data, ver, block, next, stream, wid)
        cancel_timer($p$)
        $G_s[p.block : p.block + bs] \leftarrow p.data$
        $p.ver \leftarrow (p.ver + 1)\%2$
        if $p.next = next$ then
            $p.block \leftarrow next$
            $p.data \leftarrow G_s[next : next + bs]$
            $p.next, next \leftarrow$ next non-zero block or else $\infty$
            $p.wid \leftarrow$ worker ID
            send $p$ to $agg$; start_timer($p$)
        else
            $p.next \leftarrow next$
            $p.data \leftarrow \{0\}$        // empty packet payload
            send $p$ to $agg$; start_timer($p$)
    until $p.next = \infty$
    upon timeout for $p$        // timeout handler
        send $p$ to $agg$; start_timer($p$)

At Aggregator:
    for $s$ in $0 \ldots S-1$ do        // pool initialization, 2-way versioned
        $slot_s[2] := \{0\}$
        $seen_s[2, N], count_s[2] := \{0\}$
        $min\_next_s := \infty$
    forever upon receive p(data, ver, block, next, stream, wid)
        $s \leftarrow p.stream$        // reference p's slot
        if $seen_s[p.ver, p.wid] = 0$ then
            $seen_s[p.ver, p.wid] \leftarrow 1$
            $seen_s[(p.ver + 1)\%2, p.wid] \leftarrow 0$
            $count_s[p.ver] \leftarrow (count_s[p.ver] + 1)\%N$
            if $count_s[p.ver] = 1$ then
                $slot_s[p.ver] \leftarrow p.data$
                $min\_next_s \leftarrow p.next$
            else
                $slot_s[p.ver] \leftarrow slot_s[p.ver] + p.data$
                $min\_next_s \leftarrow \min(min\_next_s, p.next)$
            if $count_s[p.ver] = 0$ then
                $p.data \leftarrow slot_s[p.ver]$
                $p.next \leftarrow min\_next_s$
                send $p$ to all workers
        else
            if $count_s[p.ver] = 0$ then
                $p.data \leftarrow slot_s[p.ver]$
                send $p$ to $p.wid$
```

Algorithm 3: Extension to sparse format

```
1  At Worker:
2      $nextkey\_idx := bs$
3      $p.nextkey := K[nextkey\_idx]$
4      $p.keys \leftarrow K[0:bs]$
5      $p.values \leftarrow V[0:bs]$
6      $p.wid \leftarrow$ Worker ID
7      send $p$ to $agg$
8      repeat upon receive $p(keys, values, nextkey, wid)$
9          update $K,V$ according to $p.keys$, $p.values$
10         if $p.nextkey \geq K[nextkey\_idx]$ then
11             $p.keys \leftarrow K[nextkey\_idx : nextkey\_idx + bs]$
12             $p.values \leftarrow V[nextkey\_idx : nextkey\_idx + bs]$
13             $p.nextkey \leftarrow K[nextkey\_idx + bs]$
14             $nextkey\_idx \leftarrow nextkey\_idx + bs$
15             $p.wid \leftarrow$ Worker ID
16             send $p$ to $agg$
17         end
18     until $update\ K,V\ is\ complete$
19 At Aggregator:
20     $nextkey[N] := \{-\infty\}$
21     $sent := 0$
22     forever upon receive $p(keys, values, nextkey, wid)$
23         $nextkey[p.wid] \leftarrow p.nextkey$
24         $send\_up\_to \leftarrow \min(nextkey)$
25         update $K,V$ accordingly
26         if $send\_up\_to > sent$ then
27             $p.keys \leftarrow$ keys from $sent$ to $send\_up\_to$ in $K$
28             $p.values \leftarrow$ values from $sent$ to $send\_up\_to$ in $V$
29             $p.nextkey \leftarrow send\_up\_to$
30             $sent \leftarrow send\_up\_to$
31             send $p$ to all workers
32         end
```

network bandwidth B; the time to perform a ring-based AllReduce operation of S elements is:

$$T_{ring} = 2(N-1)(\alpha + \frac{S}{NB})$$

Where α is the one-way network latency between workers (assumed to be uniform).

AGsparse AllReduce is a commonly used method to reduce sparse format data (key and value pairs). It consists of two steps: (1) AllGather keys and values, and (2) local reduction. Let $D \in [0,1]$ be the density of elements at each worker; the number of input elements to AllGather is $2DS$ (i.e., DS keys and DS values). An AllGather operation only performs the first phase of the AllReduce operation, halving its time for input with $2DSN$ elements. Thus, the AGsparse AllReduce time is:

$$T_{AGsparse} = (N-1)(\alpha + \frac{2DS}{B})$$

OmniReduce achieves bandwidth-optimality when the aggregator bandwidth matches the combined worker bandwidth NB and only non-zero elements are transmitted. This best-case scenario is analyzed here, which implies that block density is the same as the element density D. Note that the number of aggregator nodes used is not relevant because fine-grained parallelism enables ideal linear scaling through sharding. The aggregator receives a total of DS elements ($\frac{DS}{N}$ from each worker).

As data transmission and aggregation at the aggregator is pipelined, the latency of intermediate packets is masked. Thus, the OmniReduce time is:

$$T_{OmniReduce} = \alpha + \frac{DS}{B}$$

Speedup. To ease comparison, we distinguish two cases: (1) very sparse data and (2) sparse-to-dense data.

Very sparse data: in this case, D is very small and the α latency term dominates the bandwidth term. OmniReduce is expected to be better than both ring AllReduce and AGsparse AllReduce because OmniReduce's performance does not depend on the number of workers N.

Sparse-to-dense data: as the data volume is larger in this case, we can ignore the latency α. We calculate the theoretical speedup factor of OmniReduce relative to other approaches as follows:

	$SU_{\text{vs. ring}}$	$SU_{\text{vs. AGsparse}}$
$\frac{T_{\text{other}}}{T_{\text{OmniReduce}}}$	$\frac{2(N-1)}{ND}$	$2(N-1)$

The performance benefit of OmniReduce is two-fold. First, OmniReduce is much more scalable, and both speedup factors grow with the number of workers because OmniReduce's time does not depend on the number of workers. This speedup is fundamental and exists even with a dense input ($D = 1$). Second, in contrast to ring AllReduce, OmniReduce only sends non-zero elements, which reduces the time proportion to $\frac{1}{D}$.

Further, we observe that OmniReduce remains advantageous even in a co-location setting where the aggregator service is sharded and co-located across N workers (each of which thus has $\frac{B}{2}$ bandwidth). In this case, the benefit over ring AllReduce overall diminishes by a factor of 2 and $SU_{\text{vs. ring}} = 1$ when $D = 1$.

4 Block-based gradient sparsification

Given the performance benefits of OmniReduce for sparse gradients by sending only non-zero blocks, one can think of using OmniReduce with block-based gradient sparsification techniques when gradients are not sparse. While many element-wise sparsification techniques exist in the literature, e.g., Random-k [52], Top-k [3, 36], and threshold [15, 53], no block-based sparsification technique exists.

Hence, as a natural extension to the existing element-wise sparsification techniques, we devise and experiment with the following block-based sparsification schemes:

• *Block Random-k*: Randomly sample k blocks.

• *Block Top-k*: Select Top-k blocks according to mean absolute gradient value.

• *Block Top-k Ratio*: Select Top-k blocks according to mean absolute update ratio, where update ratio for an element is the ratio of gradient to parameter value.

• *Block threshold*: Select blocks with mean absolute gradient value higher than a given threshold.

While a new theoretical analysis for block-based sparsification is out of scope, we show that *Block Random-k* and *Block Top-k* are δ-compressors [26], and hence *converge according to the Error-Feedback theory* [52, 59].

Lemma. *Let b denote the total number of blocks. Both Block Random-k and Block Top-k are δ-compressors with $\delta = \frac{k}{b}$, and $\delta = \frac{k^2}{db^2}$, respectively.*

Proof. The proof is in Appendix A. □

Using this lemma, Theorem 1 in [59] gives the convergence result for compressed distributed SGD with error-feedback for any δ-compressor. Our results show that block-based gradient compression converges (§6.2).

5 Implementation

We implement OmniReduce using the Intel DPDK kernel bypass framework. The worker component is written in $\sim 1{,}500$ lines of codes (LoC), while the aggregator is ~ 600 LoCs in C++. We integrate OmniReduce with PyTorch's DistributedDataParallel (DDP) package (`torch.distributed`), which transparently performs distributed data-parallel training. To make full use of bandwidth, we use DPDK flow director to scale packet processing to 4 CPU cores on both workers and aggregators.

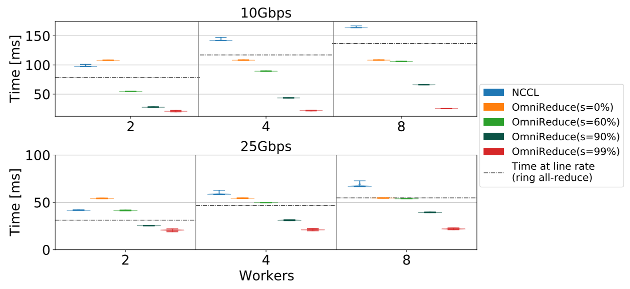


Figure 3: Time to AllReduce as workers and sparsity vary.

The number of packets processed by each worker is set to 256 (4 cores times 64 packets per core).

To determine non-zero blocks, we use the GPU to calculate a bitmap (one bit per block). This function runs whenever a part of the gradient is ready for aggregation, and we observed a minimal overhead (typically less than 1%; at most 3%). Because during back-propagation DDP partially overlaps communication and computation (i.e., communication starts as soon as the earliest partial results of back-propagation are available), the bitmap calculation overheads are negligible.

Currently, our OmniReduce prototype does not make use of RDMA. As a result, its performance is limited to $\sim$25 Gbps of line-rate bandwidth due to the limited number of CPU cores in our setup. This limitation is, however, not fundamental. To see this, *we conducted preliminary experiments with RDMA*, using block-sized RDMA messages and *observed that the point-to-point communication performance is close to the ideal goodput at 100 Gbps*. We supply to this deficit by including experimental results (§6.2) where we emulate the level of performance that, as we observed, RDMA affords us.

6 Evaluation

We evaluate OmniReduce's performance and compare it to both dense and sparse state-of-the-art collective libraries. We analyze the influence of different factors like block size, sparsity, and packet loss rates. Our experiments rely on several microbenchmarks as well as six end-to-end training workloads.

Testbed. Our testbed consists of 16 machines. Eight of them are equipped with 1 NVIDIA P100 GPU, dual 10-core CPU Intel Xeon E5-2630 v4 at 2.20 GHz, and 128 GB of RAM, acting as workers. The other eight machines are equipped with dual 8-core Intel Xeon Silver 4108 CPU at 1.80 GHz and have no GPUs; these serve as aggregators. All machines have a Mellanox ConnectX-5 NIC, and CPU frequency scaling is disabled. The machines run Ubuntu 18.04 (Linux 4.15.0), CUDA 10.1 (where applicable), PyTorch 1.5.0a0 and NCCL 2.4.8.

Microbenchmark setup. For microbenchmarks, we use AllReduce time as the performance metric. We collect measurements at each worker for 200 iterations with 10 warm-ups. Sparse tensors are generated randomly at each iteration. As the baseline, we use the ring AllReduce algorithm implemented in NCCL and run it with RDMA for maximum performance.

Training workloads. We use six real-world models for the end-to-end experiments, including two image classification models, two NLP models, and two recommendation models. The detailed information of the models, datasets, and batch sizes we use are shown in Table 1. We measure *training throughput* (the number of training samples processed per unit of time) for 200 iterations. We observed that throughput stabilizes after the first 100 iterations. Thus, we exclude them and consider throughput from the later iterations.

6.1 Microbenchmarks

Comparison with dense AllReduce. We first compare OmniReduce with NCCL on the most commonly used collective operation for DDL: dense AllReduce. We devise this micro-benchmark atop PyTorch by generating input tensors on the GPU and invoking PyTorch's all_reduce API from the torch.distributed package, switching the backend to OmniReduce or NCCL.

We use tensor sizes from 100 MB to 1,000 MB, and observe that tensor size has a low impact on the throughput. Therefore, we only report results for 100 MB tensors for these experiments. Moreover, to analyze tensor sparsity's influence on performance, we generate tensors with different sparsity s from 0% to 99%. Sparsity is the proportion of zero values in input tensors. All tensors are generated randomly, and so the non-zero blocks randomly overlap among workers.

Figure 3 shows the results as we vary the number of workers from 2 to 8. In addition to performance, we plot as a dashed line the theoretically-lowest ring AllReduce time [41] based on the maximum goodput, given the line-rate bandwidth. The results show that OmniReduce achieves up to 6.4× speedup than NCCL at 99% sparsity. With 60% sparsity or more, OmniReduce always outperforms NCCL. When data is dense, OmniReduce with two workers is slower than NCCL. Note that in this case, the aggregation is not necessary because full-duplex communication is the ideal strategy. However, we attribute this to two factors: (1) OmniReduce, unlike NCCL, does not use RDMA to accelerate point-to-point communication; (2) OmniReduce adds meta-data (e.g., *next*) within each packet, which is pure overhead when data is dense. Overall these performance gains confirm the previous theoretical insights (§3.4) and the observation that non-zero block overlap influences performance

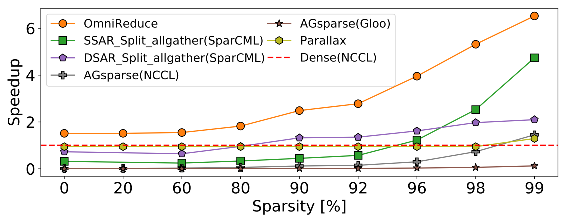


Figure 4: Comparison of OmniReduce and other sparse AllReduce methods as sparsity varies.

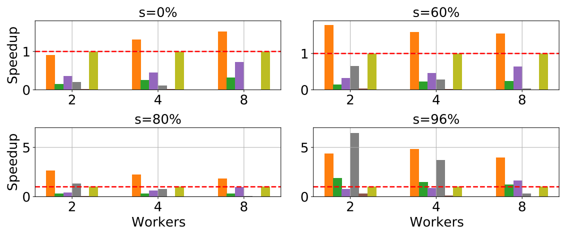


Figure 5: Scalability of OmniReduce and other sparse AllReduce methods as workers and sparsity vary. Refer to Figure 4 for legends.

(sensitivity analysis in §6.3).

As expected, OmniReduce exhibits higher scalability than NCCL. As the number of workers increases, OmniReduce with dense data ($s = 0\%$) maintains a constant AllReduce time of ∼110 ms whereas that of NCCL increases from ∼100 ms to ∼170 ms. However, when $s > 0\%$, OmniReduce's performance is affected by the number of workers, especially while $s < 90\%$. Our performance model did not capture this behavior and we discuss this apparent gap: The model assumed a uniform block sparsity across workers; however, the input tensors are generated randomly within each worker in this experiment. Thus, workers are likely to hold non-zero blocks distributed at different parts of the tensor. As long as one or more workers hold a non-zero block at a certain index, one round-trip time is needed, meaning that non-zero block overlap condition can influence OmniReduce's performance. In particular, the block sparsity decreases with a higher worker count. Nevertheless, the results show that even for dense input tensors, OmniReduce improves performance over NCCL by 1.3× and 1.5× respectively for 4 and 8 workers. Notably, performance improvement increases with a higher number of workers.

Comparison with other sparse AllReduce methods. We focus on three sparse AllReduce approaches.
1) AGsparse, which PyTorch implemented for sparse format (key-value pairs) tensors atop Gloo's AllGather operation. We also implement AGsparse atop NCCL since Gloo doesn't use RDMA in our setting.
2) Two SparCML [45] methods – SSAR_Split_allgather

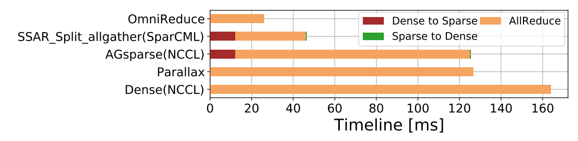


Figure 6: AllReduce execution breakdown with $s = 99\%$.

and DSAR_Split_allgather – that dominate performance for all SparCML methods in our experiments.
3) Parallax [30], which uses parameter server (PS) to aggregate sparse format tensors and AllReduce operations to aggregate dense format tensors.

We compare the performance of OmniReduce with all these sparse AllReduce methods using a 100MB tensor, with the sparsity varying from 0% to 99%. We exclude the format conversion overheads (for now), i.e., we use dense format for Omnireduce and the baseline (dense AllReduce using NCCL) while using sparse format (key-value pairs) for AGsparse and SparCML. We mimic Parallax runtime profiler by an ideal oracle: For each tensor, we separately measure the sparse format performance with PS and the dense format performance with AllReduce, then cherry-pick the better one as Parallax's performance.

To fairly compare with SparCML, we use the benchmark provided in the SparCML release [51] and limit bandwidth to 10 Gbps since SparCML was prototyped and evaluated with 1 GbE (at 25 Gbps it doesn't give sensible speedup even at high sparsity).

Figure 4 presents the performance of OmniReduce, AGsparse, SparCML and Parallax normalized to the baseline in an 8-worker setting. OmniReduce outperforms all competitors at any sparsity. Compared to the baseline, OmniReduce achieves at least 1.5× and up to 6.3× speedups at $s = 99\%$, whereas SparCML, AGsparse (NCCL), and Parallax are only beneficial when the tensor sparsity is higher than 90%, 98%, and 99%, respectively.

Figure 5 further shows the speedup for four sparsity levels as we vary the number of workers from 2 to 8.[3] Following our theoretical insights (§3.4), we expect OmniReduce to have the best scalability and AGsparse to have the worst scalability. OmniReduce is only affected by the number of workers N to the extent that N influences global sparsity, whereas AGsparse scales poorly with the number of workers (the speedup actually decreases). For dense tensors ($s = 0\%$), the speedup of OmniReduce increases with more workers. When the sparsity is higher, the speedup of OmniReduce tends to diminish as workers increase. This is because the non-zero blocks in every worker do not completely overlap, which overall results in lower global sparsity. We study this effect in §6.3.

[3]Parallax is the same as NCCL; PS is only effective at 99% sparsity.

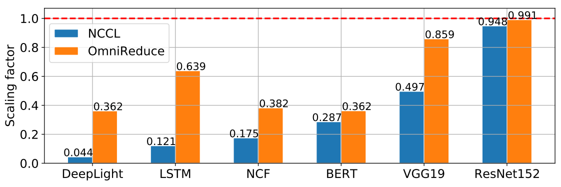


Figure 7: Scaling factor comparison of OmniReduce and NCCL. Results for 8 workers; 2 and 4 workers are similar. See Figure 1 for the definition of scaling factor.

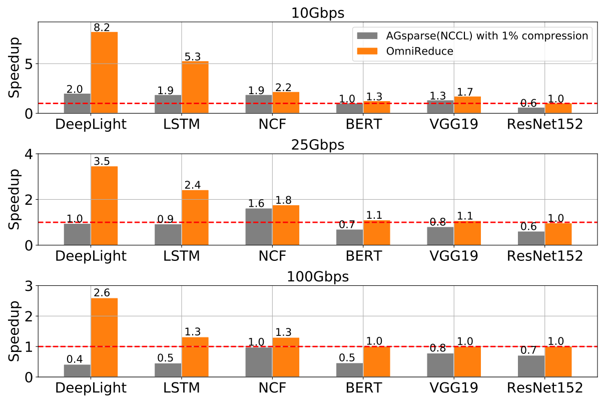


Figure 8: Training performance speedup for 6 DNNs normalized to dense AllReduce.

Amongst other sparse methods, SparCML has better scalability than AGsparse, especially the DSAR_Split_allgather method, whose speedup always increases along with workers. This method's benefit comes from automatically switching between sparse representation and dense representation. According to the results in [45], this scalability trend saturates at 16 workers, and the speedup then decreases for higher worker counts. Nevertheless, OmniReduce outperforms all SparCML methods at any sparsity with 2 to 8 workers.

Format conversion cost. The experiments above use either a dense or sparse format input matching each method's assumption. In practice, our DNNs use dense tensors, and format conversion is required for AGsparse and SparCML. Figure 6 bestows the total AllReduce time when including format conversion costs. These overheads increase with lower sparsity. In this scenario, OmniReduce's advantages are even more apparent.

6.2 End-to-end training

We demonstrate that OmniReduce increases scalability and accelerates training for real-world DNNs (Table 1).

Scalability. As discussed, inefficient collective communication in DDL results in poor scalability. Figure 7 shows that OmniReduce improves the scalability in every DNN benchmark, whereas the scaling factors for

Overlap	**DeepLight**	**LSTM**	**NCF**	**BERT**	**VGG19**	**ResNet152**	**sBERT**
None	59.49%	18.10%	27.48%	0.60%	0.03%	0.01%	83.15%
2	11.94%	4.58%	17.78%	0.11%	0.02%	0.01%	12.81%
3	5.61%	1.98%	13.10%	0.04%	0.01%	0.00%	2.63%
4	3.40%	1.11%	10.29%	0.02%	0.01%	0.00%	0.78%
5	2.36%	0.71%	8.52%	0.01%	0.02%	0.00%	0.31%
6	1.85%	0.50%	7.60%	0.01%	0.06%	0.01%	0.14%
7	1.73%	0.40%	7.39%	0.01%	1.05%	0.01%	0.07%
All	13.62%	72.61%	7.85%	99.20%	98.79%	99.96%	0.11%
Sparse %	86.38%	27.39%	92.15%	0.80%	1.21%	0.04%	99.89%

Table 2: Breakdown of OmniReduce communication by the number of workers that overlap blocks (from none to all). sBERT denotes BERT with block-based compression.

NCCL decreases with more workers. OmniReduce outperforms NCCL in these workloads and achieves a substantial scalability improvement, especially with large DNNs: 8.2× / 5.3× for DeepLight / LSTM, respectively.

Training speedup. Figure 8 shows the training speedup of OmniReduce and AGsparse (NCCL) relative to dense AllReduce in an 8-worker setup. OmniReduce accelerates training by up to 8.2× and 3.5× compared to NCCL at 10 Gbps and 25 Gbps, respectively. For certain DNNs, e.g., ResNet152, there is no speedup. This is expected because not every DNN is network-bound [58]. However, OmniReduce does not hurt performance in this case. Moreover, the speedup is understandably greater for DNNs with high gradient sparsity. OmniReduce also outperforms AGsparse (NCCL) and would provide benefits even at 100 Gbps. We now elaborate on these results.

Since AGsparse methods are beneficial only at high sparsity (§6.1), we assume element-wise gradient compression at 1% ($s = 99\%$) before invoking AGsparse AllReduce. To focus on collective communication performance, we do not consider compression overheads (even though they may be prohibitive in practice [34, 57]). SparCML and Parallax do not integrate with PyTorch, and we could not use it in these experiments. We avoid comparing them using other ML toolkits because it is not clear how to get a fair comparison as models and system components (e.g., Horovod vs DDP) differ substantially.

Table 2 breaks down the proportion of communication by the extent of block overlap among workers. The last row highlights how much the sparse optimizations of OmniReduce contribute to the training speedup.

As mentioned (§5), armed with positive preliminary results for block-based aggregation using RDMA, we obtain results at 100 Gbps by emulating the expected aggregation latency of OmniReduce. For fairness, OmniReduce results are normalized by the ideal AllReduce performance (§3.4). OmniReduce provides benefits in the range 1.2× to 2.6× for half of the workloads and precisely the DNNs with a large proportion of embedding weights that yield gradients with more than 84% sparsity (Table 1). BERT also has large embedding weights, but

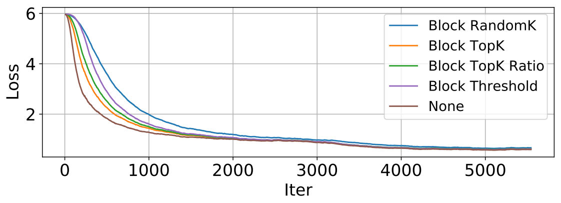


Figure 9: Training loss change with iteration (BERT).

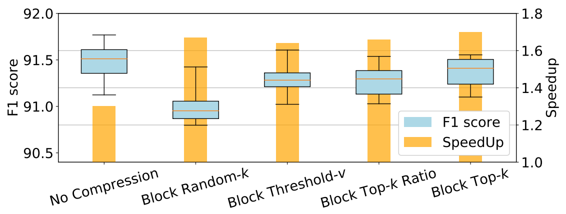


Figure 10: Training accuracy and speedup with OmniReduce after using different compression methods (BERT).

they only account for a minor part (∼20%) of the model. We next show how OmniReduce with block-based gradient compression accelerates the BERT workload.

OmniReduce speedup with block-based compression. We apply all 4 block-based compression methods introduced in §4 to speedup the BERT workload, which consists of a large model (1.2 GB), but its gradient sparsity is only ∼9%. We use 0.01 as threshold and otherwise apply $k = 1\%$ compression ratio. While evaluating the performance speedup relative to NCCL, we also track the model accuracy (F1 score). We repeat the experiments ten times, and plot ranges with quartiles. We fine-tune BERT for the question answering task on the Stanford Question Answering Dataset [44].

Figure 10 shows these results for an 8-worker setup at 10 Gbps. OmniReduce now accelerates training by ∼1.7×. The training loss change shown in Figure 9 reveals that block-based compression methods can preserve convergence for BERT. Compression affects accuracy slightly (at most a 1-point drop in F1 score), highlighting the trade-off between speedup and accuracy depending on the compression level.

6.3 Sensitivity analysis

Block size. Figure 11 (left) shows how block sparsity influences OmniReduce performance for various choices of the block size. Larger block size is better at amortizing the per-packet meta-data overheads. As the block sparsity increases, the performance gap diminishes. Figure 11 (right) shows the effective sparsity as a function of block size for various DNNs. A block size of one identifies the real gradient sparsity. Models with large embedding layers can maintain large block sparsity at

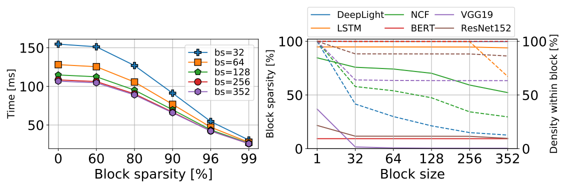


Figure 11: Influence of block size (*bs*) and sparsity. Legends right figure: the solid lines represent the block sparsity and the dashed lines represent the density within block.

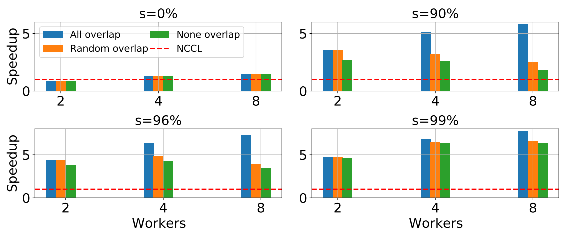


Figure 12: Effect of non-zero element overlap among workers on the OmniReduce performance.

packet-size blocks. Notably, the density of non-zero values within each block does not decrease too drastically in many cases. Given these characteristics of block sparsity in relation to performance and density within a block, we choose block size 256 as the default for our setting.

Overlap of non-zero blocks. Two extremes exist: (1) all non-zero blocks overlap at every worker, and (2) not a single non-zero block overlaps among N workers. Respectively dense AllReduce (of just the non-zero blocks) and AGsparse ideally address these extremes whereas OmniReduce – while capable of handling the entire spectrum – is best suited for when data is sparse and block overlap somewhat. Figure 12 shows the speedup of OmniReduce relative to NCCL for the extremes as well as with an amount of overlap generated at random. It is noteworthy that at both no sparsity ($s = 0\%$) or very high sparsity ($s > 95\%$), the impact of overlap is small or none. This is because when the tensor is dense, the total number of elements is equal to S in all cases, while when the tensor is very sparse, NDS is close to DS. Recall we denote with D the average data density and S the tensor size. However, when $s \in [60\%, 90\%]$, the "all overlap" performance is significantly better than the other cases.

Loss recovery. Finally, we show how different packet loss rates (between 0.01% and 1%) affect OmniReduce. Since no packet loss actually occurs in our experiments, we emulate packet loss assuming uniform probability at a given loss rate. Figure 13 shows the difference between AllReduce time with no loss minus AllReduce time with a given loss rate. We compare against Gloo and NCCL while using TCP as transport protocol in order to see a reaction to packet drops. In our setup, RDMA assumes

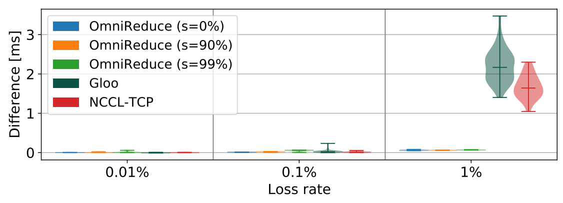


Figure 13: Performance drop of AllReduce time due to packet loss and recovery. No packet loss is the baseline.

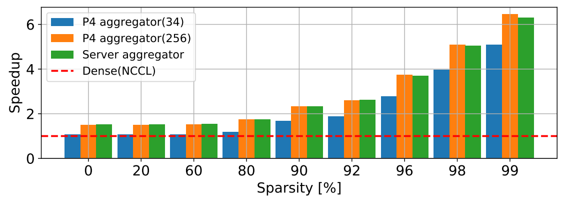


Figure 14: OmniReduce in-network P4 aggregator compared to server-based aggregator for two block sizes.

a lossless fabric. The results show that OmniReduce's packet retransmission is effective in every sparsity level and loss rate. However, with a high loss rate (1%), the performance of Gloo and NCCL-TCP drops sharply. We attribute this to TCP congestion control.

7 Extensions

In-network aggregation. Mellanox SHARP [49] and SwitchML [48] demonstrated the feasibility of streaming collective aggregation protocols where the aggregation takes place within network switches. OmniReduce lends itself to these advancements. In particular, because the time and space complexity of the OmniReduce aggregator is low and the aggregation function is the arithmetic sum, we set off to demonstrate that the aggregator can run on suitable network switches. We implement Algorithm 2 in P4 [5] and offload it to a Barefoot Tofino switch. Figure 14 shows that with this offload, OmniReduce is slightly faster than with the server-based aggregator. This implementation inherits some of the limitations described by Sapio et al. [48] in terms of numeric representation and slot size. However, SHARPv2 [49] demonstrated that 100 Gbps line-rate aggregation of floating-point values is within reach for current technology.

Generalized collective operations. We observe that our algorithms generalize to three collective operations: AllReduce, AllGather, and Broadcast. In fact, AllGather can be viewed as a sparse AllReduce with no block overlap. Broadcast is a simpler case in which there is no block overlap, and the tensor size of $N-1$ workers is 0. In these cases, the aggregator realizes both a multicast function and flow control mechanism to coordinate collective communication. By not sending zero blocks, OmniReduce improves the efficiency for these collectives.

Numeric reproducibility and non commutative ops. Due to the numeric representation of floating-point values, sum is not generally a commutative operator. OmniReduce can support numeric reproducibility and non-commutative operators by enforcing a serial order of slot updates. At the cost of a larger pool of slots, one can modify our algorithms so that every slot is writable by one worker at a time, in a pre-defined sequence, while pipelining slot updates for efficiency. For example, in an N worker group, worker 1 is $N-1$ blocks ahead of worker N, worker 2 is $N-2$ blocks ahead, and so on. The overhead for doing so is that slot aggregation latency increases with $O(\log_2 N)$; throughput, however, is unaffected. Signaling information to synchronize progress can be piggybacked by data packets to lower overheads.

8 Other related work

Efficient communication in DDL. Several efforts optimize DDL communication ranging from designing high-performance PS software [37] and transfer scheduler [42], to improving collective communication in heterogeneous network fabrics [10] and within multi-GPU servers [55], to developing in-network reduction systems [31, 33, 49, 48, 47], through customizing network congestion protocols and architecture [17]. OmniReduce leverages data sparsity to optimize communication and is complementary to these efforts.

Accelerating DDL. Orthogonal to our work, various works propose efficient distributed optimization algorithms [35, 4, 32, 60]. Besides data parallelism, other parallelization strategies include model parallelism [12, 9], and hybrids of model and data parallelism [23, 24]. Going one step further, pipeline parallelism [38] processes multiple batches simultaneously, with individual layers either having model or data parallelism. OmniReduce speeds up the data parallel aspect of these works.

9 Conclusion

We leverage sparsity in distributed deep learning to accelerate training for six real-world DNNs by up to 8.2×. OmniReduce is a generic collective communication primitive aiming especially at efficiently aggregating sparse data. We proposed streaming aggregation algorithms that outperform previous approaches, surpassing them by 2-6.4×. Our approach runs efficiently on a server, yet its modest computational complexity affords it to run on programmable switch ASICs.

References

[1] A. F. Aji and K. Heafield. Sparse Communication for Distributed Gradient Descent. In *EMNLP-IJCNLP*, 2017.

[2] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NeurIPS*, 2017.

[3] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli. The convergence of sparsified gradient methods. In *NeurIPS*, 2018.

[4] D. Basu, D. Data, C. Karakus, and S. Diggavi. Qsparse-local-SGD: Distributed SGD with quantization, sparsification and local computations. In *NeurIPS*, 2019.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. *arXiv 2005.14165*, 2020.

[7] BytePS: A High Performance and Generic Framework for Distributed DNN Training. `https://github.com/bytedance/byteps`.

[8] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv 1312.3005*, 2013.

[9] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[10] M. Cho, U. Finkler, D. S. Kung, and H. C. Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *MLSys*, 2019.

[11] Criteo's 1TB Click Prediction Dataset. `https://docs.microsoft.com/en-us/archive/blogs/machinelearning/now-available-on-azure-ml-criteos-1tb-click-prediction-dataset`.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, and e. a. Yang, Ke. Large scale distributed deep networks. In *NeurIPS*, 2012.

[13] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. *arXiv 2002.06987*, 2020.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv 1810.04805*, 2018.

[15] A. Dutta, E. Bergou, A. M. Abdelmoniem, C. Y. Ho, A. N. Sahu, M. Canini, and P. Kalnis. On the Discrepancy between the Theoretical Analysis and Practical Implementations of Compressed Communication for Distributed Deep Learning. In *AAAI*, 2020.

[16] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, 1994.

[17] N. Gebara, T. Ukyab, P. Costa, and M. Ghobadi. PANAMA: Network Architecture for Machine Learning Workloads in the Cloud. Technical report, 2020. `https://people.csail.mit.edu/ghobadi/papers/panama.pdf`.

[18] Gloo. `https://github.com/facebookincubator/gloo`.

[19] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TiiS*, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[21] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural Collaborative Filtering. In *WWW*, 2017.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*, 2019.

[23] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *ICML*, 2018.

[24] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*, 2019.

[25] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv 1602.02410*, 2016.

[26] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *ICML*, 2019.

[27] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *ICLR*, 2017.

[28] J. Keuper (Fehr) and F.-J. Pfreundt. Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability. In *MLHPC*, 2016.

[29] J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 1952.

[30] S. Kim, G. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B. Chun. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *EuroSys*, 2019.

[31] B. Klenk, N. Jiang, G. Thorson, and L. Dennison. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ISCA*, 2020.

[32] A. Koloskova, S. U. Stich, and M. Jaggi. Decentralized Stochastic Optimization and Gossip Algorithms with Compressed Communication. In *ICML*, 2019.

[33] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *ISCA*, 2019.

[34] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Gerhard Schwing, H. Esmaeilzadeh, and N. Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *Micro*, 2018.

[35] T. Lin, S. U. Stich, K. K. Patel, and M. Jaggi. Don't Use Large Mini-batches, Use Local SGD. In *ICLR*, 2019.

[36] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*, 2018.

[37] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. PHub: Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *SoCC*, 2018.

[38] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*, 2019.

[39] NVIDIA Collective Communication Library (NCCL). `https://developer.nvidia.com/nccl`.

[40] NVIDIA Ampere Architecture In-Depth. `https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/`.

[41] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J Parallel Distrib Comput*, 2009.

[42] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*, 2019.

[43] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv 1802.05365*, 2018.

[44] P. Rajpurkar, R. Jia, and P. Liang. Know what you don't know: Unanswerable questions for SQuAD. *arXiv 1806.03822*, 2018.

[45] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler. SparCML: High-Performance Sparse Communication for Machine Learning. In *SC*, 2019.

[46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 2015.

[47] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets*, 2017.

[48] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik. Scaling Distributed Machine Learning with In-network Aggregation. *arXiv 1903.06701*, 2019.

[49] Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). `https://www.mellanox.com/products/sharp`.

[50] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

[51] SparCML. `https://gitlab.com/renggliс/SparCML`.

[52] S. U. Stich, J.-B. Cordonnier, and M. Jaggi. Sparsified SGD with Memory. In *NeurIPS*, 2018.

[53] N. Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *ISCA*, 2015.

[54] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *IHPCFL*, 2005.

[55] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*, 2020.

[56] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *NeurIPS*, 2017.

[57] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation. Technical report, KAUST, Apr 2020. `http://hdl.handle.net/10754/662495`.

[58] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is Network the Bottleneck of Distributed Training? In *NetAI*, 2020.

[59] S. Zheng, Z. Huang, and J. Kwok. Communication-efficient distributed blockwise momentum SGD with error-feedback. In *NeurIPS*, 2019.

[60] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. Asynchronous stochastic gradient descent with delay compensation. In *ICML*, 2017.

A Proof of convergence

We first start with the definition of a δ-compressor, then prove that *Block Random-k*, and *Block Top-k* are δ-compressors, and then finally relate it to the convergence result for δ-compressors.

Definition. ***(δ-compressor) [26]*** *A probabilistic operator $\mathscr{C}: \mathbb{R}^d \to \mathbb{R}^d$ is called a δ-approximate compressor for $\delta \in (0,1]$ if*

$$\mathbb{E}\|x - \mathscr{C}(x)\|_2^2 \le (1-\delta)\|x\|_2^2 \quad \forall x \in \mathbb{R}^d.$$

Lemma. *Let b denote the total number of blocks. Both block-rand$_k$ (Block Random-k) and block-top$_k$ (Block Top-k) are δ-compressors with $\delta = \frac{k}{b}$, and $\delta = \frac{k^2}{db^2}$ respectively*

Proof. Without loss of generality, we assume the dimension d to be a multiple of the number of blocks b. Let for any $x \in \mathbb{R}^d$, $x_i \in \mathbb{R}^{d/b}$ denote the i^{th} block. Then, $x = [x_1^T, x_2^T, \ldots, x_b^T]^T$. Also, let $\Omega_k = \binom{[b]}{k}$ denote the set of all k element subsets of $[b]$.
Block Random-k: We have,

$$\begin{aligned}\mathbb{E}\|x - \textit{block-rand}_k(x)\|_2^2 &= \frac{1}{|\Omega_k|}\sum_{\omega \in \Omega_k}\sum_{i=1}^{b}\|x_i\|_2^2 \cdot \mathbb{I}\{i \notin \omega\} \\ &= \sum_{i=1}^{b}\|x_i\|_2^2 \sum_{\omega \in \Omega_k}\frac{\mathbb{I}\{i \notin \omega\}}{\Omega_k} \\ &= (1 - \frac{k}{b})\|x\|_2^2,\end{aligned}$$

which implies that *block-rand$_k$* is a δ-compressor with $\delta = \frac{k}{b}$.

Block Top-k: Let $S_{block\text{-}top_k}(x)$ denote the set of Top-k blocks corresponding to a given x. Then,

$$\begin{aligned}\|\textit{block-top}_k(x)\|_1 &= \sum_{i \in S_{block\text{-}top_k}(x)}\|x_i\|_1 \\ &= \sum_{i \in S_{block\text{-}top_k}(x)}\|x_i\|_1 \\ &= \sum_{i \in S_{block\text{-}top_k}(x)}\sum_{k=1}^{d/b}|x_i^k| \\ &\ge \frac{k}{b}\|x\|_1 \\ &\ge \frac{k}{b}\|x\|_2,\end{aligned}$$

where x_i^k denotes the k^{th} element of i^{th} block, and the last inequality follows from the fact that $\|z\|_1 \ge \|z\|_2$ for any $z \in \mathbb{R}^d$. Finally, we have

$$\begin{aligned}\|x - \textit{block-top}_k(x)\|_2^2 &= \|x\|_2^2 + \|\textit{block-top}_k(x)\|_2^2 \\ &\quad - 2\langle x, \textit{block-top}_k(x)\rangle \\ &= \|x\|_2^2 - \|\textit{block-top}_k(x)\|_2^2 \\ &\le \|x\|_2^2 - \frac{1}{d}\cdot\|\textit{block-top}_k(x)\|_1^2 \\ &\le \|x\|_2^2 - \frac{k^2}{db^2}\|x\|_2^2 \\ &= (1 - \frac{k^2}{db^2})\|x\|_2^2,\end{aligned}$$

where the first inequality follows from the fact that $\|z\|_2 \ge \frac{\|z\|_1}{\sqrt{d}}$ for any $z \in \mathbb{R}^d$. The above result implies that *block-top$_k$* is a δ-compressor with $\delta = \frac{k^2}{db^2}$. □

Using the above lemma, Theorem 1 in [59] gives us the convergence result for compressed distributed SGD with error-feedback for an arbitrary δ-compressor.