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 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 an efficient solution that speeds up training by up to 5.5× for a number of real-world benchmark models.

1 Introduction

Today's machine learning (ML) solutions' remarkable success derives from the ability to build increasingly sophisticated models on increasingly large data sets. To cope with the resulting increase in training time, ML practitioners use distributed training [1, 22]. Large-scale clusters use hundreds of nodes, each equipped with multiple GPUs or other hardware accelerators (e.g., TPUs [48]), to run training jobs on tens of workers that take many hours or days.

Distributed training is increasingly a *network-bound* workload. To be clear, it remains computationally intensive. But the last seven years have brought a 62× improvement in compute performance [64, 78], thanks to GPUs [74] and other hardware accelerators [11, 34, 35, 48]). Cloud network deployments have found this pace hard to match, skewing the ratio of computation to communication towards the latter. Since parallelization techniques like mini-batch stochastic gradient descent (SGD) training [37, 43] alternate computation with synchronous model updates among workers, network performance now has a substantial impact on training time.

Can a new type of accelerator *in the network* alleviate the network bottleneck? We demonstrate that an *in-network aggregation* primitive can accelerate distributed ML workloads, and can be implemented using programmable switch hardware [5, 10]. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput, diminishes latency, and speeds up training time.

Building an in-network aggregation primitive 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 that the switch can 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. Finally, the in-network aggregation primitive is an all-to-all primitive, unlike traditional unicast or multicast communication patterns. As a result, in-network aggregation requires mechanisms for synchronizing workers and detecting and recovering from 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 scoreboard 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 in switch dataplanes by having the workers scale and convert floating-point values to fixed-point using an adaptive scaling factor with negligible approximation loss.

*Equal contribution. Amedeo Sapio is affiliated with Barefoot Networks, but was at KAUST during much of this work.

SwitchML integrates with distributed ML frameworks, such as PyTorch and TensorFlow, to accelerate their communication, and enable efficient training of deep neural networks (DNNs). Our initial prototype targets a *rack-scale architecture*, where a single switch centrally aggregates parameter updates from serviced workers. Though the single switch limits scalability, we note that commercially-available programmable switches can service 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 [32, 43, 50, 98].

We show that SwitchML's in-network aggregation yields end-to-end improvements in training performance of up to 5.5× for popular DNN models. Focusing on a communication microbenchmark, compared to the best-in-class collective library NCCL [77], SwitchML is up to 2.9× faster than NCCL with RDMA and 9.1× than NCCL with TCP. While the magnitude of the performance improvements is dependent on the neural network architecture and the underlying physical network speed, 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 Horovod [89] and NCCL [77], which support several popular toolkits like TensorFlow and PyTorch. SwitchML is openly available at https://github.com/p4lang/p4app-switchML.

2 Network bottlenecks in ML training

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) or one of its many variants (e.g., using momentum, mini-batching, importance sampling, preconditioning, variance reduction) [72, 73, 83, 90]. 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.

The key to data parallelism is that each worker i, in parallel, locally computes the update $\Delta(x^t, D_i^t)$ to the model parameters based on the current model x^t and a mini-batch, i.e., a subset of the local data D_i^t. Typically, a model update contributed by worker i is a multiple of the stochastic gradient of the loss function with respect to the current model parameters x^t computed across a mini-batch of training data, D_i^t. Subsequently, workers communicate their updates, which are aggregated ($\sum$) and added to x^t to form the model parameters of the next iteration. Importantly, each iteration acts only on a mini-batch of the training data. It requires many iterations to progress through the entire dataset, which constitutes a training epoch. A typical training job requires multiple epochs, reprocessing the full training data set, until the model achieves acceptable error on a validation set.

From a networking perspective, the challenge is that data-parallel SGD requires computing the sum of model updates across all workers after every iteration. Each model update has as many parameters as the model itself, so they are often in 100s-of-MB or GB range. And their size is growing exponentially: today's largest models exceed 32 GB [84]. These aggregations need to be performed frequently, as increasing the mini-batch size hurts convergence [66]. Today's ML toolkits implement this communication phase in one of two ways:

The parameter server (PS) approach. In this approach, workers compute model updates and send them to *parameter servers* [45, 56, 64]. These servers, usually dedicated machines, aggregate updates to compute and distribute the new model parameters. To prevent the PS from becoming a bottleneck, the model is sharded over multiple PS nodes.

The all-reduce approach. An alternate approach uses the workers to run an all-reduce algorithm – a collective communication technique common in high-performance computing – to combine model updates. The workers communicate over an overlay network. A *ring* topology [6], where each worker communicates to the next neighboring worker on the ring, is common because it is bandwidth-optimal (though its latency grows with the number of workers) [79]. *Halving and doubling* uses a binary tree topology [93] instead.

2.2 The network bottleneck

Fundamentally, training alternates compute-intensive phases with communication-intensive model update synchronization. Workers produce intense bursts of traffic to communicate their model updates, whether it is done through a parameter server or all-reduce, and training stalls until it is complete.

Recent studies have shown that performance bottleneck in distributed training is increasingly shifting from compute to communication [64]. This shift comes from two sources. The first is a result of advances in GPUs and other compute accelerators. For example, the recently released NVIDIA A100 offers 10× and 20× performance improvements for floating-point

[1] In this paper, we do not consider model-parallel training [28, 82], although that approach also requires efficient networking. Further, we focus exclusively on widely-used distributed synchronous SGD [1, 37].

[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 (step size) and model averaging inside Δ.

and mixed-precision calculations, respectively [74] compared to its predecessor, the V100 – released just 2.5 years previously. This pace far exceeds advances in network bandwidth: a 10× improvement in Ethernet speeds (from 10 Gbps to 100 Gbps) required 8 years to standardize.

Second, the ratio of communication to computation in the workload itself has shifted. The current trend towards ever-larger DNNs generally exacerbates this issue. However, this effect is highly application-dependent. In popular ML toolkits, communication and computation phases can partially overlap. Since back-prop proceeds incrementally, communication can start as soon as the earliest partial results of back-prop are available. The effectiveness of this technique depends on the structure of the DNN. For DNNs with large initial layers, its effectiveness is marginal, because there is little to no opportunity to overlap communication with computation.

When is the network a bottleneck? To answer this quantitatively, we profile the training of 8 common DNNs on a cluster with 8 workers using NVIDIA P100 GPUs. To precisely factor out the contribution of communication to the processing time of a mini-batch, we emulate communication time at 10 Gbps or 100 Gbps Ethernet assuming transmission at line rate. We record the network-level events, which allows us to report the fraction of time spent in communication as well as how much can overlap with computation (Table 1). At 10 Gbps, all but three workloads spend more than 50% of their time in communication, usually with little computation-phase overlap. These workloads benefit greatly from 100 Gbps networking, but even so communication remains a significant share (at least 17%) of batch processing time for half of the workloads.

What happens when GPUs become faster? Our profile uses P100 GPUs, now two generations old. Faster GPUs would reduce the computation time, increasing the relative communication fraction. Our measurement of non-overlappable communication time allows us to determine the scaling factor α applied to GPU computation time at which point the network is saturated. There is still some speedup beyond an $\alpha\times$ faster GPU, but it is limited to the initial phase, before communication begins. Note $\alpha < 4$ for half the workloads (Table 1), suggesting that network performance will be a serious issue when using the latest GPUs with a 100 Gbps network.

Model	Size	Batch	10 Gbps		100 Gbps		
	[MB]	size	Batch [ms]	Comm [%]	Batch [ms]	Comm [%]	α
DeepLight	2319	2^{13}	2101 ± 1.4	97% (2%)	258 ± 0.4	79% (20%)	1.0
LSTM	1627	64	1534 ± 8.3	94% (10%)	312 ± 6.8	46% (56%)	1.5
BERT	1274	4	1677 ± 7.1	67% (3%)	668 ± 3.1	17% (35%)	3.5
VGG19	548	64	661 ± 1.9	73% (67%)	499 ± 1.1	10% (99%)	6.7
UGATIT	511	2	1612 ± 2.5	28% (84%)	1212 ± 3.5	4% (99%)	17.6
NCF	121	2^{17}	149 ± 0.6	72% (4%)	46 ± 0.1	23% (27%)	1.2
SSD	98	16	293 ± 0.6	26% (99%)	293 ± 1.6	3% (99%)	15.2
ResNet-50	87	64	299 ± 10.9	29% (67%)	270 ± 1.2	3% (94%)	19.8

Table 1: Profile of benchmark DNNs. "Batch [ms]" reports the average batch processing time and its standard deviation. "Comm" reports the proportion of communication activity as % of batch time. The figure in parentheses is the percentage of *that* time that overlaps with computation. For example, DeepLight at 10 Gbps spends 97% of its batch time in communication; only 2% of this 97% communication overlaps with computation. The table lists a scaling factor for an hypothetical $\alpha\times$ faster GPU that implies communication is contiguous and saturates the 100 Gbps bandwidth once communication begins.

3 In-network aggregation

We propose an alternative approach to model update exchange for ML workloads: *in-network aggregation*. In this approach, workers send their model updates over the network, where an aggregation primitive in the network sums the updates and distributes only the resulting value. Variations on this primitive have been proposed, over the years, for specialized supercomputer networks [2, 26] and InfiniBand [33]. We demonstrate that it is possible to realize in-network aggregation in an Ethernet network and benefit ML applications.

In-network aggregation offers a fundamental advantage over both all-reduce and PS. It achieves the minimum possible latency and the minimum communication cost, quantified in data volume each worker sends and receives: $2|U|$ bytes, where $|U|$ is the total number of bytes to be aggregated. This is a significant improvement over the equivalent cost for bandwidth-optimal all-reduce, which is $4|U|\frac{n-1}{n}$ [79]. The PS approach can match this communication cost of $2|U|$ bytes, at the expense of more resource cost; in the limit, it doubles the number of required machines and network bandwidth.[4] Regardless of resource costs, in-network aggregation avoids end-host processing required to perform aggregation and, therefore, provides "sub-RTT" latency [46], which the contrasted approaches cannot achieve.

Illustrating the advantages of in-network aggregation. To characterize the extent to which communication is a bottleneck for training performance, we use our profile of eight DNN models from §2.2. We evaluate the impact of communication performance using a trace of network-level events recorded during training. This trace captures real compute times and memory access latency, including the latency for barrier events that precede each synchronization, but allows us to emulate different network speeds and computation patterns. In particular, our trace records the detailed timing of individual all-reduce invocations, so it faithfully accounts for potential overlap between communication and computation.[5]

We compare the performance of in-network aggregation (INA) with the current best practice, ring all-reduce (RAR). Table 2 summarizes the batch processing speedup over the

[4] If the PS nodes are co-located with the worker nodes, then the effective bandwidth per node is halved, doubling latency.

[5] The ML toolkit adopts an optimization known as *tensor fusion* or *bucketing* that coalesces multiple all-reduce invocations to amortize setup overhead. Our traces reflect the effect of this optimization.

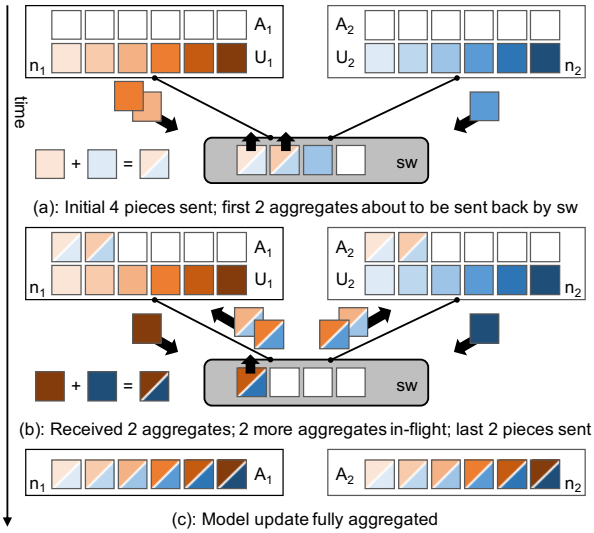

Figure 1: Example of in-network aggregation of model updates. U_i is the model update computed by worker i. Workers stream pieces of model updates in a coordinated fashion. In the example, each workers can have at most 4 outstanding packets at any time to match the slots in the switch. The switch aggregates 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.

ring all-reduce performance. INA is consistently superior to RAR. For communication-bound models (the four models in the 100 Gbps case), INA is up to 80% and up to 67% faster at 10 and 100 Gbps, respectively. Note that this analysis reflects a *theoretically optimal* implementation of RAR. The measured speedups (§6) of our real INA implementation are higher, because real RAR implementations do not achieve optimal performance; it is difficult to fully exploit all available bandwidth and avoid system overheads.

We also note that our profiling environment uses NVIDIA P100 devices. These are currently two-generation old GPU accelerators. We investigate with real benchmarks in §6 the impact of faster GPUs, which increases the relative impact of communication overheads.

Alternative: gradient compression. Another way to reduce communication costs is to reduce the data volume of model updates using lossy compression. Proposed approaches include reducing the bit-width of gradient elements (quantization) or transmitting only a subset of elements (sparsification). These approaches come with tradeoffs: too much compression loss can impact the resulting model accuracy.

We adopt the results of a recent survey of gradient compression methods [96] to emulate the behavior of Top-k [3] and QSGD [4] as two representative sparsification and quantization compressors, respectively. We use data from that study to identify the compression overhead and data reduction achieved. Our synthetic communication time, then, includes both the computational cost of compression and the communication cost of the all-gather operation used to exchange model updates (following their implementation [96]).

Model	**INA**	QSGD		Top-k	
		64	256	1%	10%
10 Gbps					
DeepLight	1.80	1.27	0.97	9.24 (-1.1%)	1.05 (-0.9%)
LSTM	1.77	1.27	0.97	7.49	1.05
NCF	1.54	1.22	0.96	4.07	1.05 (-2.2%)
BERT	1.54	1.20	0.98	3.45 (†)	1.04 (†)
VGG19	1.60	1.22	0.97	2.13 (-10.4%)	1.04 (-3.3%)
UGATIT	1.22	1.12	0.99	1.58	1.02
ResNet-50	1.05	1.07	0.95	1.15 (-1.7%)	1.02 (+0.2%)
SSD	1.01	1.00	1.00	1.01 (-2.4%)	1.00 (-0.6%)
100 Gbps					
DeepLight	↗1.67	0.93	0.78	2.96 (-1.1%)	0.47 (-0.9%)
LSTM	↗1.20	0.98	0.84	1.37	0.54
NCF	1.22	1.00	0.85	1.22	0.65 (-2.2%)
BERT	↗1.14	0.98	0.92	1.27 (†)	0.74 (†)

† The BERT task is fine-tuning from a pre-trained model, for which compression does not have a noticeable impact. The impact during pretraining is analyzed in Appendix E.

Table 2: Analysis of batch processing speedup relative to ring all-reduce based on synthetic communication. For Top-k compression, impact on model quality is shown in parentheses. Accuracy penalties greater than 1% are shaded in gray; red indicates failure to converge. At 100 Gbps, only the models that are network bottlenecked are shown. ↗ indicates 100 Gbps cases where SwitchML achieves a higher batch processing speedup due to practical system overheads.

We observe (Table 2) that, although gradient compression decreases data volume, it is not necessarily superior to INA. In general, the computational cost of compression and decompression is non-negligible [58,96]; in some cases, it outweighs the communication-reduction benefits. In particular, INA outperforms QSGD on all workloads for both the 64 and 256 levels (6 and 8 bits). Similarly, Top-k underperforms INA at the 10% compression level, and even *reduces* performance relative to RAR in the 100 Gbps setting. These observations agree with recent work [58, 96]. In particular, Li et al. [58] proposed additional hardware offloading, using an FPGA at every worker, to mitigate compression costs. As this requires additional hardware, our analysis does not consider it.

Gradient compression *does* outperform INA when it can achieve high compression ratios, as with Top-k at 1%. However, in many cases, this level of compression either requires more training iterations to converge, or hurts the accuracy of the resulting model [96]. For example, the NCF model achieves 95.8% hit rate without compression after 20 epochs of training, while with Top-k compression at 10% it achieves 93.6%. It fails to converge at 1% compression. We report convergence comparisons for various models in Appendix D.

4 Design

Our system, SwitchML, implements the aggregation primitive in a programmable dataplane switch. Such switches are now

commercially available, with only a small cost premium compared to fixed-function switches [5]. In-network aggregation is conceptually straightforward, but implementing it inside a programmable switch, however, is challenging. Although programmable switches allow placing computation into the network path, their limited computation and storage capabilities impose constraints on implementing gradient aggregation. The system must also tolerate packet loss, which, although uncommon in the rack-scale 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.

4.1 Challenges

Limited computation. Mathematically, gradient 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. Further, the operations themselves can only be simple integer arithmetic/logic operations; neither floating-point nor integer division operations are possible.

Limited storage. Model updates are large. In each iteration, each worker may supply hundreds of megabytes of gradient values. This volume far exceeds the on-switch storage capacity, 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 [10], given that speed considerations require dataplane-accessible storage to be implemented using on-die SRAM.

Packet loss. SwitchML must be resilient to packet loss, without impact on efficiency or correctness (e.g., discarding part of an update or applying it twice because of retransmission).

4.2 SwitchML overview

SwitchML aims to alleviate communication bottlenecks for distributed ML training applications using in-network aggregation, in a practical cluster setting.[6] SwitchML uses the following techniques to reduce communication costs while meeting the above challenges.

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

[6]For simplicity, 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, 56].

Algorithm 1 Switch logic.

```
1:  Initialize State:
2:      n = number of workers
3:      pool[s], count[s] := {0}
4:  upon receive p(idx, off, vector)
5:      pool[p.idx] ← pool[p.idx] + p.vector        {+ is the vector addition}
6:      count[p.idx]++
7:      if count[p.idx] = n then
8:          p.vector ← pool[p.idx]
9:          pool[p.idx] ← 0; count[p.idx] ← 0
10:         multicast p
11:     else
12:         drop p
```

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 lightweight 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 using a block floating-point-like approach [25], which is done efficiently at end hosts without impacting training accuracy.

We now describe each of these components in turn. To ease the presentation, we describe a version of the system in which packet losses do not occur. We remove this restriction later.

4.3 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*. As with the all-reduce approach, we leave the final division step to the end hosts, as the switch cannot efficiently perform this.

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 ($p.idx$). Once the slot has

aggregated vectors from each worker,[7] 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; instead, it 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 [10, 16, 92]. In our deployment, k is 64 or 256.

4.4 Worker-side aggregation protocol

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

There are two considerations in managing the pool of aggregators appropriately. 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 there is no packet loss. 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 values 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 awaits the aggregated results from the switch. Each packet received indicates that the switch has completed the aggregation of a particular slot. The worker 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 with the *next* piece of update to be 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 $k \cdot s$.

Algorithm 2 Worker logic.

```
1:  for i in 0 : s do
2:      p.idx ← i
3:      p.off ← k · i
4:      p.vector ← U[p.off : p.off + k]
5:      send p
6:  repeat
7:      receive p(idx, off, vector)
8:      A[p.off : p.off+k] ← p.vector
9:      p.off ← p.off + k · s
10:     if p.off < size(U) then
11:         p.vector ← U[p.off : p.off + k]
12:         send p
13: until A is incomplete
```

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 slots to use for which parameters. The coordination is implicit because the mapping between model updates, slots, and packets is deterministic. Also, since each packet carries the pool index and offset, the scheme is not influenced by 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 acknowledgments 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 elaborate in Appendix §C, the system achieves peak bandwidth utilization when $k \cdot s$ (more precisely, $b \cdot s$ where b is the packet size – 1100 bytes in our setting) matches the bandwidth-delay product of the inter-server links.

4.5 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 the corresponding parameter aggregations. 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

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

Algorithm 3 Switch logic with packet loss recovery.

```
1:  Initialize State:
2:      n = number of workers
3:      pool[2, s], count[2, s], seen[2, s, n] := {0}
4:  upon receive p(wid, ver, idx, off, vector)
5:      if seen[p.ver, p.idx, p.wid] = 0 then
6:          seen[p.ver, p.idx, p.wid] ← 1
7:          seen[(p.ver+1)%2, p.idx, p.wid] ← 0
8:          count[p.ver, p.idx] ← (count[p.ver, p.idx]+1)%n
9:          if count[p.ver, p.idx] = 1 then
10:             pool[p.ver, p.idx] ← p.vector
11:         else
12:             pool[p.ver, p.idx] ← pool[p.ver, p.idx] + p.vector
13:         if count[p.ver, p.idx] = 0 then
14:             p.vector ← pool[p.ver, p.idx]
15:             multicast p
16:         else
17:             drop p
18:     else
19:         if count[p.ver, p.idx] = 0 then
20:             p.vector ← pool[p.ver, p.idx]
21:             forward p to p.wid
22:         else
23:             drop p
```

Algorithm 4 Worker logic with packet loss recovery.

```
1:  for i in 0 : s do
2:      p.wid ← Worker ID
3:      p.ver ← 0
4:      p.idx ← i
5:      p.off ← k · i
6:      p.vector ← U[p.off : p.off + k]
7:      send p
8:      start_timer(p)
9:  repeat
10:     receive p(wid, ver, idx, off, vector)
11:     cancel_timer(p)
12:     A[p.off : p.off+k] ← p.vector
13:     p.off ← p.off + k · s
14:     if p.off < size(U) then
15:         p.ver ← (p.ver+1)%2
16:         p.vector ← U[p.off : p.off + k]
17:         send p
18:         start_timer(p)
19: until A is incomplete

20: upon timeout p /* Timeout Handler */
21:     send p
22:     start_timer(p)
```

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.

We modify the algorithms to address these issues by keeping two additional pieces of switch state. First, we explicitly maintain information as to which workers have already contributed updates to a given 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, organized in two pools; workers alternate between these two copies to aggregate successive chunks that are assigned to the same slot. The shadow copy allows the switch to retransmit a dropped result packet for a slot even when the switch has started reusing the slot for the next chunk.

The key insight behind this approach's correctness is that, even in the presence of packet losses, our self-clocking strategy ensures that no worker node can ever lag *more than one chunk* behind any of the others for a particular slot. This invariant is because the switch will not release a slot to be reused, by sending a response, until it has received an update packet from *every* worker for that slot. Furthermore, a worker will not send the next chunk for a slot until it has received the response packet for the slot's previous chunk, preventing the system from moving ahead further. As a result, it is sufficient to keep only one shadow copy.

Besides obviating the need for more than one shadow copy, this has a secondary benefit: the switch does not need to track full phase numbers (or offsets); 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 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 response packet from the switch, they change the pool by flipping the *ver* field – making the old copy the shadow copy – and send the next phase updates to the other pool.

A timeout detects packet loss at each worker. When this occurs, the worker does not know whether the switch received its previous packet or not. Regardless, it retransmits its earlier 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 retransmitted packet and replies with a unicast packet containing the result. The result in one slot is overwritten for

reuse only when there is the certainty that all the workers have received the slot's aggregated result. Slot reuse happens when all the workers have sent their updates to the same slot of the other pool, signaling that they have 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* has indeed been received by every worker.

This mechanism's main cost is switch memory usage: keeping a shadow copy doubles the memory requirement, and tracking the *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 (Appendix C) – is much smaller than the switch's memory capacity.

4.6 Dealing with floating-point numbers

DNN training commonly uses floating-point numbers, but current programmable 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 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. An approximation 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 and Ampere GPUs [17, 74] 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 and the gradient elements transmitted [7, 8, 60, 69, 88, 95, 99]. Further, others have explored block floating-point representations [25, 53], where a single exponent is shared by multiple tensor elements, reducing the amount of computation required to perform tensor operations. This innovation will continue (as work [40, 70] that builds upon our architecture demonstrates); our goal is not to propose new representations but to demonstrate that techniques like those in the literature are practical with programmable switches.

We use a numeric representation, inspired by block floating-point, that combines 32-bit fixed-point addition in the switch with adaptive scaling on the workers. This representation is used only when aggregating gradients; all other data (weights, activations) remain in 32-bit floating-point representation.

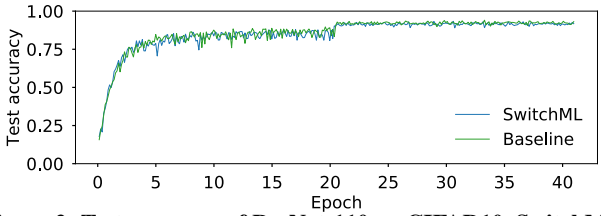

Figure 2: Test accuracy of ResNet-110 on CIFAR10. SwitchML achieves similar accuracy to the baseline.

To implement our representation, we scale gradient values using a per-packet scaling factor f, which is automatically determined for each use of an aggregator slot in the switch. The scaling factor is set so that the maximum aggregated floating point value within a block of k gradients is still representable as a 32-bit fixed point value. Namely, let h be the largest absolute value of a block of gradients; f is set to $(2^{31}-1)/(n \cdot 2^m)$, where m is the exponent of h rounded up to a power of 2 and n is the number of workers. Appendix E formally analyzes the precision of this representation.

To realize this quantization of floating-point values, workers need to agree on a global value of m prior to sending the corresponding block of gradients. We devise a simple look-ahead strategy: when workers send the j-th block to slot i, they include their local block $j+1$'s maximum gradient (rounded up to a power of 2). The switch identifies the global maximum m and piggy-backs that value when sending the aggregated gradients of the j-th block.

We verify experimentally that this communication quantization allows training to similar accuracy in a similar number of iterations as an unquantized network. We illustrate the convergence behavior by training a ResNet-110 model on CIFAR10 dataset for 64,000 steps (about 41 epochs) using 8 workers. Figure 2 shows the test accuracy over time. The accuracy obtained by SwitchML (about 91-93% in the last 5 points) is similar to that obtained by training with TensorFlow on the same worker setup, and it matches prior results [38] with the same hyperparameter settings. The training loss curves (not shown) show the same similarity. In Appendix E, we further give a detailed convergence analysis for the aforementioned representation on models in Table 1.

While the above representation is used in the remainder of the paper, we also explored the implementation of a restricted form of 16-bit floating-point. In this version, the switch converts each 16-bit floating-point value in the incoming packets into a fixed-point value and then performs aggregation. When generating responses, the switch converts fixed-point values back into floating-point values. Due to resource limitations in the switch, we were only able to support half the dynamic range of the 16-bit floating-point format; we expect this to lead to poor convergence during training. Conversely, our 32-bit integer format uses minimal switch resources, provides good dynamic range, and has a minimal overhead on workers. A 16-bit format would provide a bandwidth benefit (§6.3).

5 Implementation

We build SwitchML as a collective library which we integrate in PyTorch's DistributedDataParallel module and in TersorFlow via Horovod [89]. SwitchML is implemented as a worker component written as ∼3,100 LoCs in C++ and a switch component realized in P4 [9] with ∼3,700 LoCs. The worker is built atop Intel DPDK. We have also built a RDMA-capable implementation, but it is not yet integrated with the training frameworks. Here, we highlight a few salient aspects of our implementation. Appendix B describes more details.

Our P4 program distributes aggregation across multiple stages of the ingress pipeline, and also implements flow control, retransmission, and exponent-calculation logic. It uses the traffic manager subsystem to send multiple copies of result packets. It can process 64 elements per packet using one switch pipeline, and 256-element (1024-byte) packets using all four switch pipelines. On the worker side, we process each packet in a run-to-completion fashion and scale to multiple CPU cores using DPDK and Flow Director. We use up to 8 cores per worker. This scales well because we shard slots and chunks of tensors across cores without any shared state. The ML framework invokes our synchronous API whenever model updates are ready. In practice, model updates consist of a set of tensors that are aggregated independently but sequentially.

Supporting large packets. Good bandwidth efficiency requires processing enough integer elements in each packet to offset the network framing overhead. Our P4 program can parse and aggregate 64 × 4-byte elements per packet, but can only read 32 elements per packet when aggregation is complete. With framing overheads, a 32-element payload would limit goodput to 63% of line rate. Our P4 program supports larger packets in two additional configurations for better efficiency: a 64-element configuration with 77% goodput, and a 256-element one with 93% goodput.

We support larger packets through *recirculation*: sending packets through the switch pipelines multiple times. Our 64-element design uses a single pipeline. It makes one additional pass through the pipeline *only* when an output packet is broadcast in order to read the results: this separation of reads and writes allows us to write 64 elements in a single pass. The internal recirculation ports provided by the chip provide sufficient bandwidth. To support 256 elements, we recirculate packets through all four switch pipelines. This requires placing switch ports into loopback mode for more recirculation bandwidth, leaving 16 × 100 Gbps bandwidth available for workers. When a slot is complete, we recirculate again through all the pipelines to read the results. Tofino has sufficient bandwidth to do this recirculation at 1.6 Tbps, and the latency scales deterministically with the number of pipeline passes: we measure an additional 520 ns per pass.

Supporting RDMA. Our host-side framework, even using DPDK and multiple cores, has difficulty achieving 100 Gbps throughput due to packet processing costs. We address this by implementing a subset of RDMA in the switch. This allows workers to offload packet processing: the RDMA NIC breaks large messages into individual packets. Specifically, we use RoCE v2 [42] in Unreliable Connected (UC) mode [67]. This mode, which does not require any of RoCE's link-level flow control mechanisms, supports multi-packet messages and detects packet drops, but does not implement retransmission. SwitchML continues to rely on its existing reliability mechanism. Timeouts and duplicate packets are handled as before, except that a timeout forces a client to retransmit the entire multi-packet message. To balance the benefit of offload with the cost of retransmission, we use small, multi-packet messages (generally 16 packets per message). Although retransmissions are more expensive, the common case is much faster, even though we use a single CPU core.

RDMA Write Immediate messages are used for all communication, allowing data to move directly between the switch and GPUs, with client CPUs handling protocol operations. SwitchML metadata is encoded in RDMA headers. Concurrent messages are sent on separate queue pairs to allow packets to interleave; queue pair IDs and access keys are negotiated with the switch control plane during job setup. The switch sends aggregated results by generating RDMA Write messages to the destination buffer.

6 Evaluation

We analyze the performance benefits of SwitchML by using standard benchmarks on popular models in TensorFlow and PyTorch and by using microbenchmarks to compare it to state-of-the-art collective communications libraries and PS scenarios.

Testbed. 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 E5-2630v4 at 2.20 GHz, 128 GB of RAM, and 3 × 1 TB disks for storage (as single RAID). To demonstrate scalability with 16 nodes, we further use 8 machines with dual 8-core CPU Intel Xeon Silver 4108 at 1.80 GHz. Moreover, we use a Wedge100BF-65X programmable switch with Barefoot Networks' Tofino chip [5]. Every node is networked at both 10 and 100 Gbps.

Performance metrics. We mostly focus on two 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; lower is better. We also report *aggregated tensor elements* (ATE) per unit of time, for presentation clarity; higher is better. For these metrics, we collect measurements at each worker for aggregating 100 tensors of the same size, after 10 warmups. We measure *training throughput* defined in terms of the numbers of training samples processed per second. We measure throughput for 100 iterations that follow 100 warmups. A variant of training throughput is the batch processing throughput,

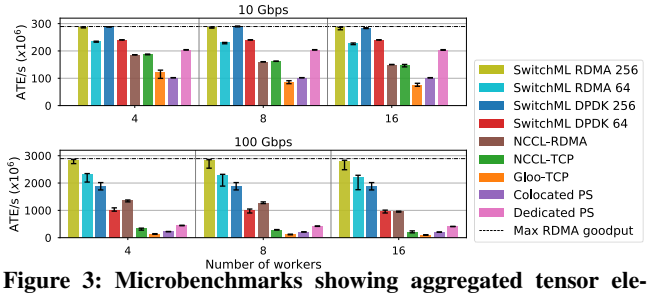

Figure 3: Microbenchmarks showing aggregated tensor elements per second on a 10 (top) and 100 (bottom) Gbps network as workers increase.

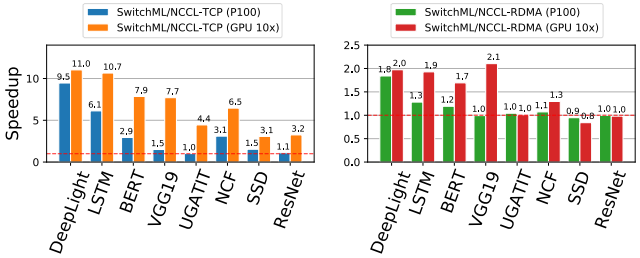

Figure 4: Training batch processing speedup at 100 Gbps considering a P100 GPU and a 10× faster GPU.

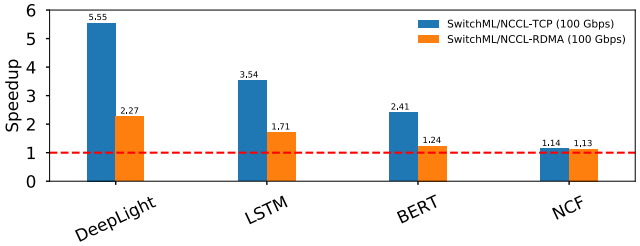

Figure 5: Training performance speedup normalized to NCCL with TCP and RDMA transport protocols.

which we use to analyze performance by replaying profile traces. This throughput metric includes communication and computation costs, but excludes the time to load data.

Benchmarks. We evaluate SwitchML by training with 8 DNNs introduced in Table 1. The detailed configuration of the benchmarks is in Table 3 in Appendix B. Half of the benchmarks execute on PyTorch and half on TensorFlow.

Setup. As a baseline, we run both PyTorch with native distributed data-parallel module and TensorFlow with Horovod. By default, we use NCCL as the communication library, and use both TCP and RDMA as the transport protocol. Our default setup is to run experiments on 8 workers.

6.1 Tensor aggregation microbenchmarks

To illustrate SwitchML's efficiency in comparison to other communication strategies, we devise a communication-only microbenchmark that performs continuous tensor aggregations, without any gradient computation on the 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 (ATE/s) is not very sensitive to the tensor size. Thus, we report results for 100 MB tensors only.

For these experiments, we benchmark SwitchML against the popular all-reduce communication libraries (Gloo [31] and NCCL [77]). We further compare against a parameter server-like scenario, i.e., a set of worker-based processes that assist with the aggregation. To this end, we build a DPDK-based program that implements streaming aggregation as in 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 8 cores) as workers so that the 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 3 shows the results at 10 and 100 Gbps on three cluster sizes. The results demonstrate the efficiency of SwitchML: its highest-performing variant, which uses RDMA with 256-value (1024-byte payload) packets, is within 2% of the maximum achievable goodput. Using smaller packets ($k = 64$ instead of 256) has a noticeable performance impact, underscoring the importance of our multi-pipeline design. The DPDK implementation has additional host-side overhead that prevents it from achieving full link utilization at 100 Gbps. In spite of this, SwitchML can still outperform the best current all-reduce system, NCCL, even when it uses RDMA and SwitchML does not. Moreover, SwitchML always maintains a predictable rate of ATE/s regardless of the number of workers. This trend should continue with larger clusters.

The Dedicated PS approach (with 256 values per packet) – while using *twice* the number of machines *and* network capacity – falls short of matching SwitchML DPDK performance. Unsurprisingly, using the same number of machines as SwitchML, the Colocated PS approach reaches only half of Dedicated PS performance. Our PS implementation is simpler than (and should outperform) a traditional PS, as we do not store the entire model in memory. It demonstrates that, in principle, our aggregation protocol could be run entirely in software on a middlebox, but with lower performance: in-network aggregation inherently requires fewer resources than host-based aggregation.

6.2 SwitchML improves training speed

We analyze training performance on eight DNN benchmarks. We normalize results to NCCL as the underlying communication backend of PyTorch and TensorFlow.

Figure 4 reports the speedup for processing a training batch for SwitchML compared to NCCL at 100 Gbps. SwitchML uses the DPDK implementation with 256-value packets. These results replay the profile traces collected on our cluster

Figure 6: 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.

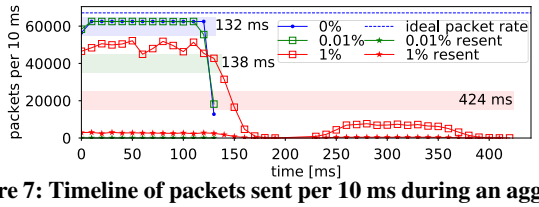

Figure 7: Timeline of packets sent per 10 ms during an aggregation with 0%, 0.01% and 1% packet loss probability. Horizontal bars denote the TAT in each case.

(§2.2), allowing us to report both the speedup on our testbed GPUs (P100s, which are two generations old) and hypothetical GPUs that are 10× faster (by uniformly scaling the traces by this factor). This emulation lets us evaluate the setting where a fast network is paired with fast GPUs. While it is hard to predict future evolution of GPU speed vs. network bandwidth, we reason that this scaling factor currently corresponds to the span of two to three GPU generations (the A100 benchmarks at 4.2× the V100 [75], which in turn is 1.4-2.2× faster than our P100 [97]) and represents a likely bound on the real-world speedups achievable, which are anyway dependent on the model (e.g., the ResNet50 model sees a nearly 10× speedup from an NVIDIA V100 GPU compared to a K80 GPU [71]) and the other infrastructure specifics.

As expected, SwitchML accelerates batch processing especially for the larger DNNs. The speedup over NCCL-RDMA is at most 2.1×, which is in line with the fundamental 2× advantage of INA over RAR (§3). In most cases, the measured speedup is higher than the emulated communication results (Table 2) predict, because NCCL's RAR implementation does not achieve the theoretical maximum efficiency. The speedup relative to NCCL-TCP is larger (up to one order of magnitude), which is attributable primarily to DPDK's kernel-bypass advantage.

SwitchML provides significant benefits for many, but not all, real-world DNNs, even with 100 Gbps networks. For example, DeepLight and LSTM enjoy major improvements. BERT sees a somewhat lower speedup, in part because its gradient consists of many relatively small (∼60 MB) tensors. Similarly, NCF, a relatively small model, has a modest speedup. Other models, like UGATIT, SSD, and ResNet are simply not network-bound at 100 Gbps. SSD is a particularly challenging case: not only is it a small model that would require an $\alpha = 15.2\times$ faster GPU to become network-bound (Table 1), it also makes many aggregation invocations for small gradients. The overheads of starting an aggregation are not well amortized, especially in the 10× scaled scenario.

Finally, we consider the end-to-end speedup on a complete training run with 16 workers. We focus on the four models that are network-bottlenecked at 100 Gbps. Figure 5 shows the training performance speedup compared to NCCL using RDMA and TCP. These measurements use SwitchML's DPDK implementation, with 256-value packets; we expect a larger speedup once SwitchML's RDMA implementation is integrated with the training framework. Even so, SwitchML's speedups range between 1.13-2.27× over NCCL-RDMA and 2.05-5.55× over NCCL-TCP. The results are not directly comparable to Figure 4, because (1) they use a larger 16-node cluster, and (2) they report total end-to-end iteration time, which also includes data loading time. Our deployment does not use any optimized techniques for data loading, an orthogonal problem being addressed by other work (e.g., DALI [76]).

6.3 Overheads

Packet loss recovery. We study how packet loss affects TAT. 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 1 ms. We run microbenchmark experiments in similar scenarios as §6.1. We report a few representative runs.

Figure 6 measures the inflation in TAT with different loss probabilities. SwitchML completes tensor aggregation significantly faster than Gloo or NCCL when the loss is 0.1% or higher. A loss probability of 0.01% minimally affects TAT in either case. To better illustrate the behavior of SwitchML, we show in Figure 7 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.

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 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 emulate this by halving the tensor size.

We find that these overheads are negligible at 10 Gbps.

This is due to our use of x86 SSE/AVX instructions. When we use float16, performance doubles, as expected. However, these overheads become more relevant as data rates increase, requiring to offload type conversion operations to the GPU at 100 Gbps and scaling up the number of cores; RDMA alleviates the pressure for CPU cycles used for I/O (see the gap between DPDK- and RDMA-based performance in Figure 3).

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 the memory requirement is less than 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.

7 Extensions

Scaling beyond a rack. We described SwitchML in the context of a rack. However, large scale ML jobs could span beyond a single rack. SwitchML's design can support multiple racks by hierarchically composing several instances of our switch logic, although we do not have a testbed large enough to test (or require) such a design. Each worker is connected to a top-of-rack switch, which aggregates updates from the workers in the rack. Rather than broadcast the result packet to the workers, it instead sends it to a tier-1 aggregation switch, which aggregates updates from multiple racks. This can continue with as many levels as are needed to support the desired network topology. Ultimately, a root switch completes the aggregation of partial aggregates and multicasts a result packet downstream. At each level, the switches further multicast the packet, ultimately reaching the workers.

The hierarchical approach also allows us to support switches with multiple processing pipelines. Suppose a switch has pipelines that can aggregate up to p ports (for the switches we use, $p = 16$). In this setting, each switch aggregates tensors from d downstream ports and forwards partial aggregates via $u = \lceil \frac{d}{p} \rceil$ upstream ports. In other words, the switch operates as u virtual switches, one for each pipeline in the switch.

This hierarchical composition is bandwidth-optimal, as it allows n workers to fully utilize their bandwidth while supporting all-to-all communication with a bandwidth cost proportional to u instead of n. That is, every switch aggregates data in a $p:1$ ratio. As a result, the system naturally supports oversubscription of up to this ratio at the aggregation or core layers. This allows it to support large clusters with relatively shallow hierarchies; using the current generation of 64-port, 4-pipeline 100 Gbps switches, a two-layer hierarchy can support up to 240 workers and a three-layer one manages up to 3,600.

Importantly (and by design), the packet loss recovery algorithm described in §4.5 works in the multi-rack scenario. Thanks to the use of bitmaps and shadow copies, a retransmission originated from a worker will be recognized as a retransmission on all switches that have already processed that packet. This triggers the retransmission of the aggregated packet toward the upper layer switch, ensuring that the switch affected by the packet loss is always ultimately reached.

Congestion control. We have not implemented an explicit congestion control algorithm; the self-clocking streaming protocol is a flow control mechanism to control access to the switch's aggregator slots. It also serves as a rudimentary congestion control mechanism, in that if one worker's link is congested and it cannot process aggregation results at full speed, the self-clocking mechanism will reduce the sending rate of all workers. This is sufficient for dedicated networks (which is common for ML clusters in practice). For more general use, a congestion control scheme may be needed; concurrent work has been developing such protocols [29].

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, 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; concurrent work has been exploring a similar approach atop an FPGA board [62].

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. Alternatively, ATP [54] – a follow up work to ours – explores

the idea of partitioning aggregation functionality between a switch (for performance) and a server (for capacity) so as to seamlessly support multi-job scenarios.

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

8 Related work

In-network computation trends. The trend towards programmable data planes has sparked a surge of proposals [20, 21, 46, 47, 55, 61] to offload, when appropriate [80], application-specific primitives into network devices.

In-network aggregation. We are not the first to propose aggregating data in the network. Targeting partition-aggregate and big data (MapReduce) applications, NetAgg [65] and CamDoop [18] 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 [64] does the same with a rack-scale parameter server. Historically, some specialized supercomputer networks [2, 26] offloaded MPI collective operators (e.g., all-reduce) to the network. SwitchML differs from all of these approaches in that it performs in-network data reduction using a streaming aggregation protocol.

The closest work to ours is DAIET [86], which was our initial but incomplete proposal of in-network aggregation for minimizing the overhead of exchanging ML model updates.

Mellanox's Scalable Hierarchical Aggregation Protocol (SHARP) is a proprietary in-network aggregation scheme available in certain InfiniBand switches [33]. SHARP uses dedicated on-chip FPUs for collective offloading. The most recent version, SHARPv2 [68] uses streaming aggregation analogous to ours. A key difference is that SHARP builds on InfiniBand where it can leverage link-layer flow control and lossless guarantees, whereas SwitchML runs on standard Ethernet[8] with an unmodified network architecture, necessitating a new packet recovery protocol. More fundamentally, SwitchML builds on programmable network hardware rather than SHARP's fixed-function FPUs, which offers two benefits. First, operators can deploy a single switch model either for SwitchML or traditional networking without waste: the ALUs used for aggregation can be repurposed for other tasks. Second, it allows the system design to evolve to support new ML training approaches. For example, we are currently experimenting with new floating-point representations and protocols for sparse vector aggregations [27]. With a fixed-function approach, these would require new hardware, just as moving from single HPC reductions (SHARPv1) to streaming ML reductions (SHARPv2) required a new ASIC generation.

[8] Although SwitchML uses RDMA, it uses only unreliable connections, and so does *not* require any of the "lossless Ethernet" features of RoCE.

Concurrently, Li et al. [57] explored the idea of in-switch acceleration for Reinforcement Learning (RL). Their design (iSwitch) differs from ours in two fundamental ways. First, while their FPGA-based implementation supports more complex computation (e.g., native floating point), it operates at much lower bandwidth (4×10 Gbps). Second, it stores an entire gradient vector during aggregation; for RL workloads with small models, this works, but it does not scale for large DNN models. Our work targets both large models and high throughput – a challenging combination given the limited on-chip memory in high-speed networking ASICs. SwitchML's software/hardware co-design approach, using a self-clocking streaming protocol, provides $40\times$ higher throughput than iSwitch, while supporting arbitrarily large models.

Finally, targeting NVIDIA's intra-GPU NVLink network, Klenk et al. [52] proposed in-network aggregation in the context of a distributed shared-memory fabric supporting multi-GPU systems where accelerators are directly attached to the fabric. While this work is orthogonal to ours, their push reduction design resembles the SwitchML protocol, suggesting that streaming in-network aggregation has broader applicability than discussed in this paper.

Accelerating DNN training. A large body of work has proposed improvements to hardware and software systems, as well as algorithmic advances for faster DNN training. 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, 13, 15, 22, 56, 64, 94]. While data parallelism is most common, it can be advantageous to combine the two approaches. Recent work even shows how to automatically find a fast parallelization strategy for a specific parallel machine [44]. Underpinning any distributed training strategy, lies parameter synchronization. Gibiansky was among the first to research [30] using fast collective algorithms in lieu of the traditional parameter server approach. Many platforms have now adopted this approach [30, 37, 41, 87, 89]. We view SwitchML as a further advancement on this line of work – one that pushes the boundary by co-designing networking functions with ML applications.

9 Conclusion

SwitchML speeds up DNN training by minimizing communication overheads at single-rack scale. SwitchML uses in-network aggregation to efficiently synchronize model updates at each training iteration among distributed workers executing in parallel. We evaluated SwitchML with eight real-world DNN benchmarks on a GPU cluster with 10 Gbps and 100 Gbps networks; we showed that SwitchML achieves training throughput speedups up to $5.5\times$ and is generally better than state-of-the-art collective communications libraries. We are in the process of integrating SwitchML-RDMA in various ML frameworks.

Acknowledgments

We are grateful to Ibrahim Abdelaziz for his work on an initial proof-of-concept of SwitchML and to Omar Alama for contributing with performance improvements and with the code release. We thank Tom Barbette and Georgios Katsikas for their help with DPDK issues. This work benefited from feedback and comments by Steven Hand, Petros Maniatis, KyoungSoo Park, and Amin Vahdat. We thank our shepherd, Sujata Banerjee, and the anonymous reviewers for their helpful feedback. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST.

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

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

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

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

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

[8] J. Bernstein, J. Zhao, K. Azizzadenesheli, and A. Anandkumar. signSGD with Majority Vote is Communication Efficient And Byzantine Fault Tolerant. *arXiv 1810.05291*, 2018.

[9] 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.

[10] 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.

[11] Cerebras. https://www.cerebras.net.

[12] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. *arXiv 1312.3005*, 2013.

[13] 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.

[14] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT*, 2017.

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

[16] 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.

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

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

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

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

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

[22] 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.

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

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv 1810.04805*, 2018.

[25] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi. Training DNNs with Hybrid Block Floating Point. In *NIPS*, 2018.

[26] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations. In *HOTI*, 2009.

[27] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. Technical report, KAUST, 2020. http://hdl.handle.net/10754/665369.

[28] O. Fercoq, Z. Qu, P. Richtárik, and M. Takáč. Fast distributed coordinate descent for minimizing non-strongly convex losses. In *MLSP*, 2014.

[29] N. Gebara, P. Costa, and M. Ghobadi. In-network Aggregation for Shared Machine Learning Clusters. In *MLSys*, 2021.

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

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

[32] 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. *arXiv 1706.02677*, 2017.

[33] 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.

[34] Graphcore. https://www.graphcore.ai.

[35] Habana Gaudi. https://www.habana.ai/training.

[36] F. M. Harper and J. A. Konstan. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.*, 5(4), Dec. 2015.

[37] 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.

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

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

[40] S. Horváth, C.-Y. Ho, L. Horváth, A. N. Sahu, M. Canini, and P. Richtárik. Natural Compression for Distributed Deep Learning. *arXiv 1905.10988*, 2019. http://arxiv.org/abs/1905.10988.

[41] 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.

[42] InfiniBand Trade Association. RoCE v2 Specification. https://cw.infinibandta.org/document/dl/7781.

[43] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.

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

[45] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[46] 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.

[47] 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.

[48] 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.

[49] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the Limits of Language Modeling. *arXiv 1602.02410*, 2016.

[50] 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.

[51] J. Kim, M. Kim, H. Kang, and K. H. Lee. U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. In *ICLR*, 2020.

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

[53] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NIPS*, 2017.

[54] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*, 2021.

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

[56] 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.

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

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

[59] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick. Microsoft COCO: Common Objects in Context. In *ECCV*, 2014.

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

[61] 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.

[62] S. Liu, Q. Wang, J. Zhang, Q. Lin, Y. Liu, M. Xu, R. C. Chueng, and J. He. NetReduce: RDMA-Compatible In-Network Reduction for Distributed DNN Training Acceleration. *arXiv 2009.09736*, 2020.

[63] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single Shot MultiBox Detector. In *ECCV*, 2016.

[64] 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.

[65] 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.

[66] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. *arXiv 1804.07612*, 2018.

[67] Mellanox RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

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

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

[70] K. Mishchenko, B. Wang, D. Kovalev, and P. Richtárik. IntSGD: Floatless Compression of Stochastic Gradients. *arXiv 2102.08374*, 2021. https://arxiv.org/abs/2102.08374.

[71] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[72] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM J. Optim.*, 19(4), 2009.

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

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

[75] NVIDIA's Ampere A100 GPU Is Unstoppable, Breaks 16 AI Performance Records, Up To 4.2x Faster Than Volta V100. https://wccftech.com/nvidia-ampere-a100-fastest-ai-gpu-up-to-4-times-faster-than-volta-v100/.

[76] NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/DALI.

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

[78] NVIDIA Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference.

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

[80] D. R. K. Ports and J. Nelson. When Should The Network Be The Computer? In *HotOS*, 2019.

[81] P. Rajpurkar, R. Jia, and P. Liang. Know What You Don't Know: Unanswerable Questions for SQuAD. In *ACL*, 2018.

[82] P. Richtárik and M. Takáč. Distributed Coordinate Descent Method for Learning with Big Data. *Journal of Machine Learning Research*, 17(1), 2016.

[83] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3), 1951.

[84] C. Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

[85] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.

[86] 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.

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

[88] 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.

[89] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv 1802.05799*, 2018.

[90] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

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

[92] 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.

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

[94] 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.

[95] 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.

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

[97] R. Xu, F. Han, and N. Dandapanthula. Deep Learning on V100. Technical report, DELL HPC Innovation Lab, 2017. https://downloads.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers16_en-us.pdf.

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

[99] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv 1606.06160*, 2016.

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 [92], 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

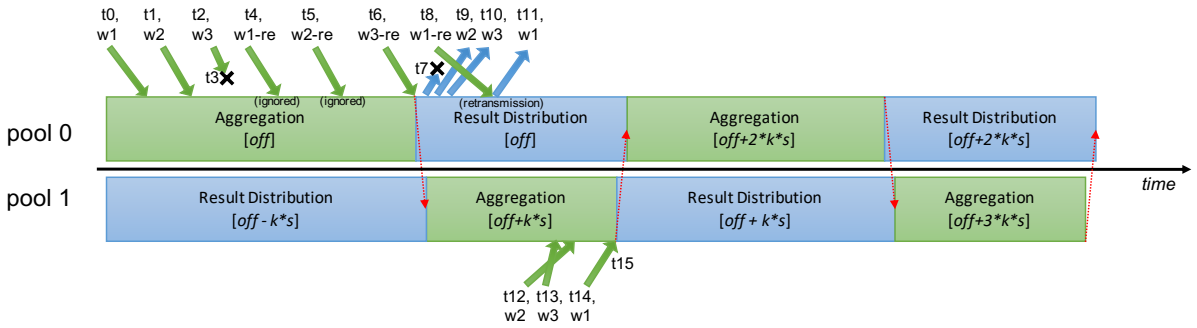

Figure 8: An example execution of a SwitchML switch interacting with three workers. The figure illustrates how a slot with index x is used during the different phases (shown in different colors) that alternate between the two pools.

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 register 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 64 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 64 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).

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 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.

One CPU core is sufficient to do reduction at line rate on a 10 Gbps network. However, to be able to scale beyond 10 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. 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.

RDMA implementation details. We found that the cost of processing individual SwitchML packets, even using DPDK with 256-element packets and multiple cores, was too high to achieve line rate. Other aggregation libraries use RDMA to offload packet processing to the NIC. In RDMA-based systems NICs implement packetization, flow control, congestion control, and reliable delivery. In normal usage, clients use RDMA to send and receive messages of up to a gigabyte; the NIC turns them into packets and ensures they are delivered reliably. Furthermore, clients may register memory regions with the NIC, allowing other clients to remotely read and write them without CPU involvement. This reduces or eliminates

work done on the clients' CPUs to complete the transfer.

Turning a Tofino switch into a fully-featured RDMA endpoint is not the solution. Implementing timeouts and retransmission in a way that is compatible with existing poorly-documented existing RDMA NICs would be complex. Furthermore, such an implementation would not be an good fit for SwitchML: the RDMA protocols are largely designed for point-to-point communication, whereas SwitchML's protocol is designed for collective communication.

Fortunately, RDMA NICs implement multiple protocols with different properties. The standard Reliable Connected (RC) mode ensures reliable delivery and supports CPU-bypassing remote reads and writes (as well as sends and receives) of up to 1 GB. The UDP-like Unreliable Datagram (UD) mode supports just sends and receives of up the network MTU. Finally, the Unreliable Connected (UC) mode fits somewhere in between. It supports packetization, allowing for sends, receives, and writes of up to 1 GB. It also generates and checks sequence numbers, allowing it to detect packet drops, but it does not retransmit: instead, if a gap in sequence numbers is detected, incoming packets are silently dropped until the first packet of a new message arrives. Then, the sequence number counter is reset to the sequence number of that packet, and normal reception continues.

We use RDMA UC to implement a RDMA-capable variant of SwitchML using a subset of the RoCE v2 protocol [42]. Its operation is very similar to what is described in Section 4, with three main differences.

First, where base SwitchML sends and receives slot-sized packets, SwitchML RDMA sends multi-slot messages. Each packet of a message is treated largely as it is in the base protocol by the switch, but the pool index for each packet is computed as an offset from the base index provided with the first packet of the message. Timeouts are tracked just as they are in the base protocol, but when a packet drop is detected, the client retransmits the entire message rather than just the dropped packet. This makes retransmissions more expensive, but it also drastically lowers the cost incurred sending packets in the common case of no packet drops; since packet drops are rare within a datacenter, the benefit is large.

Second, SwitchML consumes and generates sequence numbers on the switch. In order to allow messages with multiple packets to aggregate concurrently, each in-flight message is allocated its own queue pair, with its own sequence number register. This allows clients to to be notified when a write message from the switch has arrived with no drops; it also allows the switch to ignore packets in messages received out of sequence. However, the same per-slot bitmap used in the base protocol is still used to ensure that duplicate packets from a retransmission of a partially-received messages are not re-applied. Packets are transmitted as individual slots addressed by a message complete. This means that the packets from multiple messages may interleave on the wire, but since each is on a separate queue pair with its own sequence number space, the NIC will reassemble them successfully.

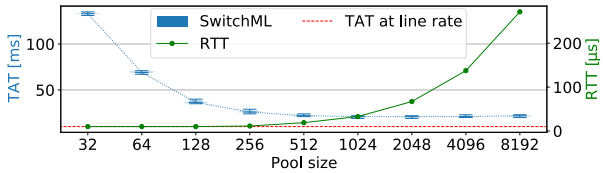

Figure 9: Effect of pool size on overall tensor aggregation time (TAT) and per-packet RTT (right y-axis) at 100 Gbps.

Third, SwitchML RDMA uses RDMA Write Immediate messages for all communication. This allows clients to send data directly from GPU memory, and the switch to write directly into GPU memory (if the host is GPU Direct-capable). Byte order conversion and scaling are done on the GPU; the CPU is responsible only for issuing writes when data from the GPU is ready, detecting completions and timeouts, and issuing retransmissions when necessary. Necessary metadata for the SwitchML protocol is encoded in fields of the RDMA header; the RDMA RKey and Address fields are used to encode the destination slot and the address to write the response to. The Immediate field is used to carry up to four scaling factors. At job setup time, the clients communicate with the switch and give it their queue pair numbers, initial sequence numbers, and an RKey for its switch-writable memory region. The switch uses these to form RDMA Write Immediate messages with appropriate sequence numbers, destination addresses, and immediate values, of the same size as the messages sent from the clients to the switch.

Finally, it is important to note that SwitchML RDMA does not require lossless Ethernet to be configured, as is common in RoCE deployments. Enabling lossless Ethernet would reduce the probability of packet drops, but would add complexity to the network deployment. SwitchML's reliability protocol makes this unnecessary.

DNN workloads. Table 3 details the models, datasets and ML toolkits used in the experiments.

C Tuning the pool size

As mentioned, the pool size s affects 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 worker. Note that the delay here refers to the end-to-end delay, including the end-host processing time, which 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, and this is possible when $s \cdot b$ matches the BDP. A significantly higher value of s, when used as the initial window size, will unnecessarily increase queuing time within the workers.

Model	Task	Dataset	Sys
DeepLight [23]	Click-through Rate Prediction	Criteo 1TB [19]	PyT
LSTM [49]	Language Modeling	GBW [12]	PyT
NCF [39]	Recommendation	ML-20m [36]	PyT
BERT [24]	Question Answering	SQuAD [81]	TF
VGG19 [91]	Image Classification	ImageNet-1K [85]	PyT
UGATIT [51]	Image-to-Image Translation	Selfie2Anime [51]	TF
ResNet50 [38]	Image Classification	ImageNet-1K [85]	TF
SSD [63]	Object Detection	COCO 2017 [59]	PyT

Table 3: DDL benchmarks. Models, task, dataset used for training and ML toolkit (PyT=PyTorch; TF=TensorFlow).

Model	Metric	No compression	Top-10%	Top-1%
DeepLight	AUC	0.9539	0.9451	0.9427
LSTM	Perplexity	32.74	86.26	82.55
NCF	Hit rate	0.9586	0.9369	-
BERT	F1 score	91.60	91.47	91.52
VGG19	Top-1 accuracy	68.12	64.85	57.70
UGATIT	-	-	-	-
ResNet50	Top-1 accuracy	74.34	74.59	72.63
SSD	Accuracy	0.2549	0.2487	0.2309

Table 4: Test metrics comparison. NCF at Top-1% did not converge. BERT result is the median of 6 runs of fine-tuning from a pre-trained model. BERT pre-training results are shown in Figure 13. UGATIT fails to execute with the compressor implementation in [96]. See Figure 10 for the convergence behavior during training.

Second, a correctness requirement for our communication scheme is 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 9), we use 128 and 512 as the pool size for 10 and 100 Gbps, respectively. This occupies 256 KB and 1 MB of register space in the switch, respectively. We note that the switch can support one order of magnitude more slots, and SwitchML uses much less than 10% of that available.

D Compression affects convergence

Table 4 reports the model quality obtained without gradient compression and with Top-k compression using $k = 1\%, 10\%$. Model quality is assessed using per-model accuracy metrics.

Results are shown in Figure 10. We observe that loss and accuracy do not necessary correlate well. For example, in the case of SSD all methods have similar loss trace, but obvious accuracy gap. For NCF, Top-k at 1% does not converge, but the accuracy can still go up.

E 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 [33]. 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? In theory either workers or the switch can perform type conversion. 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 a restricted form of 16-bit floating point on a Tofino chip by using lookup tables and ALUs to do the conversion. This means there is no type conversion overhead at end hosts. However, this approach still requires to scale the gradient values due to the limited range of floating point conversion the switch can perform. Besides, unless half-precision training is used, the worker must still convert from 32-bit to 16-bit floating points. 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 11).

What are the implications in terms of accuracy? Recently, several update compression (e.g., quantization, dithering or sparsification) strategies were proposed (see [96] for a survey) 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 [88], signSGD [7, 8], QSGD [4], which uses just the sign of the stochastic gradients to inform the update, Terngrad [95], which uses ternary quantization, and the DIANA framework [69], 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. In contrast, our 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 quantization mechanism. Model updates are divided into packet-sized blocks. With a small abuse of notation, in the following all equations refer to per-block operations. Each worker multiplies its block model update $\Delta_i^t = \Delta(x^t, D_i^t)$ by a vector of scaling factors $f > 0$, obtaining $f\Delta_i^t$. The elements of f are chosen per block, and are chosen such that that all k entries of the scaled update can be rounded to a number representable as an integer without overflow. We then perform this rounding, obtaining vectors $Q_i^t = \mathrm{p}(f\Delta_i^t) \in \mathbb{Z}^k$ for $i = 1, 2, \ldots, n$, where p is the rounding operator, which are sent to the switch and aggregated,

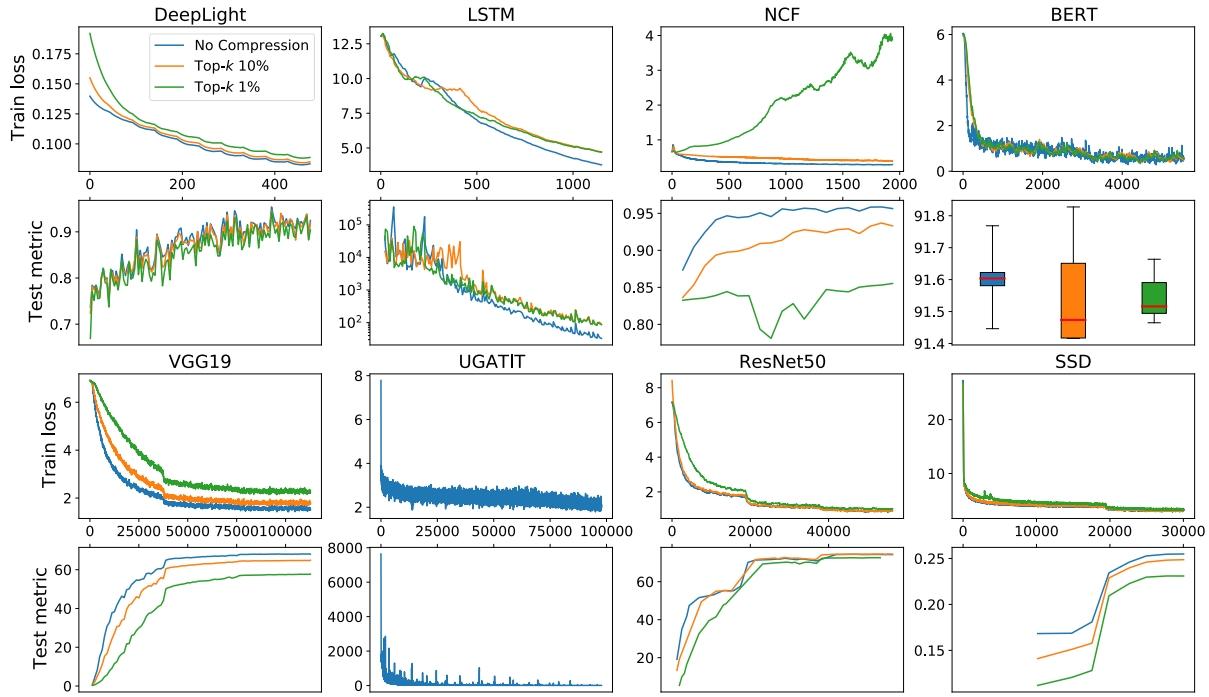

Figure 10: Convergence behavior under different compression scheme. The x axis shows the iteration number. All methods execute a fixed number of steps and hyperparameters are kept the same. Refer to Table 4 for test metrics of each task. We plot generator loss as the metric for UGATIT because there is no objective metric available for GAN workloads. Note that for the perplexity metric of LSTM, lower is better.

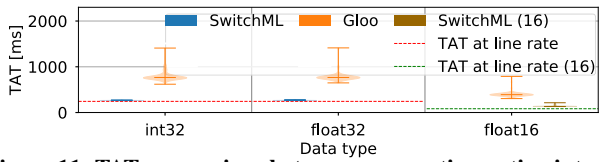

Figure 11: TAT comparison between aggregating native-integer tensors, scaling and converting from single-precision (float32) tensors, and scaling and converting from half-precision (float16) tensors.

resulting in

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

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 on the workers, 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.

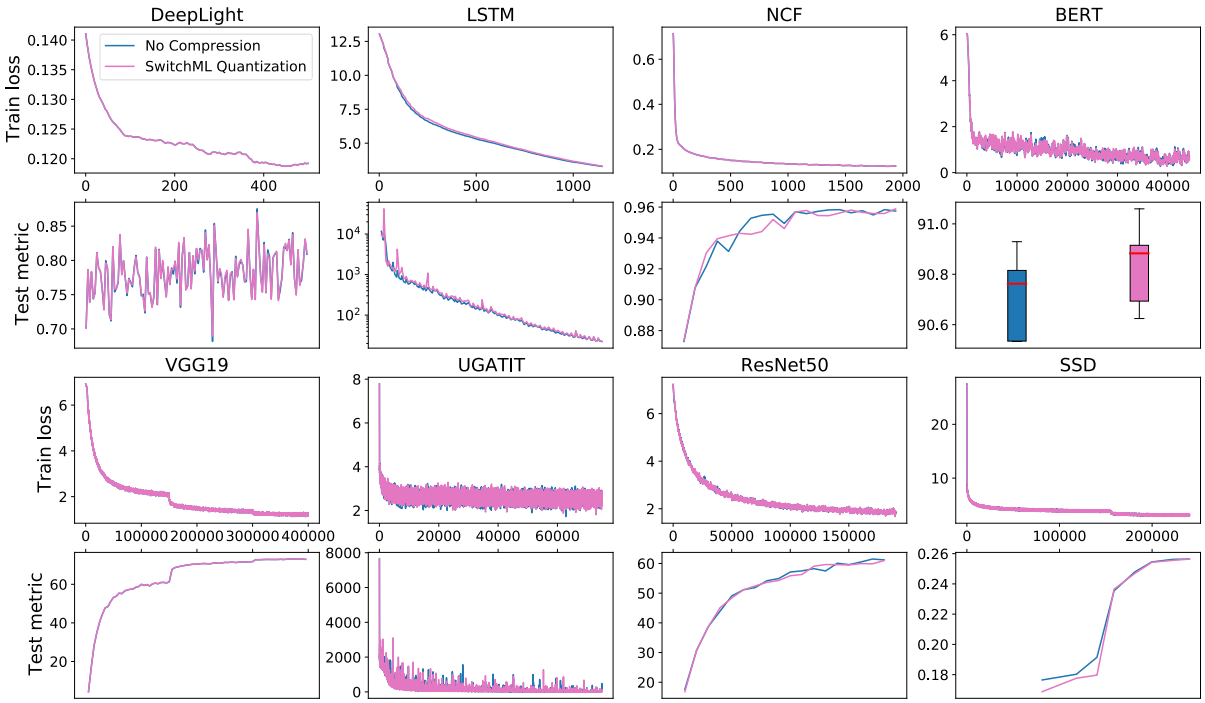

Figure 12: Convergence analysis of SwitchML quantization method on a single GPU. The x axis shows the iteration number.

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

Assumption 2. $|\sum_{i=1}^n \rho(f\Delta_i^t)| \leq 2^{31}$ *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

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

Using the same argument, we get a similar lower bound

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

□

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:[9]

Assumption 3. There exists $B > 0$ *such that* $|\Delta_i^t| \leq B$ *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 $0 < f \leq \frac{2^{31}-n}{nB}$.

Proof. We have $\rho(f\Delta_i^t) \leq f\Delta_i^t + 1 \leq f|\Delta_i^t| + 1 \leq fB + 1$. Likewise, $\rho(f\Delta_i^t) \geq f\Delta_i^t - 1 \geq -f|\Delta_i^t| - 1 = -(fB+1)$. So,

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

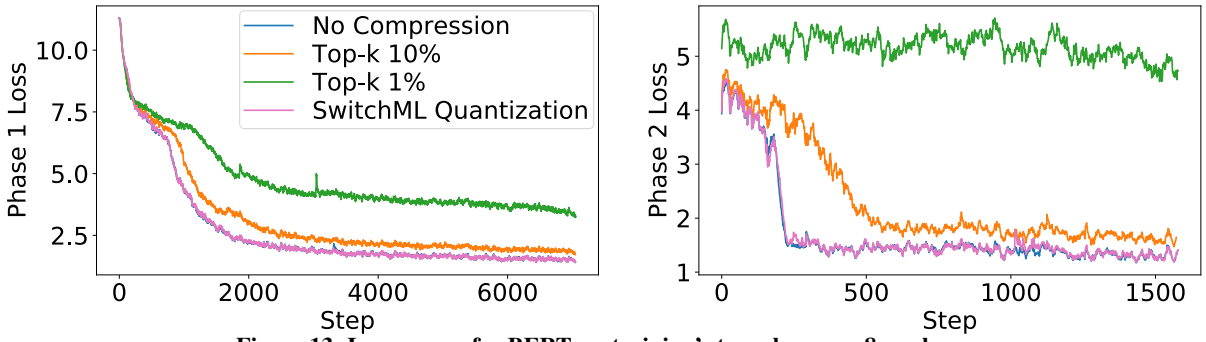

Figure 13: Loss curves for BERT pretraining's two phases on 8 workers.

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

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 $\frac{n^2B}{2^{31}-n}$. In typical applications, $n^2B \ll 2^{31}$, which means that the error we introduce is negligible.

Empirical study. We name the above method "SwitchML quantization" and run end-to-end experiments to study its convergence behavior. Figure 12 shows that SwitchML quantization achieves similar training loss and does not incur test metric decrease in all models detailed in Table 4. We further show in Figure 13 that SwitchML quantization has little to no difference in the training loss curve of BERT [24] pretraining compared to the no compression baseline, while Top-k compression with $k = 1\%, 10\%$ has a big impact on model convergence.

F 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 (where E denotes encryption). For instance, recent work by Cheon et al. [14] developed a homomorphic encryption scheme for approximate arithmetic. By customizing the end-host encryption process, the worker could encrypt all the vector elements using such a cryptosystem, knowing that the aggregated model update can be obtained by decrypting the data aggregated at the switches. We leave it to future work to validate this concept.