Scaling Distributed Machine Learning with In-Network Aggregation

Amedeo Sapio[*]
KAUST

Marco Canini[*]
KAUST

Chen-Yu Ho
KAUST

Jacob Nelson
Microsoft

Panos Kalnis
KAUST

Changhoon Kim
Barefoot Networks

Arvind Krishnamurthy
University of Washington

Masoud Moshref
Barefoot Networks

Dan R. K. Ports
Microsoft

Peter Richtárik
KAUST

ABSTRACT

Training complex machine learning models in parallel is an increasingly important workload. We accelerate distributed parallel training by designing a communication primitive that uses a programmable switch dataplane to execute a key step of the training process. Our approach, SwitchML, reduces the volume of exchanged data by aggregating the model updates from multiple workers in the network. We co-design the switch processing with the end-host protocols and ML frameworks to provide a robust, efficient solution that speeds up training by up to 300%, and at least by 20% for a number of real-world benchmark models.

1 INTRODUCTION

The remarkable success of today's machine learning solutions derives from the ability to build increasingly sophisticated models on increasingly large data sets. To cope with the resulting increase in training time, distributed ML training has become a standard practice [1, 17]. Large-scale clusters use hundreds of nodes, each equipped with multiple GPUs or other hardware accelerators (e.g., TPUs [32]), to run training jobs on tens of workers that take many hours or days.

Distributed ML training is increasingly a network-bound workload. To be clear, it remains computationally intensive, and we do not claim otherwise. But the last five years have brought a 35× performance improvement [38] (thanks to GPUs [13] and other hardware accelerators [32]), a pace cloud network deployments have found hard to match, skewing the ratio of computation to communication towards the latter. As parallelization techniques like mini-batch stochastic gradient descent (SGD) training [25, 28] alternate computation phases with synchronous model update exchanges among workers, network performance now has a substantial impact on overall training time.

Can a new type of accelerator *in the network* alleviate the network bottleneck? In this paper, we demonstrate that an *in-network aggregation* primitive can accelerate distributed ML frameworks, and can be implemented using programmable dataplane switch hardware [4, 9]. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput and diminishes latency, and in turn, speeds up training time.

Building an in-network aggregation primitive suitable for usage with ML frameworks using programmable switches presents many challenges. First, the per-packet processing capabilities are limited, and so is on-chip memory. We must limit our resource usage so the switch is able to perform its primary function of conveying packets. Second, the computing units inside a programmable switch operate on integer values, whereas ML frameworks and models operate on floating point values. Third, the in-network aggregation primitive is an all-to-all primitive that does not provide workers with the ability to recognize the loss of individual packets. Therefore, in-network aggregation requires mechanisms for not only synchronizing the workers but also for tolerating packet loss.

We address these challenges in SwitchML, showing that it is indeed possible for a programmable network device to perform in-network aggregation at line rate. SwitchML is a co-design of in-switch processing with an end-host transport layer and ML frameworks. It leverages the following insights. First, aggregation involves a simple arithmetic operation, making it amenable to parallelization and pipelined execution on programmable network devices. We decompose the parameter updates into appropriately-sized chunks that can be individually processed by the switch pipeline. Second, aggregation for SGD can be applied separately on different portions of the input data, disregarding order, without affecting the correctness of the final result. We tolerate packet loss through the use of a light-weight switch scoreboarding mechanism and a retransmission mechanism driven solely by end hosts, which together ensure that workers operate in lock-step without any decrease in switch aggregation throughput. Third, ML training is robust to modest approximations in its compute operations. We address the lack of floating-point support at switch dataplanes by having the workers efficiently convert floating point values to fixed-point (using modern x86 vector instructions) and using a model-dependent scaling factor to quantize model updates with negligible approximation loss.

SwitchML integrates with distributed ML frameworks such as TensorFlow and Caffe2, to accelerate their communication, particular in regard to efficient training of deep neural networks (DNNs). Our initial prototype targets a rack-scale architecture where a single switch centrally aggregates parameter updates from directly connected workers. Though the single switch limits scalability, we

[*] Equal contribution.

note that commercially-available programmable switches can connect directly to up to 64 nodes at 100 Gbps or 256 at 25 Gbps. As each worker is typically equipped with multiple GPUs, this scale is sufficiently large to push the statistical limits of SGD [23, 28, 33, 60].

Our results (§5) show that in-network aggregation yields end-to-end improvements in training time of up to 3× for popular DNN models (§5). This training throughput in certain cases comes close to the ideal upper bound, even matching the performance of a single-node multi-GPU system (see Table 1).

We make the following contributions:

- We design and implement SwitchML, a single-rack in-network aggregation solution for ML applications. SwitchML co-designs the switch processing logic with the end-host protocols and the ML frameworks to provide a robust and efficient solution.
- We evaluate SwitchML and show that it provides substantial reductions in end-to-end training times. While the magnitude of the performance improvements are dependent on the neural network architecture (we see from 20% to 300% speedup), it is greater for models with smaller compute-to-communication ratios – good news for future, faster DNN training accelerators.
- Our approach is not tied to any particular ML framework; we have integrated SwitchML with TensorFlow using Horovod, and with Caffe2 using Gloo.

2 BACKGROUND

In the distributed setting, ML training yields a high-performance networking problem, which we highlight below after reviewing the traditional ML training process.

2.1 Training and all to all communication

Supervised ML problems, including logistic regression, support vector machines and deep learning, are typically solved by iterative algorithms such as stochastic gradient descent (SGD) [44, 45, 48] or one of its many variants (e.g., using momentum, mini-batching, importance sampling, preconditioning, variance reduction) [53]. A common approach to scaling to large models and datasets is data-parallelism, where the input data is partitioned across workers.[1] Training in a data-parallel, synchronized fashion on n workers can be seen as learning a model $x \in \mathbb{R}^d$ over input/training data D by performing iterations of the form

$$x^{t+1} = x^t + \sum_{i=1}^{n} \Delta(x^t, D_i^t),$$

where x^t is a vector of model parameters[2] at iteration t, $\Delta(\cdot,\cdot)$ is the model update function[3] and D_i^t is the data subset used at worker i during that iteration.

At every iteration, in parallel, each worker locally computes an update $\Delta(x^t, D_i^t)$ to the model parameters based on the current version of the model x^t and a subset of the local data D_i^t. Subsequently, workers communicate their updates, which are aggregated ($\sum$) into a model update. This aggregated model update is added to x^t to form the model parameters of the next iteration. The subset of data $D^t = \bigcup_{i=1}^{n} D_i^t$ is the mini-batch of the training data used in iteration t. This process takes many iterations to progress through the entire dataset, which constitutes a training epoch. At the end of a training epoch (or multiple training epochs), the model prediction error is computed on a held out validation set. Typically, training continues for several epochs, reprocessing the training data set each time, to reach a minimum validation set error (halting point).

Model (abbrv)	Training throughput (images/s)			
	Ideal	Multi-GPU	Horovod+NCCL	**SwitchML**
inception3	1132	1079 (95.3%)	799 (70.6%)	1079 (95.3%)
resnet50	1838	1630 (88.7%)	911 (49.6%)	1412 (76.8%)
vgg16	1180	898 (76.1%)	207 (17.5%)	454 (38.5%)

Table 1: Comparison of SwitchML performance in a 8-worker 10 Gbps setting to two target baselines – calculated ideal throughput (8 times single-GPU throughput), and a single-node, eight-GPU configuration [55] – and a standard distributed architecture (Horovod [52] + NCCL [43]). Batch size is 64 [55].

Note that the size of a model update is the same as the number of model parameters, which can be billions and occupy hundreds of MBs. To exchange model updates, there are two typical approaches used in popular ML frameworks like TensorFlow, PyTorch, etc. We briefly describe them below.

The parameter server (PS) approach. In this approach, each process has one of two potential roles: a worker or a parameter server. Workers process the training data, compute model updates, and send them to parameter servers to be aggregated. Then, they pull the latest model parameters and perform the next iteration. To avoid the PS from becoming a bottleneck, one needs to identify a proper ratio of number of workers to number of PS nodes. The model is then sharded over this set of PS nodes.

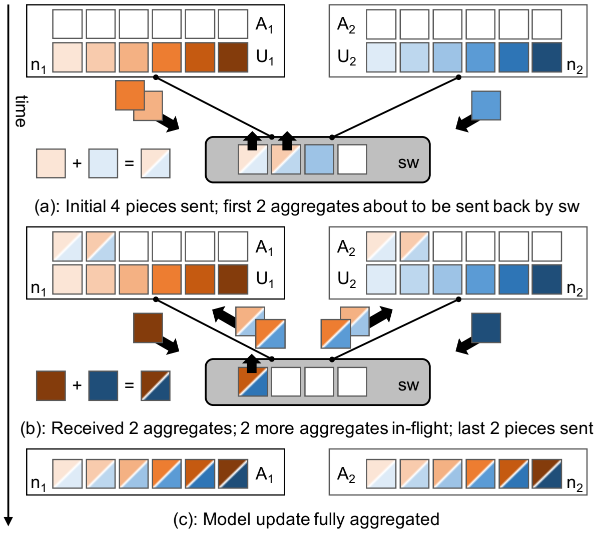
The all-reduce approach. This approach uses the all-reduce operation – a technique common in high-performance computing – to combine values from all processes and distribute the results to all processes. In this approach, each process communicates in a peer-to-peer fashion (i.e., without a PS) over an overlay network such as a ring or tree topology, depending on the all-reduce algorithm. In the case of *ring all-reduce* [5], for instance, each process communicates to the next neighboring process on the ring. This approach is bandwidth-optimal in the sense that (1) each process sends the minimum amount of data required to complete this operation, and (2) all communications are contention free [46]. Another algorithm, *halving and doubling all-reduce* [57], instead uses a binary tree topology along which updates are recursively aggregated and broadcast back to the workers.

From a communication perspective, workers produce intense bursts of traffic once they need to communicate their model updates. Moreover, the communication pattern for exchanging model updates is inherently all to all, which requires high-performance networking to avoid communication becoming a bottleneck and slowing down the training process. We now discuss why communication remains an issue for training neural networks and then propose in-network aggregation to solve it.

[1] In this paper, we do not consider model-parallel training [20, 47], although that approach also requires efficient networking. Further, we focus exclusively on distributed synchronous SGD.

[2] In applications, x is typically a 1, 2, or 3 dimensional tensor. To simplify notation, we assume its entries are vectorized into one d dimensional vector.

[3] We abstract learning rate (or step size) and model averaging inside Δ.



(a): Initial 4 pieces sent; first 2 aggregates about to be sent back by sw

(b): Received 2 aggregates; 2 more aggregates in-flight; last 2 pieces sent

(c): Model update fully aggregated

Figure 1: Example scenario of in-network aggregation of model updates. U_i is the model update computed by worker i. Workers stream pieces of model updates in a coordinated fashion. In the example, each workers can have at most 4 outstanding packets at any time to match the slots in the switch. The switch aggregates updates and multicasts back the values, which are collected into the aggregated model update A_i, then used to form the model parameters of the next iteration.

2.2 Training can be network bound

Recent studies have shown that the emergence of specialized accelerators has shifted the performance bottleneck of distributed DNN training from computation to communication [38]. For example, the throughput of GPU accelerators on complex DNNs, such as ResNet [26], has increased by more than 30x since 2012. At the same time, there hasn't been a corresponding increase in network bandwidth due to multiple reasons. Upgrading datacenter networks is expensive: network bandwidth available for compute instance on major cloud providers such as EC2 has improved only incrementally across different generational upgrades [18]. Further, the mechanisms used to coordinate distributed training, such as the parameter server architecture, not only do not scale up the total throughput on a standard cloud network stack [38] but also do not take advantage of physical network topology. As a consequence of the compound effect of these factors, the relative impact of communication overheads over computing has dramatically increased for distributed DNN training.

2.3 In-network aggregation

In-network aggregation offers a fundamental advantage over all-reduce and PS approaches. First, the communication cost of bandwidth-optimal ring all-reduce [46] – quantified in terms of data volume each worker sends and receives – is $4(n-1)\frac{|U|}{n}$, where $|U|$ is the total number of bytes to be aggregated. In contrast, in-network aggregation requires only $2|U|$ bytes. Second, the PS approach, in the dedicated setting (where nodes are exclusively either a worker or a PS), also requires $2|U|$ bytes to be sent and received at each worker. However, this setting has a much higher resource cost: in the limit, it doubles the number of required machines and network bandwidth in order to match the capability of in-network aggregation. Regardless of resource costs, in-network aggregation avoids end-host processing in order to perform aggregation and therefore provides "sub-RTT" [30] latency, which other approaches cannot achieve.

3 DESIGN

In-network aggregation is conceptually straightforward. Implementing it in a real DNN training cluster, however, is challenging. Although programmable switches allow placing computation into the network path, their limited computation and storage capabilities prevent running full aggregation of a batch of parameter updates. In addition, the system must tolerate packet loss, which, although uncommon in the cluster environment, is nevertheless possible for long-running DNN training jobs. SwitchML addresses these challenges by appropriately dividing the functionality between the hosts and the switches, resulting in an efficient and reliable streaming aggregation protocol.

3.1 Challenges

Limited computation. Mathematically, parameter aggregation is the average over a set of floating-point vectors. While a seemingly simple operation, it exceeds the capabilities of today's programmable switches. As they must maintain line rate processing, the number of operations they can perform on each packet is limited and the operations themselves can only be simple integer arithmetic/logic operations; neither floating point operations nor integer division are possible.

Limited storage. Parameter updates are large. In each iteration, each worker may supply hundreds of megabytes of gradient updates. This far exceeds the capacity of on-switch storage, which is limited to a few tens of MB and must be shared with forwarding tables and other core switch functions. This limitation is unlikely to change in the future [9], given that speed considerations require dataplane-accessible storage to be implemented using on-die SRAM.

Packet loss. While we expect packet loss to be uncommon, SwitchML must remain resilient to packet loss, without affecting efficiency and correctness (e.g., discarding part of an update or applying it twice because of a retransmission).

3.2 SwitchML overview

SwitchML aims to alleviate communication bottlenecks for distributed ML training applications using in-network aggregation, in a practical cluster setting.[4] To make this possible while meeting the above challenges, SwitchML uses the following techniques:

[4] For simplicity, in the following we assume dedicated bandwidth for the training jobs. We also assume that worker, link or switch failures are handled by the ML framework, as it is common in practice [1, 35].

Algorithm 1 Switch logic.

```
Initialize State:
    n = number of workers
    pool[s], count[s] := {0}
upon receive p(idx, off, vector)
    $pool[p.idx] \leftarrow pool[p.idx] + p.vector$        {+ is the vector addition}
    $count[p.idx]$++
    if $count[p.idx] = n$ then
        $p.vector \leftarrow pool[p.idx]$
        $pool[p.idx] \leftarrow 0$; $count[p.idx] \leftarrow 0$
        multicast $p$
    else
        drop $p$
```

Combined switch-host architecture. SwitchML carefully partitions computation between end-hosts and switches, to circumvent the restrictions of limited computational power at switches. The switch performs integer aggregation, while end-hosts are responsible for managing reliability and performing more complex computations.

Pool-based streaming aggregation. A complete model update far exceeds the storage capacity of a switch, so it cannot aggregate entire vectors at once. SwitchML instead *streams* aggregation through the switch: it processes the aggregation function on a limited number of vector elements at once. The abstraction that makes this possible is a pool of integer aggregators. In SwitchML, end hosts handle the management of aggregators in a pool – determining when they can be used, reused, or need more complex failure handling – leaving the switch dataplane with a simple design.

Fault tolerant protocols. We develop light-weight schemes to recover from packet loss with minimal overheads and adopt traditional mechanisms to solve worker or network failures.

Quantized integer-based aggregation. Floating point operations exceed the computational power of today's switches. We instead convert floating point values to 32-bit integers. We show that this can be done efficiently at end hosts without impacting either throughput or training accuracy.

We now describe each of these components in turn. To ease presentation, we describe a version of the system in which packet losses do not occur. This is already representative of a SwitchML instance running in a lossless network such as Infiniband or lossless RoCE. We remove this restriction later.

3.3 Switch-side aggregation protocol

We begin by describing the core network primitive provided by SwitchML: in-switch integer aggregation. A SwitchML switch provides a pool of s integer aggregators, addressable by index. Each slot in the pool aggregates a vector of k integers, which are delivered all at the same time in one update packet. The aggregation function is the addition operator, which is commutative and associative – meaning that the result does not depend on the order of packet arrivals. Note that addition is a simpler form of aggregation than ultimately desired: model updates need to be *averaged*. We leave the final division step to the end hosts, as this cannot efficiently be performed by the switch.

Algorithm 1 illustrates the behavior of the aggregation primitive. A packet p carries a pool index, identifying the particular aggregator to be used, and contains a vector of k integers to be aggregated. Upon receiving a packet, the switch aggregates the packet's vector ($p.vector$) into the slot addressed by the packet's pool index (idx). Once the slot has aggregated vectors from each worker,[5] the switch outputs the result – by rewriting the packet's vector with the aggregated value from that particular slot, and sending a copy of the packet to each worker. It then resets the slot's aggregated value and counter, releasing it immediately for reuse.

The pool-based design is optimized for the common scenario where model updates are larger than the memory capacity of a switch. It addresses two major limitations of programmable switch architectures. First, because switch memory is limited, it precludes the need to store an entire model update on a switch at once; it instead aggregates pieces of the model in a streaming fashion. Second, it allows processing to be done at the packet level by performing the aggregation in small pieces, at most k integers at a time. This is a more significant constraint than it may appear; to maintain a very high forwarding rate, today's programmable switches parse only up to a certain amount of bytes in each packet and allow computation over the parsed portion. Thus the model-update vector and all other packet headers must fit within this limited budget, which is today on the order of a few hundred bytes; ASIC design constraints make it unlikely that this will increase dramatically in the future [9, 12, 54]. In our deployment, k is 32.

3.4 Worker-side aggregation protocol

The switch-side logic above does not impose any particular semantics on which aggregator in the pool to use and when. Workers must carefully control which vectors they send to which pool index and, given that the pool size s is limited, how they reuse slots.

Managing the pool of aggregators imposes two requirements. For correctness, every worker must use the same slot for the same piece of the model update, and no slot can be simultaneously used for two different pieces. For performance, every worker must work on the same slot at roughly the same time to avoid long synchronization delays. To address these issues, we design a custom aggregation protocol running at the end hosts of ML workers.

For now, let us consider the non-failure case, where packets are not lost in the network. The aggregation procedure, illustrated in Algorithm 2, starts once every worker is ready to exchange its model update. Without loss of generality, we suppose that the model update's size is a multiple of k and is larger than $k \cdot s$, where k is the size of the vector aggregated in each slot and s denotes the pool size. Each worker initially sends s packets containing the first s pieces of the model update – each piece being a contiguous array of k model parameters from offset *off* in that worker's model update U. Each of these initial packets is assigned sequentially to one of the s aggregation slots.

After the initial batch of packets is sent, each worker enters a loop where it awaits the aggregated results from the switch. Each packet received indicates that the switch has completed the aggregation of a particular slot. The worker then consumes the result carried in the packet, copying that packet's vector into the aggregated model update A at the offset carried in the packet ($p.off$). The worker then sends a new packet containing the *next* piece of update to be

[5] For simplicity, we show a simple counter to detect this condition. Later, we use a bitmap to track which workers have sent updates.

Algorithm 2 Worker logic.

```
for i in 0 : s do
    p.idx ← i
    p.off ← k · i
    p.vector ← U[p.off : p.off + k]
    send p
repeat
    receive p(idx, off, vector)
    A[p.off : p.off+k] ← p.vector
    p.off ← p.off + k · s
    if p.off < size(U) then
        p.vector ← U[p.off : p.off + k]
        send p
until A is incomplete
```

aggregated. This reuses the same pool slot as the one just received, but contains a new set of k parameters, determined by advancing the previous offset by $k \cdot s$.

A key advantage of this scheme is that it does not require any explicit coordination among workers and yet achieves agreement among them on which pools to use for which parameters. The coordination is implicit because the mapping between model updates, slots and packets is deterministic. Also, since pool index and offset are carried in each packet, the scheme is not influenced by packet reorderings. A simple checksum can be used to detect corruption and discard corrupted packets.

This communication scheme is self-clocked after the initial s packets. This is because a slot cannot be reused until all workers have sent their contribution for the parameter update for the slot. When a slot is completed, the packets from the switch to the workers serve as flow-control acknowledgment that the switch is ready to reuse the slot, and the workers are free to send another packet. Workers are synchronized based on the rate at which the system aggregates model updates. The pool size s determines the number of concurrent in-flight aggregations; as we discuss in §3.6, the system achieves peak bandwidth utilization when $k \cdot s$ (more precisely, $b \cdot s$ where b is the packet size – 180 bytes in our setting) exceeds the bandwidth-delay product of the inter-server links.

3.5 Dealing with packet loss

Thus far, we have assumed packets are never lost. Of course, packet loss can happen due to either corruption or network congestion. With the previous algorithm, even a single packet loss would halt the system. A packet loss on the "upward" path from workers to the switch prevents the switch from completing aggregation of one of the pieces, and the loss of one of the result packets that are multicast on the "downward" paths not only prevents a worker from learning the result, but also prevents it from ever completing *A*.

We tolerate packet loss by retransmitting lost packets. In order to keep switch dataplane complexity low, packet loss detection is done by the workers if they do not receive a response packet from the switch in a timely manner. However, naïve retransmission creates its own problems. If a worker retransmits a packet that was actually delivered to the switch, it can cause a model update to be applied twice to the aggregator. On the other hand, if a worker retransmits a packet for a slot that was actually already fully aggregated (e.g., because the response was lost), the model update can be applied to the wrong data because the slot could have already been reused by other workers who received the response correctly. Thus, the challenges are (1) to be able to differentiate packets that are lost on the upward paths versus the downward ones; and (2) to be able to retransmit an aggregated response that is lost on the way back to a worker.

Algorithm 3 Switch logic with packet loss recovery.

```
Initialize State:
    n = number of workers
    pool[2, s], count[2, s], seen[2, s, n] := {0}
upon receive p(wid, ver, idx, off, vector)
    if seen[p.ver, p.idx, p.wid] = 0 then
        seen[p.ver, p.idx, p.wid] ← 1
        seen[(p.ver+1)%2, p.idx, p.wid] ← 0
        count[p.ver, p.idx] ← (count[p.ver, p.idx]+1)%n
        if count[p.ver, p.idx] = 1 then
            pool[p.ver, p.idx] ← p.vector
        else
            pool[p.ver, p.idx] ← pool[p.ver, p.idx] + p.vector
        if count[p.ver, p.idx] = 0 then
            p.vector ← pool[p.ver, p.idx]
            multicast p
        else
            drop p
    else
        if count[p.ver, p.idx] = 0 then
            p.vector ← pool[p.ver, p.idx]
            forward p to p.wid
        else
            drop p
```

We modify the previous algorithms to address these issues by keeping two additional pieces of state at the switch. First, we explicitly maintain information as to which workers have already contributed updates to a particular slot. This makes it possible to ignore duplicate transmissions. Second, we maintain a *shadow copy* of the *previous* result for each slot. That is, we have two copies or versions of each slot in two pools that are used in alternate phases. This allows the switch to retransmit a dropped result packet for a slot even when the switch has started reusing the slot for the next phase.

The key insight behind the correctness of this approach is that, even in the presence of packet losses, our self-clocking strategy ensures that no worker node can ever lag *more than one phase* behind any of the others. That is, packet loss – specifically, the loss of a downward-path result packet – can cause a pool slot to have been reused only up to *once* before any given node finishes its aggregation, but not twice. This is guaranteed because, according to Algorithm 2, the worker who didn't receive the result packet will not reuse the yet-to-be-reduced (in its own view) slot and hence will not generate a new update packet with the same slot id (i.e., *idx*). Meanwhile, according to Algorithm 1, the switch will not release a slot until it receives an update packet from every single worker for the slot. As a result, it is sufficient to keep only one shadow copy.

To see that this is true, observe that a pool slot is only reused for phase i after updates have been received from all workers for phase $i - 1$. As discussed above, this does not guarantee that all workers have received the result from phase $i - 1$. However, a worker will only send its update for phase i after receiving the result from phase $i - 1$, preventing the system from moving on to phase $i + 1$ until all workers have completed phase $i - 1$.

Besides obviating the need for more than one shadow copy, this has a secondary benefit: the switch does not need to track full phase

Algorithm 4 Worker logic with packet loss recovery.

```
for i in 0 : s do
    $p.wid \leftarrow$ Worker ID
    $p.ver \leftarrow 0$
    $p.idx \leftarrow$ i
    $p.off \leftarrow$ k · i
    $p.vector \leftarrow$ U[$p.off$ : $p.off$ + k]
    send $p$
    start_timer($p$)
repeat
    receive $p(wid, ver, idx, off, vector)$
    cancel_timer($p$)
    A[$p.off$ : $p.off$+k] $\leftarrow p.vector$
    $p.off \leftarrow p.off$ + k · s
    if $p.off$ < size(U) then
        $p.ver \leftarrow$ ($p.ver$+1)%2
        $p.vector \leftarrow$ U[$p.off$ : $p.off$ + k]
        send $p$
        start_timer($p$)
until A is incomplete

Timeout Handler:
    upon timeout $p$
        send $p$
        start_timer($p$)
```

numbers (or offset); a single bit is enough to distinguish the two active phases for any slot.

In keeping with our principle of leaving protocol complexity to end hosts, the shadow copies are kept in the switch but in fact managed entirely by the workers. The switch simply exposes the two pools to the workers, and the packets specify which slot acts as the active copy and which as the shadow copy by indicating a single-bit pool version (*ver*) field in each update packet. The pool version starts at 0 and alternates each time a slot with the same *idx* is reused.

Algorithms 3 and 4 show the details of how this is done. An example illustration is in Appendix A. In the common case, when no losses occur, the switch receives updates for slot *idx*, pool *ver* from all workers. When workers receive the result packet from the switch, they change pool by flipping the *ver* field – making the old copy the shadow copy – and send the next phase updates to the other pool.

Packet loss is detected by a timeout at each worker. When this occurs, the worker does not know whether the switch received its previous packet or not. Regardless, it retransmits its previous update with the same slot *idx* and *ver* as before. This slot is guaranteed to contain the state for the same aggregation in the switch. The *seen* bitmask indicates whether the update has already been applied to the slot. If the aggregation is already complete for a slot, and the switch yet receives an update packet for the slot, the switch recognizes the packet as a retransmission and replies with a unicast packet containing the result. The result in one slot is overwritten for reuse only when there is certainty that all the workers have received the aggregated result in that slot. This happens when all the workers have sent their update to the same slot of the other pool, signaling that they all moved forward. Note this scheme works because the completion of aggregation for a slot *idx* in one pool *safely and unambiguously* confirms that the previous aggregation result in the shadow copy of slot *idx* is indeed received by every worker.

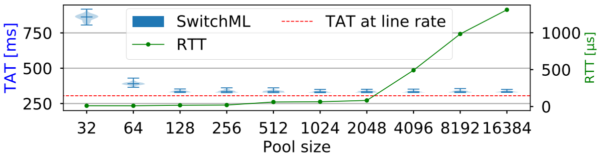


Figure 2: Effect of pool size on overall tensor aggregation time (TAT) and per-packet RTT (right y-axis).

The main cost of this mechanism is switch memory usage: keeping a shadow copy doubles the memory requirement, and tracking *seen* bitmask adds additional cost. This may appear problematic, as on-switch memory is a scarce resource. In practice, however, the total number of slots needed – tuned based on the network bandwidth-delay product (§3.6) – is much smaller than the switch's memory capacity.

3.6 Tuning the pool size

As mentioned, the pool size s has an effect on performance and reliability. We now analyze how to tune this parameter.

Two factors affect s. First, because s defines the number of in-flight packets in the system that originate from a worker, to avoid wasting each worker's network bandwidth, s should be no less than the bandwidth-delay product (BDP) of each server. Note the delay here refers to the end-to-end delay, including the end-host processing time, and this can be easily measured in a given deployment. Let b be the packet size, which is constant in our setting. To sustain line rate transmission, the stream of response packets must arrive at line rate. This occurs when $s \cdot b$ matches the BDP. A higher value of s, when used as the initial window size, will unnecessarily increase queuing time within the workers.

Second, a correctness condition of the communication scheme presented in the previous section requires that no two in-flight packets from the same worker use the same slot (as no worker node can ever lag behind by more than one phase). To sustain line rate and preserve correctness, the lower bound on s is such that $s \cdot b$ matches the BDP. Therefore, the optimal s is for $\lceil BDP/b \rceil$.

In practice, we select s as the next power of two of the above formula because the DPDK library – which we use to implement SwitchML – performs batched send and receive operations to amortize system overheads. Based on our measurements (Figure 2), we use 128 and 512 as the pool size for 10 and 100 Gbps, respectively. This occupies 32 KB and 128 KB of register space in the switch, respectively. We note that the switch can support two orders of magnitude more slots, and SwitchML uses much less than 10% of that available.

3.7 Dealing with floating points

DNN training commonly uses floating-point numbers, but current Ethernet switches do not natively support them. We explored two approaches to bridging this gap.

Floating point numbers are already an approximation. SGD and similar algorithms are defined over real numbers. Floating point numbers approximate the real numbers by trading off range, precision, and computational overhead to provide a numerical representation that can be broadly applied to applications with widely different properties. However, many other approximations are possible. One designed for a specific application can obtain acceptable accuracy with lower overhead than standard floating-point offers.

In recent years, the community has explored many specialized numerical representations for DNNs. These representations exploit the properties of the DNN application domain to reduce the cost of communication and computation. For instance, NVidia Volta GPUs [13] include mixed-precision (16-/32-bit) TPUs that can train with accuracy matching full-precision approaches. Other work has focused on gradient exchange for SGD, using fixed-point quantization, dithering, or sparsification to reduce both the number of bits or gradient elements transmitted [6, 7, 36, 41, 51, 59]. This innovation will continue; our goal is not to propose new representations but to demonstrate that techniques like those in the literature are practical with high-speed programmable switches.

We explore two numerical representations for gradient exchange: the first implements 16-bit floating point in the switch; the second uses 32-bit fixed point in the switch, with conversion done on the workers. In the former case, the switch actually converts each 16-bit floating-point values in the incoming model updates into a 32-bit fixed-point and then performs aggregation. When generating responses, the switch converts fixed-point values back into equivalent floating-point values. All other data (weights, activations) remains in 32-bit floating point format. Both approaches require gradients to be scaled to avoid truncation. Following [40], we profile the gradients and choose a scaling factor so that the maximum value is still representable. We verify experimentally for a number of models that *this quantization allows training to similar accuracy in a similar number of iterations* as an unquantized network for *a large range of scaling factors*. Appendix C discusses this issue in detail.

The two approaches have different properties. The 16-bit floating point implementation reduces bandwidth demand between the workers and the switch (and thus roughly halves the tensor aggregation time as shown later §5), but consumes more switch resources in terms of lookup tables and arithmetic units. The 32-bit fixed-point representation uses minimal switch resources and equivalent bandwidth as existing techniques, but adds conversion overhead on workers. With vector SSE/AVX instructions, this overhead is negligible.

4 IMPLEMENTATION

We build SwitchML as a drop-in replacement of the Gloo (based off commit revision a9c74563) [22] library, which is used in Caffe2 (0.8.2) and PyTorch (1.0). We also integrate SwitchML within Horovod (0.15.2) to support TensorFlow (1.12.0). SwitchML is implemented as a worker component written as ∼3,100 LoCs in C++ and a switch component realized in P4 [8] with ∼4,000 LoCs, but a significant fraction of the P4 code corresponds to logic that is repeated across stages and tables. The worker component is built atop Intel DPDK (18.11). Here, we highlight a few salient aspects of our implementation. Appendix B describes it in more detail.

Our P4 program is carefully optimized to distribute processing of 32 elements per packet over the multiple stages of the ingress pipeline, while minimizing data dependencies to ensure line rate processing. It exploits the traffic manager subsystem to send multiple copies of result packets. On the worker side, we process each packet in a run-to-completion fashion and scale to multiple cores using DPDK flow director. This gives good performance because we can easily shard slots and chunks of tensors across cores without any shared state. The ML framework invokes our synchronous all-reduce API whenever model updates are ready. In practice, model updates consist of a set of tensors that are aggregated independently but sequentially. We are careful to ensure processing is NUMA aware and we use SSE/AVX instructions to efficiently perform tensor scaling and data type conversion.

5 EVALUATION

We analyze the performance benefits of SwitchML using standard benchmarks on popular models in TensorFlow and Caffe2 and using microbenchmarks to compare it to state-of-the-art collective communications libraries and PS-like strategies. Since our testbed is limited to 16 nodes (8 of which only are equipped with GPUs) and 1 switch, our evaluation should be viewed as an initial validation of SwitchML, one within the means of academic research. An extended validation would require deploying SwitchML in production environments and exposing it to ML training in-the-wild.

5.1 Experimental setup

We conduct most of our experiments on a testbed of 8 machines, each with 1 NVidia P100 16 GB GPU, dual 10-core CPU Intel Xeon E5-2630 v4 at 2.20GHz, 128 GB of RAM, and 3 x 1 TB disks for storage (as single RAID). To demonstrate scalability with 16 nodes, we further use 8 machines with dual 8-core Intel Xeon Silver 4108 CPU at 1.80 GHz but without GPUs. Both machine types have dual Intel 82599ES and Mellanox Connect-X 5 NICs, capable of 10 Gbps and 100 Gbps, respectively. Moreover, we use a 64x100 Gbps programmable switch with Barefoot Networks' Tofino chip [4]. The machines run Ubuntu (Linux kernel 4.4.0-122), Mellanox OFED 4.4-2.0.7.0, and NVidia CUDA 9.0. CPU frequency scaling was disabled. We use 4 CPU cores per worker. This introduces a penalty gap at 100 Gbps; but due to a bug in our Flow Director setup we are unable to use more cores. This means that our results at 100 Gbps are a lower bound.

We mostly focus on three performance metrics. We define *tensor aggregation time (TAT)* as the time to aggregate a tensor starting from the time a worker is ready to send it till the time that worker receives the aggregated tensor. Clearly, lower TAT is better. As we generally observe little influence of tensor size on the processing rate, in some cases, we report *aggregated tensor elements (ATE)* per unit of time, for presentation clarity. When reporting on the above metrics, we collect measurements at each worker for aggregating 100 tensors of the same size and report statistics as violin plots, which also highlight the statistical median, min, and max values. We also consider *training throughput* defined in terms of the numbers of images processed per second by the training process. To measure throughput, we run an experiment for 500 iterations. We observed that throughput stabilizes after the first ∼100 iterations, which we therefore exclude, and we report throughput from the later iterations.

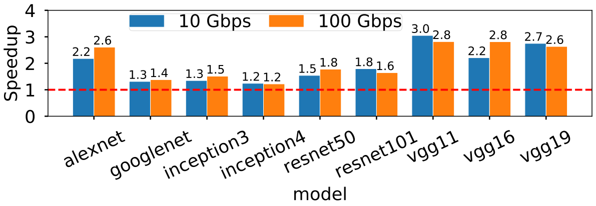


Figure 3: Training performance speedup on a 10 (blue) and 100 (orange) Gbps network with 8 workers. Results are normalized to NCCL (fastest TensorFlow baseline with direct GPU memory access).

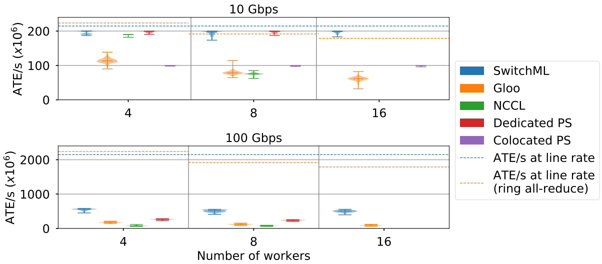


Figure 4: Microbenchmarks of aggregated tensor elements per time unit on a 10 (top) and 100 (bottom) Gbps network as workers increase. As 8 machines do not have GPUs, we run NCCL up to 8 workers. Dedicated PS is up to 8 workers because it then uses all 16 machines.

For evaluating training performance, we adopt the standard TensorFlow benchmarking suite [56], which contains implementations of several popular convolutional models. We train models over the standard Imagenet dataset, loaded from disk during training, using a mini-batch size of 128 unless otherwise stated (Table 1 models use 64). For AlexNet, we follow the suggestions in [55]; we use synthetic data and 512 as mini-batch size.

To obtain a fair comparison, we use TCP as the transport protocol in the communication libraries Gloo and NCCL (2.3.7), unless otherwise specified. We also perform experiments with RDMA. Our default setup is to run experiments on 8 workers on a 10 Gbps network using our suite of microbenchmarks with 100 MB tensors.

As discussed earlier, we pick scaling factors such that quantization does not introduce precision loss. The models converge to accuracies within state-of-the-art results. We verify that, over a broad range of scaling factors (5 orders of magnitude in some cases), the choice of scaling factor does not influence training throughput.

5.2 SwitchML improves training speed

We first analyze training performance on the benchmark suite of popular models by accelerating training in TensorFlow and Caffe2. As baselines, we choose to normalize results based on NCCL for TensorFlow as this is the configuration that achieves the highest training throughput. We note, however, that NCCL is an NVidia CUDA library that benefits from directly accessing GPU memory – a type of HW acceleration that SwitchML does not currently use. Gloo's training performance is a more faithful baseline, since SwitchML is built integrated with Gloo. Thus, we normalize with respect to Gloo performance in the context of Caffe2.

Figure 3 shows the training performance speedup with TensorFlow relative to NCCL. We see similar speedups with Caffe2 relative to Gloo. Blue and orange bars refer to speedups on a 10 and 100 Gbps network, respectively. Overall, SwitchML's speedups range between 20%-300%. As expected, different models benefit from in-network aggregation differently, depending on the extent to which they are network bound. The absolute training throughput is reported in Table 1 for select models. We note that the baseline throughput numbers are comparable to publicized benchmark results [55].

5.3 Tensor aggregation microbenchmarks

To illustrate SwitchML's efficiency in comparison to other communication strategies, we devise a microbenchmark that performs a number of continuous tensor aggregations, without performing any actual gradient computation on GPU. We verify that the tensors – initially, all ones – are aggregated correctly. We test with various tensor sizes from 50 MB to 1.5 GB. We observe that the number of aggregated tensor elements per time unit is not influenced by the tensor size. (We also include some results later that SwitchML remains close to line rate while tensor size increases.) Therefore, we report results for 100 MB tensors only.

For these experiments, we benchmark SwitchML against the previously mentioned all-reduce communication libraries (Gloo and NCCL), and we further compare against an efficient PS-like scenario, i.e., a set of processes that assist with the aggregation by centralizing updates at a shared location. To this end, we build a multi-core DPDK-based program that implements the logic of Algorithm 1. To capture the range of possible PS performance, we consider two scenarios: (1) when the PS processes run on dedicated machines, effectively doubling the cluster size, and (2) when a PS process is co-located with every worker. We choose to run as many PS processes (each using 4 cores) as workers so that tensor aggregation workload is equally spread among all machines (uniformly sharded) and avoids introducing an obvious performance bottleneck due to oversubscribed bandwidth, which is the case when the ratio of workers to PS nodes is greater than one.

Figure 4 shows the results as we vary the number of workers from 4 to 16. In addition to performance, we plot the highest theoretically achievable rate based on the maximum goodput, given the line rate (10 and 100 Gbps), for a given packet payload size and communication strategy. The results demonstrate very high efficiency of SwitchML. In every condition, SwitchML outperforms all other strategies. Moreover, SwitchML always maintains a predictable rate of ATE/s regardless of the number of workers. We believe this would also apply with more workers, up to 64 in our testbed.

The Dedicated PS approach matches SwitchML performance but uses twice the number of machines *and* network capacity. Unsurprisingly, using the same number of machines as SwitchML, the Colocated PS approach reaches only half of SwitchML's performance. Although our PS implementation is technically not representative of a traditional PS (as we do not store the entire model in memory), it serves to highlight that, in the limit, our aggregation

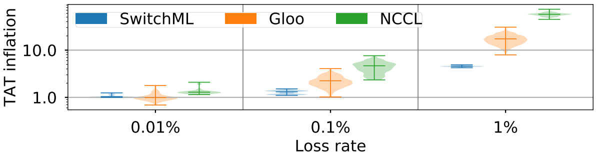


Figure 5: Inflation of TAT due to packet loss and recovery. Results are normalized to a baseline scenario where no loss occurs and the worker implementation does not incur any timer-management overhead.

protocol could be deployed to run entirely in SW. (We discuss an alternate deployment model in §6.) However, in-network aggregation inherently requires fewer resources than host-based aggregation.

5.4 Can SwitchML be faster than RDMA?

Where possible, all-reduce implementations make use of RDMA transport to speed up communication. In our setting, we observed a sensible 4x speedup exchanging 50MB tensors with Gloo at 100Gbps using RDMA versus TCP. RDMA's advantage is its support for remote memory access with no CPU involvement, improving latency and throughput.

RDMA and SwitchML are not mutually exclusive: it would be possible to extend SwitchML to use the RDMA protocols, but at a cost. RDMA treats reliability in terms of pairwise connections, with the NIC storing 100's of bytes per connection [42]. SwitchML would have to use storage and execution resources to check sequence numbers and timeouts, whereas the current design keeps that complexity on the worker. Furthermore, RDMA retransmission and flow control are also pairwise, whereas in SwitchML they are collective.

5.5 Overheads

Packet loss recovery. We study how packet loss affects TAT. We note that in our experiments we do not observe any packet loss. To quantify the change in TAT due to packet loss, we experiment with a uniform random loss probability between 0.01% and 1% applied on every link. The retransmission timeout is set to 1ms. We run microbenchmark experiments in similar scenarios as §5.3. We report a few representative runs.

Figure 5 measures the inflation in TAT with different loss probabilities. SwitchML completes tensor aggregation significantly faster than Gloo when the loss is 0.1% or higher. A loss probability of 0.01% only minimally affects TAT in either case. To better illustrate the behavior of SwitchML, we show in Figure 6 the evolution of packets sent per 10 ms at a representative worker for 0.01% and 1% loss. We observe that SwitchML generally maintains a high sending rate – relatively close to the ideal rate – and quickly recovers by retransmitting dropped packets. The slowdown past the 150 ms mark with 1% loss occurs because some slots are unevenly affected by random losses and SwitchML does not apply any form of work-stealing to rebalance the load among aggregators. This presents a further opportunity for optimization that we leave for future work.

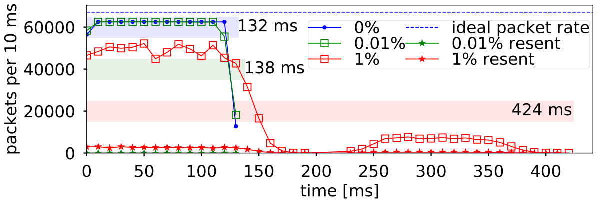


Figure 6: Timeline of packets sent per 10 ms during an aggregation with 0%, 0.01% and 1% packet loss probability. Horizontal bars denote the TAT in each case.

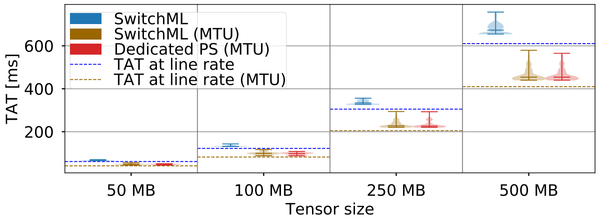


Figure 7: TAT of SwitchML compared to an enhanced baseline that emulates SwitchML using MTU-sized packets and to a dedicated PS setting using MTU-sized packets.

Limited payload size. As described, we are limited to 32 elements per packet. We now quantify the performance overhead as compared to the ideal case where MTU-sized packets would carry 366 elements (1516-byte packets, including all headers). Despite the hardware limits, we can still provide an upper bound of how a switch capable of processing MTU-sized packets might perform by having the switch in our testbed process just the first 32 elements and forward the remaining payload bytes as is. We compare SwitchML to this enhanced baseline as well as to the Dedicated PS benchmark using MTU-sized packets, which serves to illustrate the effects of increased per-packet SW processing costs (albeit amortized by having fewer packets to process). Figure 7 shows the performance comparison with using MTU-sized packets across a range of tensor sizes. The results illustrate that SwitchML pays only a modest performance cost due to using packets that are an order-of-magnitude smaller.

On the other hand, a switch capable of processing MTU-sized packets would enable SwitchML to increase the goodput by reducing from 28.9% to 3.4% the overhead of packet headers and further improve TAT by 31.6%, suggesting an alternate deployment model (§6).

Tensor scaling and type conversion. We analyze whether any performance overheads arise due to the tensor scaling operations (i.e., multiply updates by f and divide aggregates by f) and the necessary data type conversions: float32-to-int32 → htonl → ntohl → int32-to-float32.

To quantify overheads, we use int32 as the native data type while running the microbenchmarks. This emulates a native float32 scenario with no scaling and conversion operations. We also illustrate the potential improvement of quantization to single-precision (float16) tensors, which halves the volume of data to be sent to the

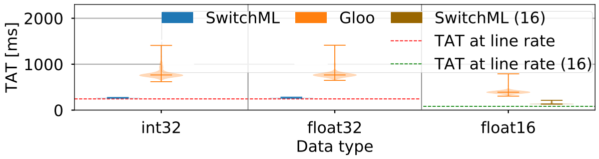


Figure 8: TAT comparison between aggregating native-integer tensors, scaling and converting from single-precision (float32) tensors, and scaling and converting from half-precision (float16) tensors.

network. (We include a conversion from/to float32.) This setting is enabled by the ability to perform at line rate, in-switch type conversion (float16 ↔ int32), which we verified with the switch chip vendor. However, for this experiment, we only emulate this by halving the tensor size (shown as SwitchML (16)). We further compare with the performance of Gloo in the same settings.

Figure 8 shows how these operations affect TAT. We notice that the overheads are negligible. We attribute this to the implementation, which uses x86 SSE/AVX instructions. Whereas, using float16 doubles the performance, as expected.

Switch resources. We comment on SwitchML's usage of switch resources in relation to network bandwidth and number of workers. As discussed in Appendix B, our implementation only makes use of the switch ingress pipeline in maximizing the number of vector elements that is processed at line rate. Aggregation bandwidth affects the required pool size. We verified that even at 100 Gbps the memory requirement is << 10% of switch resources. The number of workers does not influence the resource requirements to perform aggregation at line rate. That said, the number of switch ports and pipelines obviously pose a cap on how many directly-attached workers are supported. A single pipeline in our testbed supports 16-64 workers depending on network speed. We describe how to move beyond a single rack scale in the next section.

6 DISCUSSION AND LIMITATIONS

Scaling beyond a rack. We described SwitchML in the context of a rack. However, large scale ML jobs could span beyond a single rack. We envision that SwitchML can be extended to work across multiple racks by hierarchically composing several instances of our switch logic. Let i be the switch layer in the hierarchy, with $i = 1$ being the rack layer, and $i = H$ being the root layer. Workers are connected to layer-1 switches. Further, let each switch contain multiple packet processing pipelines with p being the size of the group of ports that can be aggregated together within a switch using a single pipeline. In this setting, a layer-i switch aggregates tensors from d downstream ports and forwards partial aggregates to layer-$(i+1)$ switch(es) via $u = \lceil \frac{d}{p} \rceil$ upstream ports (assuming each upstream port has at least as much bandwidth as a worker's link to the ToR). In other words, the switch operates as u virtual switches, one for each pipeline inside of the switch. The root switch completes the aggregation of partial aggregates and multicasts downstream on the same ports used as upstreams by its children in the hierarchy. When a layer-$(j < H)$ switch receives a complete aggregate, it multicasts downstream over d ports. Finally, the rack-layer switch multicasts the aggregates to the workers. While we are unable to test this approach due to testbed limitations, we provide a few interesting observations below.

This hierarchical composition is bandwidth-optimal in the sense that it allows all n workers in a ML job to fully occupy the host line rate while supporting all-to-all communication with a bandwidth cost proportional to u instead of n. Because every switch aggregates data in a $p : 1$ ratio, this approach can also work with oversubscribed networks at the aggregation or core layers, provided the oversubscription ratio is no worse than the above ratio. In the limit, however, a very large n (~256-1024) coupled with a relatively small p (~16-32) would require a hierarchy with $H > 3$. We note that one can instead shard the tensor over multiple root switches to scale using a "fatter" hierarchy.

The described packet loss recovery algorithm will work also in a multi-rack scenario. In fact, thanks to the bitmap and the shadow copy, a retrasmission originated from a worker would be recognized as a retrasmission on switches that have already processed that packet. This retransmission can trigger the retransmission of the updated value toward the upper layer switch, so that the switch affected by the loss is always reached. We leave a full exploration of this scenario to future work.

Extrapolating performance. As mentioned, our evaluation is within the reach of an academic testbed. Based on the above discussion on scaling to many more workers, we conjecture – using extrapolation from our results – that SwitchML will see even greater performance improvements on larger jobs. Key to this is that the tensor aggregation time does not depend on first order on the number of workers n.

Lack of congestion control. Given that we currently operate at rack-scale, we did not have to deal explicitly with congestion. We note that, given the tight coupling between a worker's communication loop and pool size, our flow control scheme can react to congestion because the system would self-clock to the rate of the slowest worker. If the downlink from the switch to a worker is congested, it reduces the bandwidth of the aggregation results that the worker is able to receive and in turn reduces the bandwidth at which the worker contributes new parameters to the switch for aggregation. This self-clocking mechanism therefore reduces the sending rate of all workers and alleviates congestion. One should, however, take care to adapt the retransmission timeout according to variations in end-to-end RTT. It is also worth noting that the self-clocking mechanism is also effective at slowing down the system in the presence of stragglers.

Deployment model. Thus far, we presented SwitchML as an in-network computing approach, focusing on the mechanisms to enable efficient aggregation of model updates at line rate on programmable switching chips with very limited memory. While that might be a viable deployment model in some scenarios, we highlight that our design may have more ample applicability. In fact, one could use a similar design to create a dedicated "parameter aggregator," i.e., a server unit that combines a programmable switching chip with a typical server board, CPU and OS. Essentially a standard server with an advanced network attachment, or in the limit, an array of programmable Smart NICs like the Netronome Agilio-CX, each hosting a shard of aggregator slots. The switch component of SwitchML would run on said network attachment. Then, racks could be equipped with such a parameter aggregator, attached for example to the legacy ToR using several 100 Gbps or 400 Gbps ports, or via a dedicated secondary network within the rack directly linking worker servers with it. We expect this would provide similar performance improvements while giving more options for deployment configurations.

Multi-job (tenancy). In multi-job or multi-tenant scenarios, the question arises as to how to support concurrent reductions with SwitchML. The solution is conceptually simple. Every job requires a separate pool of aggregators to ensure correctness. As discussed, the resources used for one reduction are much less than 10% of switch capabilities. Moreover, modern switch chips comprise multiple independent pipelines, each with its own resources. Thus, an admission mechanism would be needed to control the assignment of jobs to pools.

Encrypted traffic. Given the cluster setting and workloads we consider, we do not consider it necessary to accommodate for encrypted traffic. Appendix D expands on this issue.

Asynchronous SGD. As mentioned, we only target synchronous SGD, since it is commonly used in the cluster setting to enable reproducibility.

7 RELATED WORK

Our proposal draws inspiration from and creates new linkages between several research efforts across networking, big data systems and, of course, machine learning.

In-network computation trends. The trend towards programmable data planes recently boosted by the reconfigurable match table (RMT) architecture [9], the P4 programming language [8], and commercial availability of protocol-independent switch architecture (e.g., Tofino [4]), has sparked a surge of proposals to offload certain application-specific yet broadly-useful primitives into network devices.

These primitives include consensus algorithms [15, 16, 30], caching [31, 37], and multi-sequencing for concurrency control [34]. In common with these works, we too find it challenging to live within the constraints of today's programmable switches. Our technical solution is distinct from any of these works due to our focus on data aggregation.

In-network aggregation. We are not the first to propose aggregating data in the network. Targeting partition-aggregate and big data (MapReduce) applications, NetAgg [39] and CamDoop [14] demonstrated significant performance advantages, by performing application-specific data aggregation at switch-attached high-performance middleboxes or at servers in a direct-connect network topology, respectively. Parameter Hub [38] does the same with a rack-scale parameter server. Historically, some specialized supercomputer networks [2, 19] offloaded MPI collective operators (e.g., all-reduce) to the network. Certain Mellanox Infiniband switches support collective offloads through SHArP [24], which builds a reduction tree embedding laid over the actual network topology. Operators are applied to data as it traverses the reduction tree; however, a tree node's operator waits to execute until all its children's data has been received. SwitchML differs from all of these approaches in that it performs in-network data reduction using a streaming aggregation protocol. Moreover, as we operate over Ethernet instead of Infiniband, we develop a failure recovery protocol. SwitchML keeps the network architecture unmodified and exploits programmable switches; its low resource usage implies it can coexist with standard Ethernet switch functionality.

The closest work to ours is DAIET [49]. Sapio at al. also proposed in-network aggregation for minimizing communication overhead of exchanging ML model updates. However, their short paper does not describe a complete design, does not address the major challenges (§3.1) of supporting ML applications, and provides only a simple proof-of-concept prototype for MapReduce applications running on a P4 emulator. It is not clear it could be made to work with a real programmable switch.

Accelerating DNN training. A large body of work has proposed improvements to hardware and software systems, as well as algorithmic advances to train DNN models faster. We only discuss a few relevant prior approaches. Improving training performance via data or model parallelism has been explored by numerous deep learning systems [1, 10, 11, 17, 35, 38, 58]. Among the two strategies, data parallelism is the most common approach; but it can be advantageous to devise strategies that combine the two. Recent work even shows how to automatically find a fast parallelization strategy for a specific parallel machine [29]. Underpinning any distributed training strategy, lies parameter synchronization. Gibiansky was among the first to research [21] using fast collective algorithms in lieu of the PS approach, which has been a traditional mechanism in many ML frameworks. This approach is now commonly used in many platforms [21, 25, 27, 50, 52]. Following this line of work, we view SwitchML as a further advancement – one that pushes the boundary by co-designing networking functions with ML applications.

8 CONCLUSION

This paper presents SwitchML, a deep learning system that speeds up DNN training by minimizing communication overheads at single-rack scale. SwitchML uses in-network aggregation to efficiently synchronize model parameters at each training iteration among distributed workers executing in parallel. We evaluate SwitchML with nine real-world DNN benchmarks on a GPU cluster with 10 Gbps and 100 Gbps networks; we show that SwitchML achieves training throughput speedups up to 300% and is consistently better than state-of-the-art collective communications libraries.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill,

F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An Overview of the BlueGene/L Supercomputer. In *SC*, 2002.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-Efficient SGD via Randomized Quantization. In *NIPS*, 2017.

[4] Barefoot Networks. Tofino. https://barefootnetworks.com/products/brief-tofino/.

[5] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (InterCom). In *SHPCC*, 1994.

[6] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signSGD: Compressed Optimisation for Non-Convex Problems. In *ICML*, 2018.

[7] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar. signSGD with Majority Vote is Communication Efficient And Byzantine Fault Tolerant. *CoRR*, abs/1810.05291, 2018. http://arxiv.org/abs/1810.05291.

[8] 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.*, 44(3), July 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Workshop on Machine Learning Systems*, 2016.

[11] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.

[12] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017.

[13] J. Choquette, O. Giroux, and D. Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2), 2018.

[14] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI*, 2012.

[15] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2), 2016.

[16] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *SOSR*, 2015.

[17] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[18] EC2Instances.info Easy Amazon EC2 Instance Comparison. https://www.ec2instances.info/?region=us-west-2.

[19] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. Mpi collective communications on the blue gene/p supercomputer: Algorithms and optimizations. In *IEEE Symposium on High Performance Interconnects*, 2009.

[20] O. Fercoq, Z. Qu, P. Richtárik, and M. Takáč. Fast distributed coordinate descent for minimizing non-strongly convex losses. *IEEE International Workshop on Machine Learning for Signal Processing*, 2014.

[21] A. Gibiansky. Effectively Scaling Deep Learning Frameworks. http://on-demand.gputechconf.com/gtc/2017/presentation/s7543-andrew-gibiansky-effectively-scakukbg-deep-learning-frameworks.pdf.

[22] Gloo. https://github.com/facebookincubator/gloo.

[23] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR*, abs/1706.02677, 2017. http://arxiv.org/abs/1706.02677.

[24] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COM-HPC*, 2016.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*, 2018.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[27] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. In *CVPR*, 2016.

[28] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Technical report, Microsoft Research, 2018. https://www.microsoft.com/en-us/research/uploads/prod/2018/05/gpu_sched_tr.pdf.

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

[30] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*, 2018.

[31] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[33] 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.

[34] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *SOSP*, 2017.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[36] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training, 2018. https://openreview.net/pdf?id=SkhQHMW0W.

[37] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ASPLOS*, 2017.

[38] 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.

[39] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-Specific On-path Aggregation in Data Centres. In *CoNEXT*, 2014.

[40] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. In *ICLR*, 2018.

[41] K. Mishchenko, E. Gorbunov, M. Takáč, and P. Richtárik. Distributed Learning with Compressed Gradient Differences. *CoRR*, abs/1901.09269, 2019. http://arxiv.org/abs/1901.09269.

[42] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting Network Support for RDMA. In *SIGCOMM*, 2018.

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

[44] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.

[45] A. Nemirovski and D. B. Yudin. *Problem complexity and method efficiency in optimization.* Wiley Interscience, 1983.

[46] P. Patarasuk and X. Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[47] P. Richtárik and M. Takáč. Distributed coordinate descent method for learning with big data. *Journal of Machine Learning Research*, 17(75):1–25, 2016.

[48] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[49] 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.

[50] F. Seide and A. Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[51] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs. In *Interspeech*, 2014.

[52] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR*, abs/1802.05799, 2018. http://arxiv.org/abs/1802.05799.

[53] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: from theory to algorithms.* Cambridge University Press, 2014.

[54] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[55] TensorFlow benchmarks. https://www.tensorflow.org/guide/performance/benchmarks.

[56] TensorFlow benchmarks. https://github.com/tensorflow/benchmarks.

[57] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1), Feb. 2005.

[58] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics. In *SoCC*, 2015.

[59] 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 *NIPS*, 2017.

[60] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Speeding up ImageNet Training on Supercomputers. In *SysML*, 2018.

A EXAMPLE EXECUTION

We illustrate through an example how Algorithms 3 and 4 behave in a system with three workers: w1, w2, and w3. We focus on the events that occur for a particular slot (x) starting from a particular offset (*off*).

- **t0:** Worker w1 sends its model update for slot x with offset = *off*.
- **t1:** Worker w2 sends its model update for slot x with offset = *off*.
- **t2:** Worker w3 sends its model update for slot x with offset = *off*. This update packet is lost on the upstream path at time **t3**, and hence the switch does not receive it.
- **t4:** w1's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet, but it ignores the update because it already received and aggregated an update from w1 for the given slot and offset.
- **t5:** w2's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet, but it ignores the update because it already received and aggregated an update from w2 for the given slot and offset.
- **t6:** w3's timeout kicks in for the model update sent at **t0**, leading to a retransmission of the same model update for slot x with offset = *off*. The switch receives the packet and aggregates the update properly. Since this update is the last one for slot x and offset *off*, the switch completes aggregation for the slot and offset, turns the slot into a shadow copy, and produces three response packets (shown as blue arrows).
- **t7:** The first response packet for w1 is lost on the downstream path, and w1 does not receive it.
- **t8:** Not having received the result packet for the update packets sent out earlier (at **t0** and **t4**), w1 retransmits its model update the second time. This retransmission reaches the switch correctly, and the switch responds by sending a unicast response packet for w1.
- **t9** and **t10:** w2 and w3 respectively receives the response packet. Hence, w2 and w3 respectively decides to reuse slot x for the next offset ($off + k \cdot s$) and sends their new updates at **t12** and **t13**.
- **t11:** The unicast response packet triggered by the second model-update retransmission (sent at **t8**) arrives at w1.
- **t14:** Now that w1 has received its response, it decides to reuse slot x for the next offset ($off + k \cdot s$) and sends its new updates. This update arrives at the switch at **t15**, upon which the switch realizes that the slot for offset ($off + k \cdot s$) is complete. This confirms that the result in the shadow-copy slot (the slot in pool 0) is safely received by every worker. Thus, the switch flips the roles of the slots again.

B IMPLEMENTATION DETAILS

Switch component. The main challenge we faced was to find a design that best utilizes the available resources (SRAM, TCAMs, hashing machinery, etc.) to perform as much computation per packet as possible. Data plane programs are typically constrained by either available execution resources or available storage; for SwitchML, execution resources are the tighter constraint. For example, a data plane program is constrained by the number of stages per pipeline [54], which limits the dependencies within the code. In fact, every action whose execution is contingent on the result of a previous operation has to be performed on a subsequent stage. A program with too many dependencies cannot find a suitable allocation on the hardware pipeline and will be rejected by the compiler. Moreover, the number of memory accesses per-stage is inherently limited by the maximum per-packet latency; a switch may be able to parse more data from a packet than it is able to store into the switch memory during that packet's time in the switch.

We make a number of design trade-offs to fit within the switch constraints. First, our P4 program makes the most use of the limited memory operations by performing the widest memory accesses possible (64 bits). We then use the upper and lower part of each register for alternate pools. These parts can execute different operations simultaneously; for example, when used for the received work bitmap, we can set a bit for one pool and clear a bit for the alternate pool in one operation. Second, we minimize dependencies (e.g., branches) in our Algorithm 3 in order to process 32 elements per packet within a single ingress pipeline. We confine all processing to the ingress pipeline; when the aggregation is complete, the traffic manager duplicates the packet containing the aggregated result and performs a multicast. In a first version of our program, we used both ingress and egress pipelines for the aggregation, but that required packet recirculation to duplicate the packets. This caused additional dependencies that required more stages, preventing the processing of more than 32 elements per packets. Moreover, this design experienced unaccounted packet losses between the two pipelines and during recirculation, which led us to search for a better, single pipeline, program.

Worker component. Our goal for implementing the worker component is to achieve high I/O performance for aggregating model updates. At the same time, we want to support existing ML frameworks without modifications.

In existing ML frameworks, a DNN model update U comprises of a set of tensors T, each carrying a subset of the gradients. This is because the model consists of many layers; most existing frameworks emit a gradient tensor per layer and reduce each layer's tensors independently. Back-propagation produces the gradients starting from the output layer and moving towards the input layer. Thus, communication can start on the output layer's gradients while the other gradients are still being computed, partially overlapping communication with computation. This implies that for each iteration, there are as many aggregation tasks as the number of tensors (e.g., 152 for ResNet50 in Caffe2).

Our implementation exposes the same synchronous all-reduce interface as Gloo. However, rather than treating each tensor as an independent reduction and resetting switch state for each one, our implementation is efficient in that it treats the set of tensors

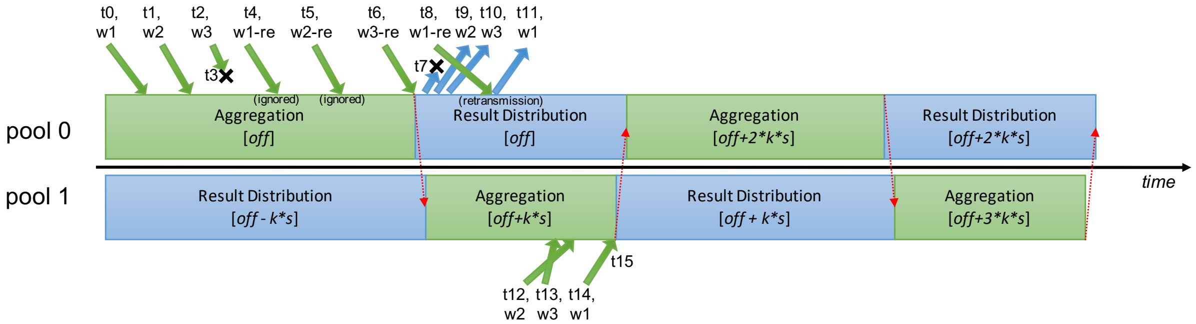


Figure 9: An example execution of a SwitchML switch interacting with three workers. The figure illustrates how a slot with index x is used during the different phases (shown in different colors) that alternate between the two pools.

virtually as a single, continuous stream of data across iterations. Upon invocation, our API passes the input tensor to a virtual stream buffer manager which streams the tensor to the switch, breaking it into the small chunks the switch expects. Multiple threads may call SwitchML's all-reduce, with the requirement that each worker machine's tensor reduction calls must occur in the same order; the stream buffer manager then performs the reductions and steers results to the correct requesting thread. This ordering requirement matches Horovod's design, whereas we modify one line of code in Caffe2 to enforce this.[6]

One CPU core is sufficient to do reduction at line rate on a 10 Gbps network. However, to be able to scale to 100 Gbps, we use multiple CPU cores at each worker and use the Flow Director technology (implemented in hardware on modern NICs) to uniformly distribute incoming traffic across the NIC RX queues, one for each core. Every CPU core runs an I/O loop that processes every batch of packets in a run-to-completion fashion and uses a disjoint set of aggregation slots. Packets are batched in groups of 32 to reduce per-packet transmission overhead. To improve cache locality, when using more than one core, we partition the tensor into as many contiguous memory regions as the number of cores. We use x86 SSE/AVX instructions to scale the model updates and convert between types. We are careful to ensure all processing is NUMA aware.

C MODEL QUANTIZATION

To the best of our knowledge, no Ethernet switching chip offers floating-point operations in the dataplane for packet processing. Some InfiniBand switching chips have limited support for floating-point operations for scientific computing [24]. We also confirmed that the state-of-the-art programmable Ethernet switching chips do not support native floating-point operations either. These observations lead us to two main questions.

Where should the type conversion occur? Either the conversion takes place at end hosts or at the switch. In the former case, packets carry a vector of integer types while in the latter case the switch internally performs the type conversions. It turns out to be possible to implement 16-bit floating point conversion on a Barefoot Network's Tofino chip using lookup tables. This means there is no type conversion overhead at end hosts. At the same time, an efficient implementation that uses modern x86 vector instructions (SSE/AVX) to implement type conversion sees only a negligible overhead (see Figure 8). Thus, both options are practical.

What are the implications in terms of accuracy? Recently, several update compression (e.g., quantization, dithering or sparsification) strategies were proposed to be used with standard training methods, such as SGD, and bounds were obtained on how these strategies influence the number of iterations until a sufficient convergence criterion is satisfied (e.g., being sufficiently close to minimizing the empirical loss over the data set). These include aggressive 1-bit compression of SGD for DNNs [51], signSGD [6, 7], QSGD [3], which uses just the sign of the stochastic gradients to inform the update, Terngrad [59], which uses ternary quantization, and the DIANA framework [41], which allows for a wide array of compression strategies applied to gradient differences. All the approaches above use lossy randomized compression strategies that preserve unbiasedness of the stochastic gradients at the cost of increasing the variance of gradient estimators, which leads to worse iteration complexity bounds. Thus, there is a trade-off between savings in communication and the need to perform more training rounds.

To the best of our knowledge, no existing approaches to compression work by replacing floats by integers, which is what we do here. In addition, our compression mechanism is not randomized, and for a suitable selection of a scaling parameter f, is essentially lossless or suffers negligible loss only.

We shall now briefly describe our compression mechanism. Each worker multiplies its model update $\Delta_i^t = \Delta(x^t, D_i^t)$ by a fixed scaling factor $f > 0$, obtaining $f\Delta_i^t$. This factor is chosen to be not too large so that all d entries of the scaled update can be rounded to a number representable as an integer without overflow. We then perform this rounding, obtaining vectors $Q_i^t = \rho(f\Delta_i^t) \in \mathbb{Z}^d$ for $i = 1, 2, \ldots, n$, where ρ is the rounding operator, which are sent to the switch and aggregated, resulting in

$$A^t = \sum_{i=1}^{n} Q_i^t.$$

[6] By default Caffe2 runs 16-way parallel aggregations, which we allow when running as a baseline for comparison.

Again, we need to make sure f is not too large so that A^t can be represented as an integer without overflow. The aggregated update A^t is then scaled back, obtaining A^t/f, and the model gets updated as follows:

$$x^{t+1} = x^t + \frac{1}{f}A^t.$$

Let us illustrate this on a simple example with $n = 2$ and $d = 1$. Say $\Delta_1^t = 1.56$ and $\Delta_2^t = 4.23$. We set $f = 100$ and get

$$Q_1^t = \rho(f\Delta_1^t) = \rho(156) = 156$$

and

$$Q_2^t = \rho(f\Delta_2^t) = \rho(423) = 423.$$

The switch will add these two integers, which results in $A^t = 579$. The model then gets updated as:

$$x^{t+1} = x^t + \frac{1}{100}579 = x^t + 5.79.$$

Notice that while the aggregation was done using integers only (which is necessitated by the limitations of the switch), the resulting model update is identical to the one that would be applied without any conversion in place. Let us consider the same example, but with $f = 10$ instead. This leads to

$$Q_1^t = \rho(f\Delta_1^t) = \rho(15.6) = 16$$

and

$$Q_2^t = \rho(f\Delta_2^t) = \rho(42.3) = 42.$$

The switch will add these two integers, which results in $A^t = 58$. The model then gets updated as:

$$x^{t+1} = x^t + \frac{1}{10}58 = x^t + 5.8.$$

Note that this second approach leads to a small error. Indeed, while the true update is 5.79, we have applied the update 5.8 instead, incurring the error 0.01.

Our strategy is to apply the above trick, but take special care about how we choose the scaling factor f so that the trick works throughout the entire iterative process with as little information loss as possible.

A formal model. Let us now formalize the above process. We first assume that we have a scalar $f > 0$ for which the following holds:

Assumption 1. $|\rho(f\Delta_i^t)| \le 2^{31}$ for all $i = 1, 2, \ldots, n$ and all iterations t.

Assumption 2. $|\sum_{i=1}^n \rho(f\Delta_i^t)| \le 2^{31}$ for all iterations t.

The above assumptions postulate that all numbers which we obtain by scaling and rounding on the nodes (Assumption 1), and by aggregation on the switch (Assumption 2), can be represented as integers without overflow.

We will now establish a formal statement which characterizes the error incurred by our aggregation procedure.

Theorem 1 (Bounded aggregation error). The difference between the exact aggregation value $\sum_{i=1}^n \Delta_i^t$ (obtained in case of perfect arithmetic without any scaling and rounding, and with a switch that can aggregate floats) and the value $\frac{1}{f}A^t = \frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t)$ obtained by our procedure is bounded by $\frac{n}{f}$.

Proof. To prove the above result, notice that

$$\begin{aligned}\frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t) &\le \frac{1}{f}\sum_{i=1}^n \lceil f\Delta_i^t \rceil \\ &\le \frac{1}{f}\sum_{i=1}^n (f\Delta_i^t + 1) \\ &= \left(\sum_{i=1}^n \Delta_i^t\right) + \frac{n}{f}.\end{aligned}$$

Using the same argument, we get a similar lower bound

$$\begin{aligned}\frac{1}{f}\sum_{i=1}^n \rho(f\Delta_i^t) &\ge \frac{1}{f}\sum_{i=1}^n \lfloor f\Delta_i^t \rfloor \\ &\ge \frac{1}{f}\sum_{i=1}^n (f\Delta_i^t - 1) \\ &= \left(\sum_{i=1}^n \Delta_i^t\right) - \frac{n}{f}.\end{aligned}$$

□

Note that the error bound postulated in Theorem 1 improves as f increases, and n decreases. In practice, the number of nodes is constant $n = O(1)$. Hence, it makes sense to choose f as large as possible while making sure Assumptions 1 and 2 are satisfied. Let us give one example for when these assumptions are satisfied. In many practical situations it is known that the model parameters remain bounded:[7]

Assumption 3. There exists $B > 0$ such that $|\Delta_i^t| \le B$ for all i and t.

As we shall show next, if Assumption 3 is satisfied, then so is Assumption 1 and 2.

Theorem 2 (No overflow). Let Assumption 3 be satisfied. Then Assumptions 1 and 2 are satisfied (i.e., there is no overflow) as long as $0 < f \le \frac{2^{31}-n}{nB}$.

Proof. We have $\rho(f\Delta_i^t) \le f\Delta_i^t + 1 \le f|\Delta_i^t| + 1 \le fB + 1$. Likewise, $\rho(f\Delta_i^t) \ge f\Delta_i^t - 1 \ge -f|\Delta_i^t| - 1 = -(fB + 1)$. So, $|\rho(f\Delta_i^t)| \le fB + 1$. Hence, as soon as $0 < f \le \frac{2^{31}-1}{B}$, Assumption 1 is satisfied. This inequality is less restrictive as the one we assume. Similarly, $|\sum_i \rho(f\delta_i^t)| \le \sum_i |\rho(f\Delta_i^t)| \le \sum_i (fB + 1) = n(fB + 1)$. So, Assumption 2 is satisfied as long as $n(fB + 1) \le 2^{31}$, i.e., as long as $0 < f \le \frac{2^{31}-n}{nB}$. □

We now put all of the above together. By combining Theorem 1 (bounded aggregation error) and Theorem 2 (no overflow), and if we choose $f = \frac{2^{31}-n}{nB}$, then the difference between the exact update $\sum_i \Delta_i^t$ and our update $\frac{1}{f}\sum_i \rho(f\Delta_i^t)$ is bounded by $\frac{n^2 B}{2^{31}-n}$. In typical applications, $n^2 B \ll 2^{31}$, which means that the error we introduce is negligible.

Experimental verification. We verified experimentally for a number of models that the above relations hold, and are not significantly influenced by the training iteration. Figure 10 shows this for one network, GoogLeNet, trained on the ImageNet dataset. Training

[7] If desirable, this can be enforced explicitly by the inclusion of a suitable hard regularizer, and by using projected SGD instead of plain SGD.

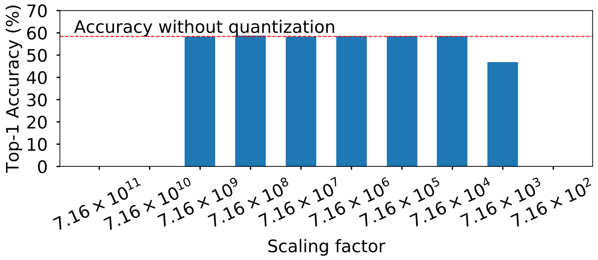


Figure 10: GoogleNet trained with quantization achieves similar accuracy to training with no quantization. The largest gradient value observed in the first 5000 iterations with no quantization was 29.24. Training was stopped after 30 epochs due to time constraints.

was stopped after 30 epochs due to time constraints. The maximum gradient value found in the first 5000 iterations without quantization was 29.24; quantization factors that bring this value close to the maximum 32-bit integer value supported accurate training, while smaller and larger ones caused training to diverge. We also conducted similar experiments on the CIFAR10 dataset with similar conclusions. Thus, it is relatively easy to pick an appropriate f by considering just the first few iterations of a ML job; moreover, this selection could be automated.

D ENCRYPTED TRAFFIC

A recent trend, especially at cloud providers, is to encrypt all datacenter traffic. In fact, data encryption is generally performed at the NIC level itself. While addressing this setting is out of scope, we wish to comment on this aspect. We believe that given our substantial performance improvements, one might simply forego encryption for ML training traffic.

We envision a few alternatives for when that is not possible. One could imagine using HW accelerators to enable in-line decryption/re-encryption at switches. However, that is likely costly. Thus, one may wonder if computing over encrypted data at switches is possible. While arbitrary computations over encrypted data are beyond current switches' capabilities, we note that the operation performed at switches to aggregate updates is simple integer summation. The appealing property of several partially homomorphic cryptosystems (e.g., Paillier) is that the relation $E(x) \cdot E(y) = E(x + y)$ holds for any two values x, y (E denotes encryption). By customizing the end-host encryption process, the worker could encrypt all the vector elements using such cryptosystem, knowing that the aggregated model update can be obtained by decrypting the data aggregated at the switches.