BLINK: FAST AND GENERIC COLLECTIVES FOR DISTRIBUTED ML

Guanhua Wang [1] **Shivaram Venkataraman** [2] **Amar Phanishayee** [3] **Jorgen Thelin** [3] **Nikhil Devanur** [3] **Ion Stoica** [1]

ABSTRACT

Model parameter synchronization across GPUs introduces high overheads for data-parallel training at scale. Existing parameter synchronization protocols cannot effectively leverage available network resources in the face of ever increasing hardware heterogeneity. To address this, we propose `Blink`, a collective communication library that dynamically generates optimal communication primitives by *packing spanning trees*. We propose techniques to minimize the number of trees generated and extend `Blink` to leverage heterogeneous communication channels for faster data transfers. Evaluations show that compared to the state-of-the-art (NCCL), `Blink` can achieve up to $8\times$ faster model synchronization, and reduce end-to-end training time for image classification tasks by up to 40%.

1 INTRODUCTION

Large high-quality datasets and massive compute clusters have enabled machine learning algorithms, such as Deep Neural Networks (DNNs), to tackle hard problems in a number of domains including image classification, object detection, machine translation, and speech processing. Models developed for such tasks can take a long time to train; for example, models for image classification tasks (Russakovsky et al., 2015) can often take days or even weeks to train on a single GPU. Thus, fast training of large deep learning models requires distributed training on many GPUs. The most widely used method for reducing DNN training time is to perform data-parallel training (Abadi et al., 2016; Goyal et al., 2017). In data-parallel training, each GPU has a full copy of the model parameters and GPUs frequently exchange parameters with other GPUs involved in training.

Parameter synchronization across GPUs introduces significant overheads when training at scale with communication overheads that can range from 50% to 90% for popular ML models (Narayanan et al., 2019). This problem is accentuated by the fact that GPU computation is getting faster and model sizes are growing larger, thus making communication overheads stand out. But two recent trends seem to *suggest* that their arrival might alleviate, or even eliminate, such communication bottlenecks for DNN training. First, on the hardware front, state-of-the-art multi-GPU servers, like NVIDIA's DGX-1 and DGX-2, now have fast interconnects between GPUs – NVLink offers 20-25GBps pairwise and bi-directional peak throughput (NVLink; NVSwitch). Second, modern communication libraries such as NVIDIA's Collective Communications Library (NCCL) (Jeaugey, 2017), Uber's Horovod (Sergeev & Balso, 2018), and Baidu's Ring AllReduce (Ng, 2017), with techniques such as wait-free backpropagation designed to hide communication overheads (Zhang et al., 2017), are solutions specifically targeted at speeding up parameter synchronization.

In this paper, we focus on multi-GPU servers with NVLink/NVSwitch and find that despite recent advances, modern communication libraries for parameter exchange are unable to fully mitigate communication bottlenecks in data-parallel training. The central hurdle in achieving peak performance for inter-GPU collectives is link under-utilization due to *topology heterogeneity*. We find this occurs due to two main reasons:

First, topology heterogeneity can occur due to *differing server configurations*. Fig. 1 shows an example of two generations of servers, the DGX-1-P100 (DGX-1P) and DGX-1-V100 (DGX-1V), and their NVLink topologies. Protocols have to be topology aware to effectively use hardware.

Second, schedulers that allocate GPUs to jobs, especially in *multi-tenant* clusters, are oblivious to interconnect topologies between GPUs. Many jobs can potentially be co-located on the same machine. Furthermore, even topology aware schedulers must embrace *fragmentation* to avoid queuing delays (e.g., a 8-GPU job might have to contend with 3 GPUs on one machine and 5 GPUs on another) (Jeon et al., 2018). In an analysis of over 40,000 multi-GPU jobs over a three month period on a multi-tenant cluster at Cloud-X (Figure 2), we find that it is common for jobs to be allocated 3, 5, 6, or 7 GPUs on individual 8-GPU servers despite multi-GPU jobs overwhelmingly requesting GPUs in powers of 2. While, fragmentation can be mitigated, not avoided, by making schedulers topology aware *and* capable

[1]University of California, Berkeley [2]University of Wisconsin, Madison [3]Microsoft Research. Correspondence to: Guanhua Wang <guanhua@cs.berkeley.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020.

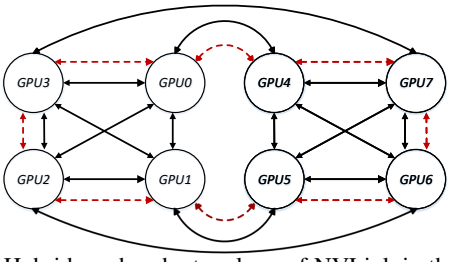


Figure 1. Hybrid mesh-cube topology of NVLink in the DGX-1 8-GPU server. Solid lines here indicate the bi-directional NVLinks on the DGX-1-P100, red dashed-lines are the additional NVLinks in DGX-1-V100 servers. NVLink Gen1 has bi-directional pairwise throughput of 18-20GB/s (DGX-1-P100); Gen2 goes up to 22-25GB/s (DGX-1-V100).

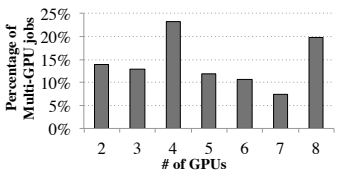


Figure 2. Number of GPUs within each 8-GPU server allocated to 40,000 multi-GPU jobs

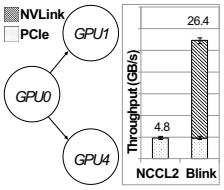


Figure 3. Broadcast throughput (partially-connected GPUs)

of migration (Xiao et al., 2018), such solutions face a higher barrier of entry as there are many independent scheduling frameworks that all need to be changed and not all jobs can be placed appropriately given variable arrival rates.

The resulting topology heterogeneity caused by scheduler allocation can result in link under-utilization in current ring-based protocols for parameter exchange. For example, in Figure 3, NCCL is unable to utilize the bi-directional NVLinks between the 3-GPUs; the lack of NVLink between GPUs 1 and 4 prevents NCCL from constructing NVLink-only rings and it has to fall back on PCIe based communication. But link under-utilization can also occur even when rings can be constructed using NVLink. Figure 4 shows a 6 GPU allocation on a DGX-1P, where despite being able to construct two NVLink-based rings, NCCL has to drop some of the links connecting the GPUs as they don't contribute to ring construction.

Contributions. In this paper, we propose `Blink`, a communication library for inter-GPU parameter exchange that achieves near-optimal link utilization. To handle topology heterogeneity from hardware generations or partial allocations from cluster schedulers, `Blink` dynamically generates optimal communication primitives for a given topology. `Blink` probes the set of links available for a given job at runtime and builds a topology with appropriate link capacities. Given the topology, `Blink` achieves the optimal communication rate by *packing spanning trees*, that can utilize more links (Lovasz, 1976; Edmonds, 1973) when compared to rings. We use a multiplicative-weight update based approximation algorithm to quickly compute the maximal packing and extend the algorithm to further minimize the number of trees generated. We also describe how this scheme can handle one-to-many primitives like Broadcast or Gather and how we can extend this to many-to-many primitives like AllReduce using bi-directional links and hardware capability to compute at line rate. `Blink`'s collectives extend across multiple machines effectively utilizing all available network interfaces.

Based on the spanning trees chosen, `Blink` dynamically generates code to implement common collective primitives. Our generated code automatically chunks data and uses CUDA streams to efficiently pipeline transfer and computation. From the programmer's perspective, `Blink` provides NCCL-compatible API. It can be seamlessly plugged into distributed ML frameworks like TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), etc. `Blink` does not requires user program modifications and only relies on *preloading* (`LD_PRELOAD`).

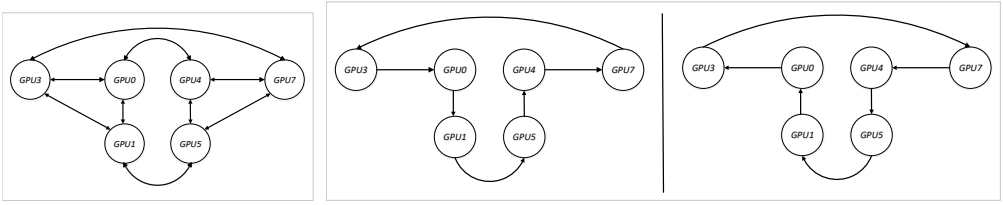
We evaluate `Blink`'s performance on a number of multi-GPU platforms including DGX-1P, and DGX-1V and DGX-2 (dgx2). Results show that, compared with NCCL, on DGX-1V, `Blink` achieves up to 6× speed-up in all-to-one/one-to-all collective communications (e.g. Broadcast, Gather), and is up to 8× faster in all-to-all collective communications (e.g. AllReduce). On DGX-2, we show that single-hop trees in `Blink` are especially effective for smaller data sizes offering up to 3x lower latency and higher throughput, compared to NCCL's double-binary trees and rings (NCCL 2.4). Finally, we also find that Blink can accelerate DNNs training on single and multi-machine setups. For instance, on a single DGX-1V machine, compared to NCCL, `Blink` can reduce communication cost up to 87% (51% on average), and speeds up end-to-end training by up to 40%.

2 Motivation

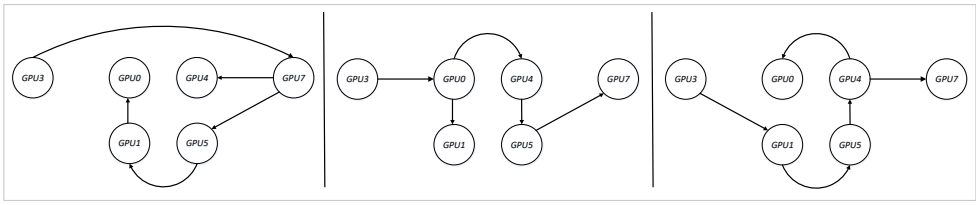
In this section, we first discuss the need for more efficient communication primitives and why ring-based solutions like NCCL cannot handle topology heterogeneity. We highlight the case for spanning tree-based protocols and the need to pack trees to achieve peak performance in the face of topology heterogeneity. We then present micro-benchmarks characterizing the capabilities of modern GPU hardware that helps guide Blink's design.

2.1 The case for packing trees

The motivation for our work stems from the high communication overheads experienced by deep learning workloads when running data-parallel training even on fast multi-GPU servers like the NVIDIA DGX-1 (Narayanan et al., 2019). These overheads occur despite setting per-GPU minibatch sizes to the largest values that fit in GPU memory, using state of the art libraries like NCCL, and using optimizations



(a) 6-GPU group topology (b) NCCL rings. Links between GPU 1&3, 5&7, and 0&4 are unused.



(c) Blink 6-GPU spanning trees

Figure 4. Broadcast comparison between NCCL and Blink over 6-GPUs in DGX-1P

common in modern frameworks such as Wait-free Backpropagation (Zhang et al., 2017). Communication overheads arise from a number of factors including increased model sizes and faster computation on newer hardware generations. Recent work has made the case for large batch sizes for ResNet (Goyal et al., 2017; Smith et al., 2017), which indirectly affects communication overhead by reducing the number of synchronization rounds per-epoch. However, these techniques lack generality when it comes to DNNs other than ResNet and there continues to be a debate in the machine learning community with regard to their efficacy (Masters & Luschi, 2018; LeCun, 2018).

Crucially, we find that even within a single high-performance server like the DGX-1 (dgx1), communication overheads are amplified due to one of the main shortcoming of existing communication libraries like NCCL or Horovod: their inability to handle topology heterogeneity. These libraries typically use a fixed ring-based scheme for doing data transfers. However, ring-based protocols have structural limitations: for each ring, every node can only have one input and one output. This strong restriction makes it impossible for rings to fit into irregular topologies caused due to scheduler allocations (Figure 2) and this leads to link under-utilization as shown in Figures 3 and 4.

Figure 5 shows the communication overhead (best-to-worst-case range), as a percentage of per-iteration time, for four popular image classification DNNs within a DGX-1V when using NCCL[1]. Given n GPUs there could be many n GPU configurations. We bin these configurations by *topology uniqueness*. For example, a 4 GPU configuration consisting of GPUs [0, 1, 2, 3] is in the same bin as the [4, 5, 6, 7] configuration. We pick one representative configuration from each bin and report the best and worst case overheads for each of the n GPU configuration. Figure 5 highlights that the communication overheads can be as high as 50% for these DNNs on a DGX-1V.

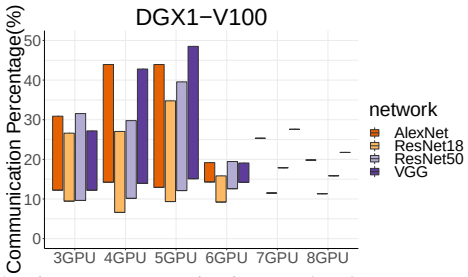


Figure 5. Min-Max communication overhead (percentage of overall runtime) for different DNNs when using NCCL on DGX-1V.

By modeling the links between GPUs as a graph, classic results from Edmonds (Edmonds, 1973) and Lovasz (Lovasz, 1976) show that *packing spanning trees* leads to the maximum flow from a chosen root vertex to all the other vertices in a directed graph. Thus, one-to-many protocols like Broadcast using spanning trees from the root node is a potential option to overcome link under-utilization. In addition to operations like Broadcast that just forward data, communication libraries also need to implement primitives like AllReduce which can be modeled as a reduce-and-forward in one direction (towards the root) followed by a Broadcast in the other direction. But this introduces two important questions which we explore next: How close to line rate can GPUs perform computation on data that is being transferred, and can GPUs support multiple transfer trees efficiently?

2.2 Micro Benchmarks

We validate the potential of computing inline with communication over spanning trees on modern GPU hardware. We do this using a series of micro-benchmarks mimicking transfer patterns when using spanning trees. First we test how deep spanning trees perform as number of the GPUs increases (*depth* tests). Next we test how well *multiple trees* passing through a GPU can transfer data at the same time.

We present our test results from AWS P3.16xlarge EC2 in-

[1]We use NCCL and NCCL2 interchangeably for v2.4.2

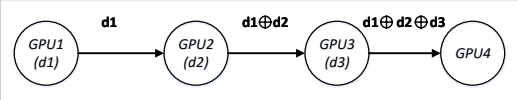


Figure 6. Depth test of Reduce+forward, over a chain of GPUs.

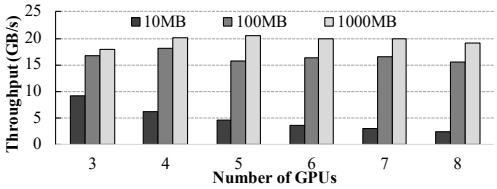


Figure 7. Throughput for Reduce+forward over a chain of GPUs.

stance, a DGX-1V with 8x NVIDIA V100 GPUs connected over an NVLink topology shown in Figure 1. We also ran the same group of experiments on a DGX-1P machine. For the sake of brevity, we do not include those results here.

Depth Test. The first topology class we consider is a depth test where we vary depth of trees that are used. To do this we consider a simple *chain* topology (Figure 6). Given a chain topology, we consider a reduce+forward traffic pattern. Results from other traffic patterns (data forward and Reduce-Broadcast) are included in Appendix A.1. For Reduce+forward (Figure 6), each GPU has its own data. When a GPU receives data from its predecessor, it invokes a reduction function (denoted as ⊕) on the received data with its own data, passing the result to its successor.

We test these operations over different number of GPUs (3-8GPU) and vary data sizes from 10MB to 1000MB (Figure 7). As we increase the chain length, throughput decreases to around 19 GB/s from around 21 GB/s for 1000MB. We also see that throughput drops as the dataset size becomes smaller; it is hard to saturate fast links with small data sizes and the constant overheads in invoking CUDA operations are significant at smaller data sizes.

Multi-transfer Test. Next we consider effect of having multiple transfers simultaneously take place in a given topology. These tests are important to ascertain if we can have multiple data transfers happen in parallel. To do this we consider two topologies: a multi-input, multi-output (MIMO) as Fig. 8(a) and a multi-chain aggregation (MCA) as Fig. 8(b).

In the MIMO topology, two nodes on the left concurrently send data to the center node. The center node aggregates its local data (d3, d3') with received data blocks (d1, d2) respectively, and then forwards the aggregated result (d1⊕d3, d2⊕d3') to two different destinations. We test performance with multiple dataset sizes as shown in Figure 8(c). We find that for datasets larger than 10MB, we can achieve around 18GB/s throughput, which is around 15% lower than maximum throughput on NVLink Gen2.

In the MCA topology (Figure 8(b)), we consider a center node that merges two reduce+forward chains. Figure 8(c) shows that MCA has roughly the same throughput as MIMO and achieves around 18 GB/s for data larger than 10 MB.

Summary. From the micro-benchmark results, we see modern GPUs with NVLink interconnects provide good support for deep and broad trees while forwarding data. We also see that GPUs can perform reductions while forwarding data and also support multiple transfers at the same time. While these scenarios do show some drop in performance compared to pairwise NVLink transfers, this drop is only minor, and the resultant throughput is much higher than that achievable when using PCIe. Overall, these results make it promising to explore the use of spanning trees to implement collective communication protocols.

2.3 `Blink` Approach

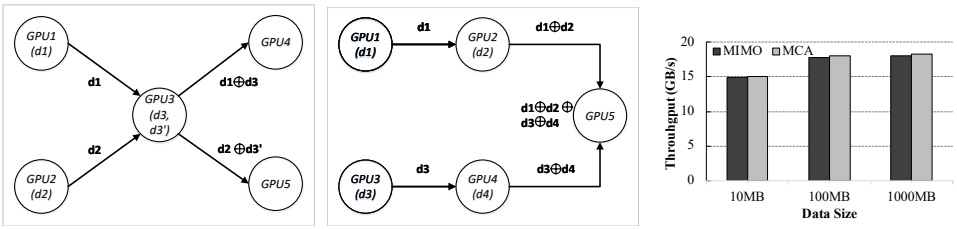
We next outline our approach to building high performance collective communication primitives in `Blink` and present an end-to-end workflow as shown in Figure 9.

Our main approach in `Blink` is to *dynamically* generate the appropriate collective communication primitives to make it best utilize a given topology. We achieve high utilization by packing spanning trees and use algorithms that can maximize the transfer rate achieved while minimizing the number of trees used. Finally, we implement many-to-many algorithms like AllReduce by performing many-to-one and one-to-many operations on each direction of bi-directional links. The workflow of using `Blink` consists of:

- At *runtime*, once a deep learning job has been scheduled and assigned a set of GPUs, `Blink` is able to probe the topology of the machine and infer the interconnect topology across only the GPUs allocated.
- Once we have the topology, we model collective communication operations as flows on a directed graph and compute the maximum fractional packing of spanning trees. We denote this step as `TreeGen` and this step outputs a set of spanning trees and weights corresponding to how much data should be sent over them.
- Next, `CodeGen` parses the spanning trees and *generates* CUDA code. The code generated matches the API offered by NCCL and is packaged into a shared library `libblink.so`.
- Finally we set the `LD_PRELOAD` flag to dynamically load the `Blink` implementations when the main program in invoked. This ensures that existing programs can be run without any modification.

3 DESIGN

In this section we outline the design of `Blink` and describe our techniques for creating protocols that address the dual challenges of high link utilization and heterogeneous topologies. We first study one-to-many protocols like Broadcast



(a) MIMO (b) MCA (c) MIMO, MCA Throughput

Figure 8. MIMO, MCA topology and test throughput.

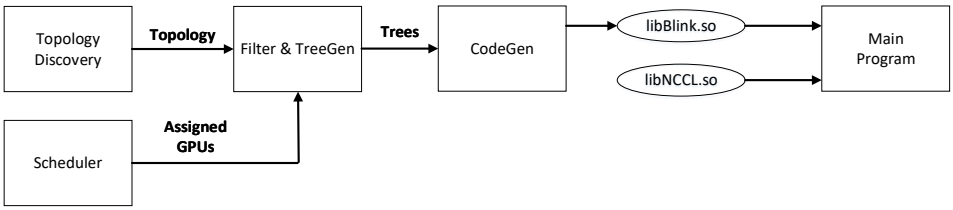


Figure 9. Blink toolchain workflow

or Gather and describe our approach to packing spanning trees and the approximation framework we use to efficiently generate spanning trees. Finally we discuss how our techniques can be extended to handle all-to-all protocols like AllReduce.

3.1 Packing Spanning Trees

We first consider the problem of broadcasting data from one root GPU to all the other GPUs in the system. The topology we infer from the allocated resources can be modeled as a directed graph where every GPU is a vertex V and every link (NVLink or PCIe) is marked as a directed edge E. Each directed edge also has a bandwidth proportional capacity.

Given the above model, the optimal rate possible for Broadcast is the maximum weight of flows that originate from a given root vertex r and reach all the other vertices in the graph. This problem is well studied in graph theory (Edmonds, 1973) and prior work has shown that the optimal rate can be achieved by finding the maximal packing of a number of *directed spanning trees* or arborescences in the graph (Lovasz, 1976). Each arborescence T_i originates at the root vertex and follows directed links to span every other vertex. Thus the problem of finding the optimal schedule for Broadcast can be solved by finding the set of maximum weight arborescences that satisfy the capacity constraints.

$$\max \sum_i w_i \tag{1}$$

$$\text{such that } \forall e \in E, \sum_i \kappa_i * w_i < c_e \tag{2}$$

$$\text{where } \kappa_i = \begin{cases} 1, & \text{if } e \in T_i \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

More formally, our problem statement is given a graph G with vertices V, edges E and root vertex r and spanning trees $T_1, T_2, T_3 ... T_i$ we wish to find the weights w_i such that the sum of weights trees passing through any edge does not exceed the capacity of the particular edge.

While the above formulation can be viewed as an optimization problem, the number of arborescences in a graph can be exponentially large ($O(n^{n-2})$ for a complete graph) and hence is not a practical model to use. A number of more efficient exact algorithms (Gabow & Manu, 1998) have been proposed for this problem but their running time is still $O(n^3 m log(n^2/m))$ for a graph with n vertices and m edges. In this paper we instead use a recently proposed approximate packing scheme and then discuss how we minimize the number of trees used to achieve the optimal rate.

3.2 Approximate Packing

The multiplicative weight update (MWU) is an algorithmic technique that is used in a number of domains ranging from optimization to game theory. Our specific use of MWU here follows a recently proposed algorithm to achieve near-linear time approximation for fractional packing problems (Chekuri & Quanrud, 2017). For the case of packing spanning trees, this approach finds a $(1-\epsilon)$-approximation in $O(m \ln m/\epsilon^2)$, where m is the number of edges.

The MWU procedure for finding the optimal set of packing spanning trees proceeds in the following fashion: We initialize every edge with a capacity and a weight that marks how much of the capacity has been used. Given this, we run an iterative method where at each iteration we find the minimum weight spanning tree given the current assignment. We then increment the weight on this chosen tree by an ϵ factor and update weights on the graph correspondingly. The algorithm provably converges after $O(\ln m/\epsilon^2)$ iterations and on convergence we get a set of directed spanning trees $T_1 ... T_i$ and corresponding weights w_i for each of them. The

total rate for Broadcast will be the sum of weights $\Sigma_i w_i$.

While the MWU procedure has very low execution time and achieves the optimal rate, there is no bound on the number of spanning trees returned. For example we find that with the DGX-1V topology of 8 GPUS, MWU procedure returns 181 spanning trees while the minimum number of trees that can be used to achieve the same optimal rate is 6. The weights on the trees generated by MWU vary from 0.002 to 0.899. Having a larger number of trees will mean that the amount of data transmitted per tree will be much smaller leading to lower throughput (Section 2.2) and higher overhead in scheduling transfers in the generated code (Section 4).

3.2.1 *Minimizing Number of Trees*

We design an integer-linear program based solution to minimize the number of spanning trees that are used. From the above described MWU procedure we get the optimal rate b^* and a set of candidate spanning trees $T_1, ...T_k$. To minimize the number of spanning trees, we formulate an *integer* linear program (ILP) similar to the one presented before but with each weight is restricted to be 0 or 1. This problem can be expressed as

$$\max \sum_{i=1}^{k} w_i \tag{4}$$

$$\text{such that } \forall e \in E, \sum_i \kappa_i * w_i < c_e \tag{5}$$

$$\forall w_i \in \{0, 1\} \tag{6}$$

$$\text{where } \kappa_i = \begin{cases} 1, & \text{if } e \in T_i \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

k here is controlled by the number of trees returned by the MWU procedure and thus is much smaller than the overall number of spanning trees present in the graph. Solving this ILP will yield $\hat{c}$, the maximum rate that is feasible by only using integer capacities for each tree. However $\hat{c}$ might be much lower than c^* and we thus iteratively relax the constraints (i.e. allowing w_i to take fractional values) until $\hat{c}$ is within a configured threshold (e.g., 5%) of c^*.

Using this procedure reduces the number of trees from 181 to 6 for the 8-GPU case in DGX-1V topology with each tree having a rate of 1.0. In terms of data size, this improves the amount of data transferred through a single tree leading to better link utilization. For a 1000MB transfer, each tree will now transfer 166MB while without the ILP the transfer sizes vary from 0.33MB to 148MB.

3.3 Handling many-to-many operations

The above discussion focused on one-to-many operations like broadcast and gather where packing directed spanning trees yields the optimal rate. To handle many-to-many operations we exploit the fact that all the links found in these machines are bi-directional in nature and hence we can create an undirected graph to run a many-to-one primitive using one direction of links and correspondingly run a one-to-many primitive in the other direction. For example, to do an all-reduce operation on the directed graph, we first run a *reduce* operation to a chosen root vertex using the undirected graph and then do a *broadcast* operation from the root vertex using the same tree but with links going in the reverse direction.

This strategy of using two undirected trees also matches the lower bound of number of messages required for AllReduce operations. As shown in prior work (Patarasuk & Yuan, 2009), the minimum number of messages that need to be sent by a process for AllReduce, is $2 \times \lceil \frac{N-1}{N} \rceil$. The spanning tree over N vertices contains $N-1$ edges and accounting for trees in both directions (one for Reduce and one for Broadcast) we similarly have $2 \times (N-1)$ messages. Assuming a continuous forwarding model (similar to our benchmarks in Section 2.2), messages sent by all N processes simultaneously and we can thus achieve a similar bound of $2 \times \lceil \frac{N-1}{N} \rceil$ messages per process.

3.4 DGX-2 and Multi-server settings

We next extend our design to switch-based settings like DGX-2 and multi-machine training. The DGX-2 consists of 16 V100 GPUs connected over NVSwitch; each GPU is connected to the switch over 6x NVLinks (150GBps bidirectional throughput). On the DGX-2, NCCL constructs binary trees for small dataset sizes ($< 16KB$) and rings for larger datasets. In contrast, on the DGX-2, Blink's generated spanning trees for AllReduce (Reduce-Broadcast) are deceptively simple: with m GPUs, each GPU acts as a root for $1/m$ of the data chunks and each root is directly connected to $(m-1)$ leaf nodes, resulting in m one-hop trees. Blink's one-hop trees have a significant latency and throughput advantage over NCCL's double-binary trees for smaller dataset sizes; we show this quantitatively in Section 5.2.

When the GPUs of a training task span multiple servers, connected over a switch or a hierarchy of switches, Blink uses a three phase protocol (example of two 4-GPU machines in Appendix A.2). The first phase consists of a per-server reduction over local spanning trees – the root of each tree within each server aggregates data from its children as before. The second, new, phase consists of cross-server Reduce-Broadcast (similar to within the DGX-2) – across n servers, there are n one-hop cross-server trees, with each server-local root connected to $(n-1)$ roots on other servers. The third phase consists of each server-local root Broadcasting the result of the second phase to all nodes in their server. We evaluate our multi-server protocol in Section 5.3.

4 IMPLEMENTATION

In this section, we first discuss our code generation implementation and discuss how choosing the appropriate chunk size is important to achieve good performance.

4.1 CodeGen Implementation

For ease of illustration, we discuss two types of collective communications: Broadcast and AllReduce. We note that these are the most frequently used primitives by deep learning workloads and other collective primitives follow similar patterns. For example, Gather is the inverse of Broadcast, and AllGather is AllReduce without using a reduction function.

Broadcast: We first parse the spanning trees generated by the procedure described in Section 3, with each spanning tree having a different weight associated with it. Once we receive the input buffer to be Broadcast from the root node, we split the buffer among all the spanning trees based on their weights.To perform data transfer on a link in the tree, we issue a `cudaMemCpy` command from the source to the destination GPU. To reduce latency, instead of transmitting all the data assigned to this tree at once, we further divide data in each tree into multiple small chunks. Once a chunk has been transferred, we issue a CUDA event to notify the destination. To enable parallel transfers across trees, we use CUDA streams and by using a stream per link, per tree we can achieve high utilization.

AllReduce: As described in Section 3.3, we execute AllReduce by leveraging bi-directional links. We perform reductions in one direction to a root node. Once the root node computes the final reduce result, it is Broadcast in the reverse direction. We implement all the reduction functions supported by NCCL (e.g. min, max, etc.) as CUDA kernels.

4.2 CodeGen Optimizations

We next discuss two issues we faced during `Blink` implementation that stem from limitations of existing hardware.

4.2.1 *Automatic chunk size selection*

Within each CUDA stream, a *chunk* is our atomic unit for data copy / synchronization between sender and receiver. For spanning trees, chunk size is an important factor in determining overall latency, because each node cannot start forwarding until it receives a complete chunk from its predecessor. Figure 10 shows a simple example in a four GPU scenario. Splitting data into two chunks reduces transfer time by a third when compared to a setting with no chunking. Our goal is to parallelize (pipeline) data transfers while minimizing multi-hop latency. Thus intuitively, making the chunk size small should improve performance and link utilization. However for each chunk we need to issue at least three CUDA commands for copying/synchronization and having a large number of small chunks leads to increased overhead in scheduling these commands.

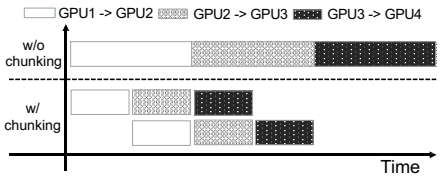


Figure 10. Data chunking to reduce multi-hop latency.

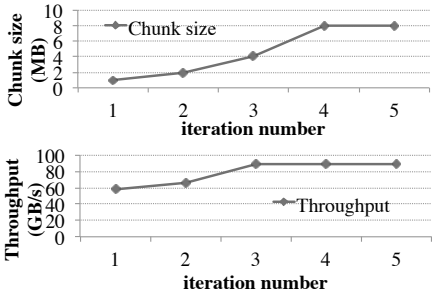


Figure 11. Automatic chunk size selection with MIAD (multiple-increase, additive-decrease.)

Thus we use an adaptive scheme to automatically select the chunk size. As machine learning models are typically run for a large number of iterations, we observe that we can use the first few iterations to explore how changing the chunk-size affects overall performance. This is necessary as in our experience the optimal chunk size varies based on the data size, number of spanning trees in the topology and maximum depth of each tree.

Our algorithm follows a *multiplicative increase, additive decrease* (**MIAD**) scheme across iterations. We initialize the chunk size with a small value and increase the chunk size by a multiplicative factor as long as the measured throughput is increasing. If the throughput decreases we additively decrease the chunk size until we reach a steady state. Figure 11 shows an example execution of our chunk size selection algorithm when running Broadcast over 4 GPUs. Here, we start with a chunk size of 1MB and multiplicatively increase it by 2× on every iteration. We find that after four iterations the throughput stabilizes to the optimal value.

4.2.2 *Link Sharing*

One of the other challenges with using multiple trees on existing hardware is that the CUDA functions do not provide any direct control on how links are shared. For example if say there are two trees with weight 0.5 that are passing through the same link, then a fair sharing scheme would transmit one chunk from the first tree followed by one chunk from second tree. However in our experiments we find that the CUDA implementation does not always result in fair sharing and that chunks from one of the trees could be arbitrarily delayed. This introduces gaps in the forwarding

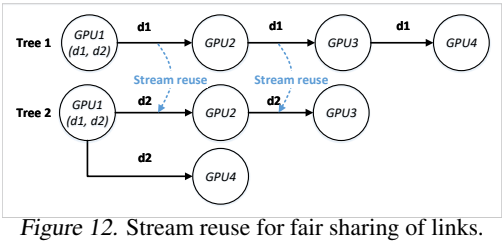


Figure 12. Stream reuse for fair sharing of links.

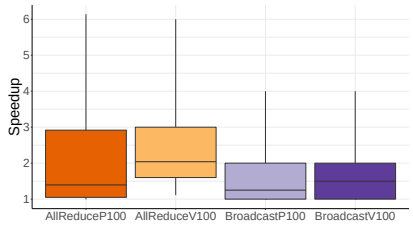


Figure 13. Theoretical speedups from packing spanning trees compared rings on P100 and V100. Boxplot shows a distribution for possible configurations and whiskers show 5th and 95th percentile.

pipeline and harms the effective throughput achieved.

Since ordering guarantees are only provided by CUDA streams, we address this problem by reusing CUDA streams when the same link is used in multiple trees at roughly the same position.

For example, as shown in Figure 12, we have two spanning trees both starting from GPU1, which contain two data pieces (d1 for tree1, d2 for tree2). Once we have created streams for first tree, we compare pairwise link positions between the two trees. Note that link GPU1 <–>GPU2 (first hop from the source) is in the same position on both trees. Thus when creating streams for tree 2, instead of initializing a new stream, we re-use the stream from tree 1 and schedule transfers to ensure fair sharing.

5 Evaluation

In this section, we evaluate `Blink`'s performance along three fronts. First, we discuss the benefits of packing trees and present theoretical comparisons between `Blink` with NVIDIA NCCL, the start-of-the-art ring-based collectives library. Second, we show experimental results highlighting throughput comparison between NCCL and `Blink` for Broadcast and AllReduce on three different hardware settings (DGX-1P, DGX-1V, DGX-2). Third, we provide end-to-end speed-up results of using `Blink` with four popular DNNs on both single DGX-1 and multi-DGX-1 settings. We also present results from combining transfers over PCIe and NVLink in Appendix A.3.

5.1 Tree Packing Benefits

We first evaluate the theoretical benefits of packing spanning trees vs. a ring-based approach used by libraries like NCCL. We compare the number of rings that are created in a given topology by NCCL and the total weight of spanning trees packed by `Blink` for all possible allocations from 3 GPUs to 8 GPUs on both the V100 and P100 machine. We translate this to a Broadcast rate using the lower bounds on messages required for Broadcast $\lceil \frac{N-1}{N} \rceil$ and AllReduce ($2 \times \lceil \frac{N-1}{N} \rceil$). That is given 4 rings for the 8 GPU case, each ring will operate at $\frac{8}{14}$ of link bandwidth and with 4 such rings our effective rate is $\frac{32}{14}$. We approximate the bandwidth for PCIe rings to have half as much bandwidth as NVLink.

Figure 13 shows the distribution of speedups we can achieve by packing spanning trees. We see in all cases packing spanning trees should be at least as fast as using rings and that in some cases (i.e. where rings have to go through PCIe), we can achieve up to 6x speedup. We note that our speedups could be higher in practice due to PCIe performing worse than our model or lower due to chunking overheads.

5.2 Broadcast, AllReduce Micro-benchmarks

We next compare the performance of `Blink` with state-of-the-art NCCL2 on the two most frequently used collective primitives, namely Broadcast and AllReduce. Considering the topology (Figure 1), and accounting for the different number of GPUs in use and their positions, we have 46 different topology settings for DGX-1V, and 14 different topology settings for the DGX-1P machine. For both Broadcast and AllReduce (Fig 14, Fig 16), the number list on x-axis indicates the allocated GPUs in each configuration.

NVLink Broadcast. We provide Broadcast throughput comparison between NCCL and `Blink` for all possible topologies induced by GPU allocations on a DGX-1 (V100) on AWS (p3.16xlarge). The number of GPUs we use range from 3 to 8. To fully saturate our interconnects, we test with a total data size of 500MB (50MB to 1000 MB error-bars).

In Fig. 14, `Blink` can achieve up to 6× (2x geometric mean) speed up in performance compared to NCCL. In the cases where GPUs are not fully connected over NVLink (e.g. GPU 1,4,5,6, as shown in Figure 1), NCCL cannot form NVLink-only rings across these GPUs, thus forcing it to fall back on using PCIe for data transfers. This results in many links going unused, leading to dramatically lower throughput. NCCL matches `Blink` when it can form a fully connected NVLink ring and when `Blink` can only create one spanning tree (e.g., when using GPU 2,3,6,7, as depicted in Figure 1, NCCL2 can form one bi-directional ring: GPU2<–>GPU6<–>GPU7<–>GPU3<–>GPU2). However, even in these cases, `Blink` still achieves 3-5 GB/s higher performance due to optimized chunked transfers.

Given the topology difference of DGX-1P and DGX-1V, we repeat the throughput comparison on DGX-1P. We have 14 unique topology configurations (Figure 15) and we see similar throughput gains as DGX-1V. Overall, `Blink` achieves up to 3x speed up (1.6x geometric mean) over NCCL.

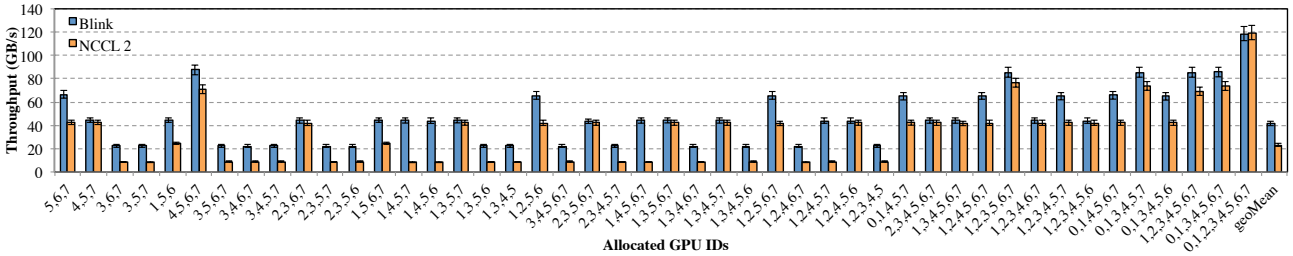


Figure 14. Broadcast throughput comparison between NCCL2 and Blink for all unique topologies on DGX-1V.

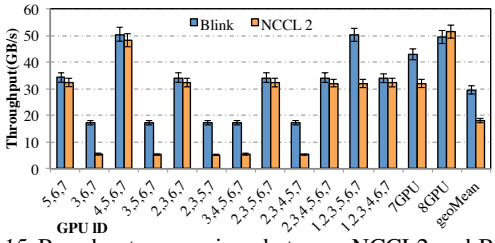


Figure 15. Broadcast comparison between NCCL2 and Blink in all possible topologies on DGX-1P.

NVLink AllReduce. Compared to Broadcast throughput in Figure 14, AllReduce achieves lower performance for all 46 configurations for both NCCL and Blink (Figure 16). This is consistent with the micro-benchmark results from Section 2.2. For example, in the 3 and 4 GPU settings on the DGX-1 (V100), AllReduce achieves an average 20-30GB/s less than corresponding Broadcast settings. For the 8 GPU configuration, AllReduce only achieves half of the corresponding Broadcast throughput for both NCCL and Blink. For NCCL's AllReduce, each data chunk needs to go through the ring twice, once for Reduce then for Broadcast, which leads to roughly half the performance. Similarly for `Blink`, Reduction takes place in one direction of the spanning tree, and Broadcast in the other direction.

For AllReduce, `Blink` outperforms NCCL with up to 8× (2× geometric mean) speed up in throughput. Similar to Broadcast, `Blink` has higher throughput gains in the cases where NCCL cannot form NVLink rings over the allocated GPUs or has to drop some links due to the constraint of forming rings. Results from DGX-1P also closely match these findings and we omit them here due to space constraints.

DGX-2 AllReduce. We next compare Blink to NCCL when using 16 GPUs on a DGX-2 machine. As described in Section 3.4, with single-hop trees on the DGX-2, Blink is especially effective for smaller data sizes offering lower latency and higher throughput, compared to NCCL's double-binary trees and rings. Blink can get up to 3.32× lower latency (Appendix A.4, Figure 27) and up to 3.5× better AllReduce throughput (Figure 18) than NCCL.

5.3 End-to-end Training

We incorporate `Blink` with PyTorch (Paszke et al., 2017), and evaluate the end-to-end performance gains for training. We use four popular CNNs: AlexNet, ResNet18, ResNet50 and VGG16 and train these models on ImageNet-1K (ILSVRC12) dataset (Russakovsky et al., 2015). For all models, we use the same *per-GPU* mini-batch size and hyper-parameters used in the original papers.

Single server training. We evaluate these models by training them over 3 to 8 GPUs on the DGX-1 (V100). For a fixed number of GPUs, we pick multiple configurations where appropriate, but to save space, we limit ourselves only to a subset of the unique configurations from before. Specifically, from Figure 16, for configurations with n GPUs, if we have more than one configuration, we pick ones where the speed-up of `Blink` over NCCL is unique. As shown in Figure 17, switching collective communication backend from NCCL2 to `Blink`, can reduce up to 40% time spent in end-to-end DNN training iterations (6.3% geometric mean) , and achieve up to 87% communication time reduction (31% in geometric mean).

Multi-server training. `Blink`'s multi-server AllReduce consists of a per-server reduction over spanning trees (t_1), cross-server Broadcast and Reduce (t_2), followed by a Broadcast within each server as before (t_3). We consider scenarios where the GPU allocation is fragmented across machines, prevalent in multi-tenant clusters as shown in Figure 2. For example we consider a 8GPU job spread across two DGX-1V servers with 3 and 5 GPUs allocated respectively. Figure 19(a) shows that `Blink` outperforms Horovod with NCCL/MPI by up to 11%. `Blink`'s reduction in improvement over NCCL, compared to the gains in single-server training, stem from commodity cloud interconnects. In commodity networks, inter-server AllReduce throughput (40Gbps) is much lower than intra-server throughput (40GBps). Thus while `Blink` can reduce t_1 and t_3, there isn't much that can be done for t_2.

To understand how faster interconnects will change performance, we present results from a simulation varying the cross-machine bandwidth (Figure 19(b)). We compare AllReduce throughput for 100MB of data and see that as cross-machine bandwidth increases (Thomas et al., 2018; Verizon-400Gbps), `Blink`'s design will lead to more pronounced end-to-end benefits. NCCL is bound by intra-server PCIe throughput where as Blink can keep up with inter-server throughput until the intra-DGX-1V NVLinks become a bottleneck (for the 3-5 GPU case this is ∼300Gbps).

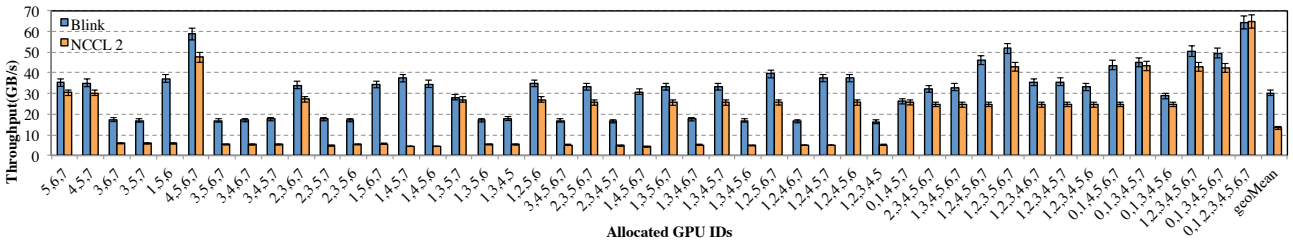


Figure 16. AllReduce throughput comparison between NCCL2 and Blink for all unique topologies on DGX-1V.

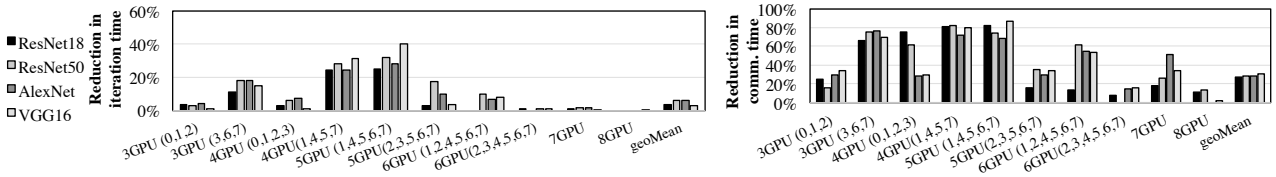


Figure 17. Blink end-to-end training time reduction (ImageNet1K) within a DGX-1V machine.

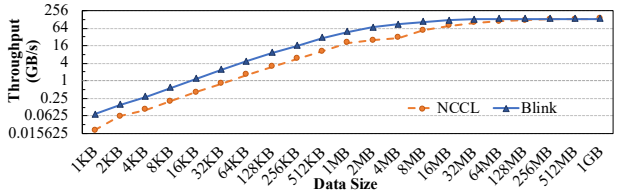
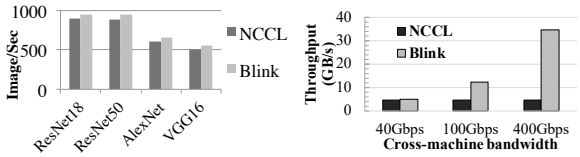


Figure 18. AllReduce (`Blink` and NCCL2) on a 16-GPU DGX-2.



(a) Using 2 DGX-1Vs (b) AllReduce Projections

Figure 19. Multi-DGX-1 DNN training with `Blink`.

6 Related Work

Work on collectives fall in one of two buckets (below):

Topology-fixed Schemes. Basic collective operations (e.g. Broadcast, AllReduce) are fully supported in the MPI (Message Passing Interface) standard (Blaise Barney, 2018). Earlier work has mainly focused on designing optimal collectives over regular, well-defined network structures like hypercube (Scott, 1991; Bhuyan & Agrawal, 1984), full mesh (Barnett et al., 1993), etc. Recent work has looked at more general networks, with optimizations for scenarios when number of communication nodes are not power of two (Thakur et al., 2005), and automatic algorithm selection for a specific system architecture (Vadhiyar et al., 2000).

Under specific network settings, there are many algorithms that achieve better performance than MPI. For example, the latency-optimal all-reduce solution, "butterfly algorithm" (van de Geijn, 1994; Zhao & Canny, 2013), divides all-reduce into two steps: first is a recursive reduce-scatter and then followed by a recursive all-gather. But, the communication pattern of butterfly algorithms often cause network contention, which makes it less practical. Within a tree or ring topology, ring-based collectives were shown to be bandwidth optimal in homogeneous network settings (Faraj et al., 2008; Patarasuk & Yuan, 2009). Several companies have developed their own implementations of this algorithm, such as Horovod (Sergeev & Balso, 2018) from Uber, Baidu Ring All-Reduce (Ng, 2017), Facebook's Gloo (Noordhuis, 2017), IBM Power AI DDL (Hunter, 2017). However, they all operate under the assumption of a fixed topology, which is not a good fit for cloud computing where topology may change dynamically. `Blink` is designed to handle irregular topologies and yield optimal solutions.

Topology-aware Protocols. Techniques that exploit hierarchy in wide area networks for collective communication center around the idea of minimizing data transfer over slow (wide-area) links (Karonis et al., 2000; Kielmann et al., 1999). The same idea has been extended to cloud environments where node locality is determined by pairwise network bandwidth measurements (Gong et al., 2015). Smelt adopts similar idea in NUMA multi-core environment (Kaestle et al., 2016). Blueconnect decouples AllReduce into ReduceScatter and AllGather, pipelining these two sub-operations (Cho et al., 2019). However it only works on symmetric topologies, making it less flexible than `Blink` spanning trees. Recent literature (Wang et al., 2018a;b) also tries to optimize link utilization within a DGX-1 box via spanning-tree scheme. `Blink` is general and is optimized for multi-GPU/machine collective communication, over symmetric or asymmetric topologies, and can combine heterogeneous links (such as PCIe, NVLink, Ethernet, InfiniBand) for data transfer.

7 Conclusion

`Blink` is a fast and generic collective communication library to accelerate distributed ML. To handle topology heterogeneity prevalent in modern GPU hardware, `Blink` dynamically packs spanning trees to maximize link utilization. Compared with state-of-the-art, ring-based protocols like NCCL2, `Blink` achieves up to 8× faster model synchronization and reduces end-to-end training time by up to 40%.

ACKNOWLEDGEMENTS

Guanhua Wang and Ion Stoica are supported by a NSF CISE Expeditions Award CCF-1730628, and their research was also supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. Shivaram Venkataraman is also supported by a Facebook faculty research award and support for this research was also provided by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation.

Additionally, we thank the MSR Lab LT, especially Ricardo Bianchini and Donald Kossmann, for their enthusiastic and unwavering support of Project Fiddle, and for their generous support in procuring the many resources required to develop and evaluate Blink. We also thank the MSR GCR staff, especially Jim Jernigan and Steven Dahl, for supporting our DGX-1, DGX-2 needs.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

Barnett, M., Littlefield, R., Payne, D., and van de Geijn, R. Global combine on mesh architectures with wormhole routing. In *Proceedings of the 7th International Parallel Processing Symposium*, 1993.

Bhuyan, L. N. and Agrawal, D. P. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 1984.

Blaise Barney. Message Passing Interface. https://computing.llnl.gov/tutorials/mpi/, 2018.

Chekuri, C. and Quanrud, K. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 801–820. SIAM, 2017.

Cho, M., Finkler, U., Kung, D., and Hunter, H. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *sysML*, 2019.

dgx1. NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/, 2017.

dgx2. NVIDIA DGX-2. https://www.nvidia.com/en-us/data-center/dgx-2/, 2018.

Edmonds, J. Edge-disjoint branchings. *Combinatorial algorithms*, 1973.

Faraj, A., Patarasuk, P., and Yuan, X. Bandwidth efficient all-to-all broadcast on switched clusters. *International Journal of Parallel Programming*, 2008.

Gabow, H. N. and Manu, K. Packing algorithms for arborescences (and spanning trees) in capacitated graphs. *Mathematical Programming*, 82(1-2):83–109, 1998.

Gong, Y., He, B., and Zhong, J. Network performance aware mpi collective communication operations in the cloud. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.

Goyal, P., Dollar, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

Hunter, H. IBM Research achieves record deep learning performance with new software technology. https://www.ibm.com/blogs/research/2017/08/distributed-deep-learning/, 2017.

InfiniBand. Introduction to InfiniBand. https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf, 2007.

Jeaugey, S. Optimized inter-GPU collective operations with NCCL 2. https://developer.nvidia.com/nccl, 2017.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. *Microsoft Research Technical Report (MSR-TR-2018-13)*, 2018.

Kaestle, S., Achermann, R., Haecki, R., Hoffmann, M., Ramos, S., and Roscoe, T. Machine-aware atomic broadcast trees for multicoress. In *USENIX OSDI*, 2016.

Karonis, N., de Supinski, B., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium*, IEEE IPDPS'00, 2000.

Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM PPoPP'99, 1999.

LeCun, Y. Training with large minibatches is bad for your health. https://twitter.com/ylecun/status/989610208497360896?lang=en, 2018.

Lovasz, L. On two minimax theorems in graph. *Journal of Combinatorial Theory, Series B*, 21(2):96–103, 1976.

Masters, D. and Luschi, C. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018. URL http://arxiv.org/abs/1804.07612.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N., Granger, G., Gibbons, P., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM Symposium on Operating Systems Principles (SOSP 2019)*, October 2019.

NCCL 2.4. Massively Scale Your Deep Learning Training with NCCL 2.4 . https://bit.ly/2lFwFQ4, 2019.

Ng, A. Bringing HPC Techniques to Deep Learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning/, 2017.

Noordhuis, P. Accelerating machine learning for computer vision. https://github.com/facebookincubator/gloo, 2017.

NVLink. NVIDIA NVLINK. http://www.nvidia.com/object/nvlink.html, 2017.

NVSwitch. NVIDIA NVSWITCH. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, 2018.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *Proceedings of the 31st Conference on Neural Information Processing Systems*, NIPS'17, 2017.

Patarasuk, P. and Yuan, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, pp. 117–124, 2009.

PCI Express. PCI Express: An Overview of the PCI Express Standard. http://www.ni.com/white-paper/3767/en/, 2014.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

Scott, D. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proceedings of the 6th Distributed Memory Computing Conference*, 1991.

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

Smith, S. L., Kindermans, P., and Le, Q. V. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 2005.

Thomas, S., Voelker, G. M., and Porter, G. Cachecloud: Towards speed-of-light datacenter communication. In *USENIX hotcloud 2018*, 2018.

Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, 2000.

van de Geijn, R. On global combine operations. In *Journal of Parallel and Distributed Computing*, 1994.

Verizon-400Gbps. Verizon marks milestone with successful 400G technology trial. https://bit.ly/2lKgAs7, 2018.

Wang, G., Phanishayee, A., Venkataraman, S., and Stoica, I. Blink: A fast nvlink-based collective communication library. In *sysML*, 2018a.

Wang, L., Li, M., Liberty, E., and Smola, A. J. Optimal message scheduling for aggregation. In *sysML*, 2018b.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 595–610, Carlsbad, CA, 2018. USENIX Association.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 181–193, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6.

Zhao, H. and Canny, J. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, 2013.

A APPENDIX

A.1 Micro Benchmarks (DGX-1V)

We continue our discussion of micro benchmarks from Section 2.2, highlighting results for forwarding on a chain and fan in/out tests.

A.1.1 Depth Test

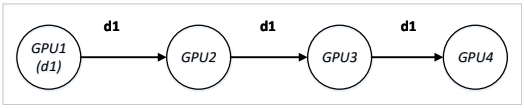
For the forwarding benchmark (Figure 20(a)), GPU1 is the source node with data named d1, and it passes the data d1 to GPU2 and then GPU2 forwards it to GPU3 etc. For "reduce+broadcast" (Figure 20(c)), we perform "reduce+forward" in one direction and "forward" in the other direction, as such a capability can be used for all-to-all reductions.

A.1.2 Breadth Test

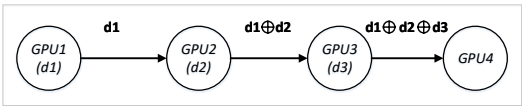
As illustrated in Figure 22(a), in fan-in forward, a center node (i.e. GPU4) collects data from multiple nodes and then forwards the collected data to its successor. Instead of just forwarding data, in the case of fan-in reduce+forward (Figure 22(b)), the center node computes a reduction function over the incoming data and its own data, then forwards the result to it successor. Fan-out forward (Figure 22(c)), is just the reverse of fan-in forward, in which the center node receives data from one node (i.e. GPU5), then multicasts the received data to its successors (i.e. GPU 1,2,3).

We experiment with different data size as we vary the number of GPUs that serve as fan-in source nodes or fan-out destination nodes. For DGX-1s, the maximum fan-in and fan-out degrees are limited to three. For brevity, we omit the graphs and highlight the key findings. Similar to the depth tests, with data size >50MB, fan-in and fan-out forward achieves near maximum throughput. Compared with fan-in forward, the throughput of fan-in reduce+forward decreases 1-2 GB/s on average due to the latency of launching reduction function kernels on the center node (GPU4).

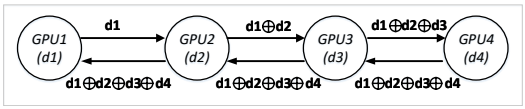
Figure 22 depicts result of breadth tests with different data size as we vary the number of GPUs that serve as fan-in source nodes or fan-out destination nodes. We'd like to note that for the given topology of V100, the maximum fan-in and fan-out degrees are limited to three. In Figure 22(a), with data size >50MB, in all three cases, fan-in forward achieves near maximum throughput. Compared with fan-in forward, the throughput of fan-in reduce+forward (in Figure 22(b)) decreases 1-2 GB/s on average due to the latency of launching reduction function kernels on the center node (GPU4). We also note that running with 1000MB and a fan-in of 3 requires allocating memory for each incoming link and this exceeds the amount of memory available. Finally, for fan-out forward in Figure 22(c), the throughput is again close to the peak link bandwidth.



(a) chain forward

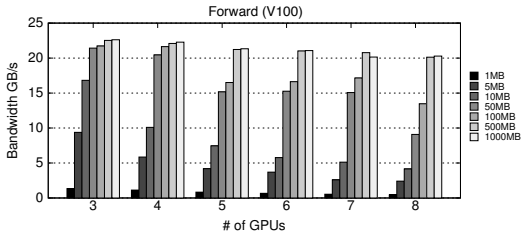


(b) chain reduce+forward

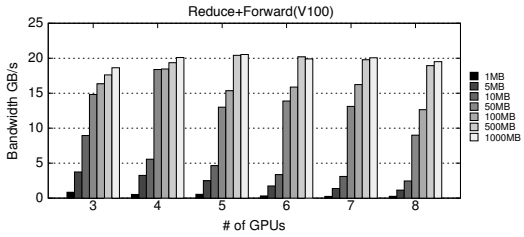


(c) chain reduce-broadcast

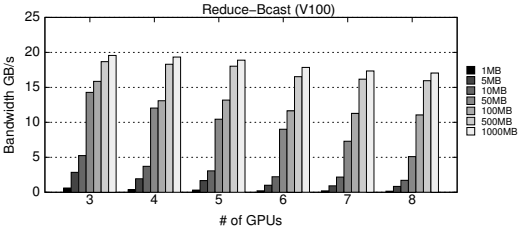
Figure 20. Depth test over a chain of GPUs.



(a) chain forward throughput



(b) chain reduce+forward throughput

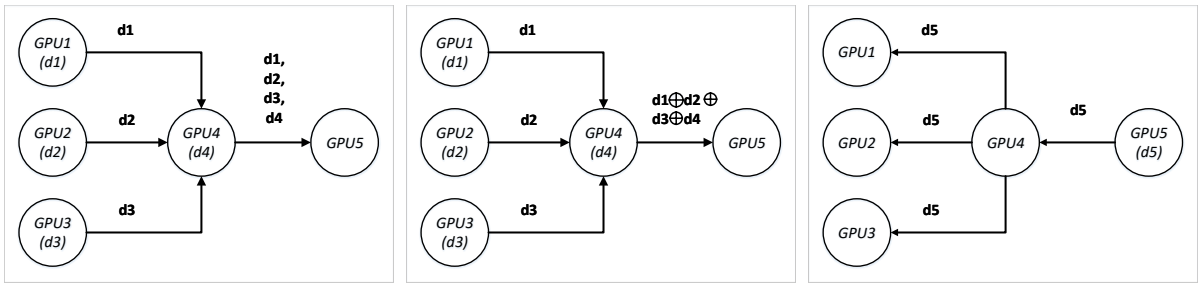


(c) chain reduce-broadcast throughput

Figure 21. Depth test throughput over a chain of GPUs.

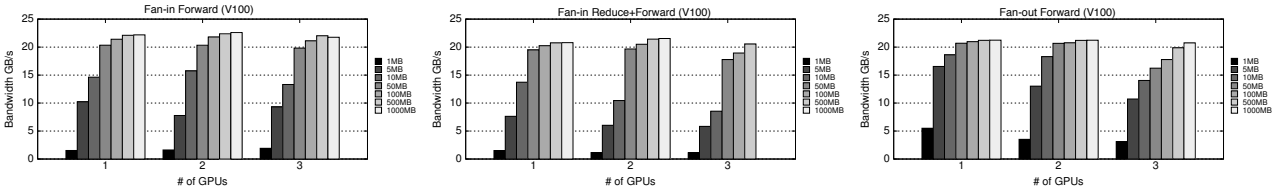
A.2 Three-phase AllReduce protocol for cross-machine settings

As shown in Figure 24, we first partition data based on the number of spanning trees we have (i.e. 4 in this case). Data item $X_{m.g}$ refers to data partition X on server m and GPU g. Each data partition has a distinct server-local root. Figure 24 shows the reduction (function is denoted as $+$) for partition



(a) Fan-in forward (b) Fan-in reduce+forward (c) Fan-out forward

Figure 22. Breadth test of data forward, reduce+forward in fan-in and fan-out topologies.



(a) Fan-in forward throughput (b) Fan-in reduce+forward throughput (c) Fan-out forward throughput

Figure 23. Breadth test throughput for Fan-in forward, Fan-in reduce+forward, Fan-out forward.

B which has a root at $GPU2$. Similar protocol is followed for other data partitions (e.g. A, C, D).

A.3 Exploiting Link Heterogeneity

For intra-node communication, servers such as the DGX-1 have both inter-GPU point-to-point (P2P) interconnects such as NVLink (NVLink) and shared interconnects such as PCIe (8-12GB/s) (PCI Express). PCIe connects multiple GPUs to each other within a machine, and to the CPU and IO deices, through a PCIe switch hierarchy. For inter-node communication, servers are equipped with multiple Ethernet or InfiniBand (InfiniBand) ports with a throughput of 3GB/s and 7GB/s per-port respectively. State-of-the-art collectives, such as NCCL and Horovod, all use ring-based protocols which fail to leverage link heterogeneity. The throughput of a ring is limited by the link with lowest bandwidth and hence these protocols either restrict themselves to high bandwidth, homogeneous links, or limit throughput to the link with lowest bandwidth in the ring. For example, for multi-GPU communication within a machine, NCCL prioritizes using only NVLink over PCIe, as PCIe will be the bottleneck if included in a NVLink ring. Figure 25 shows an example 3 GPU setup for a `Broadcast` from GPU 0: when fully connected with NVLink, NCCL builds two rings (0–>1–>3–>0 & 0–>3–>1–>0) using bi-directional NVLinks, and ignores PCIe. If we replace GPU3 with GPU4, the lack of NVLink between GPUs 1 and 4 prevents NCCL from constructing NVLink-only rings and it has to fall back on PCIe based communication.

To handle heterogeneous links, `Blink` simultaneously transfers data on PCIe and NVLink within a machine and and balances the amount of data transferred across hybrid links. We next discuss how we handle hybrid PCIe and NVLink topologies in the context of our design presented above. The main challenge in using both PCIe and NVLink comes from the fact that NVIDIA driver does not directly allow users to control access to both links and if NVLinks are detected, the system will automatically enable P2P data transfer among GPUs using NVLinks. In our experience we find that using `cudaDeviceDisablePeerAccess` disables NVLinks and forces data transfer through PCIe links. However this still has the limitation that we cannot construct a unified topology with both sets of links. We address this problem by constructing two separate sets of trees, one over PCIe links and another over NVLinks.

One of the challenges with this approach is to balance the amount of data that is transferred over each link type. Our approach here is to minimize the maximum time taken by each of the transfers i.e. minimize $max(T_{PCIe}, T_{NVL})$.

We denote D_{total} as the total data needs to be transferred, and D_{PCIe}, D_{NVL} as the data size assigned on either PCIe or NVLink respectively. T_{dpa} is the latency for calling the `disable_peer_access()` and we denote BW_{PCIe} and BW_{NVL} as the bandwidth of PCIe and NVLink trees. Given this notation and objective, we can see that the opti-

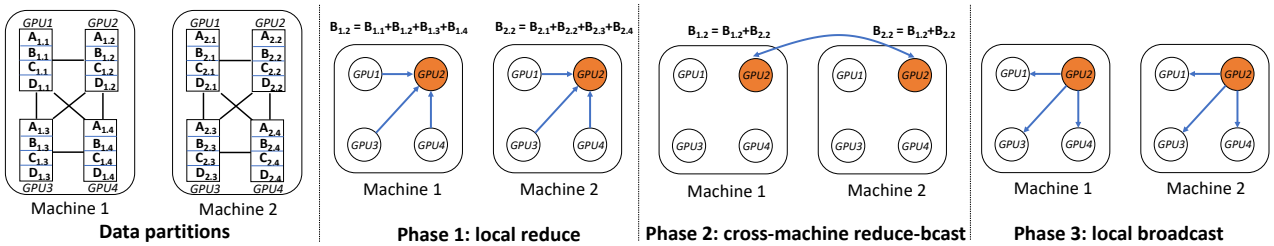
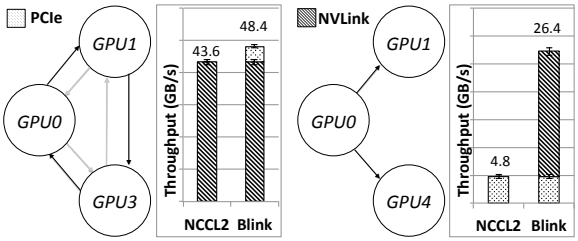


Figure 24. Three-phase AllReduce protocol for cross-machine settings.



(a) Fully connected (b) Partially connected

Figure 25. Broadcast throughput, from GPU 0, using both NCCL and Blink on a DGX-1V.

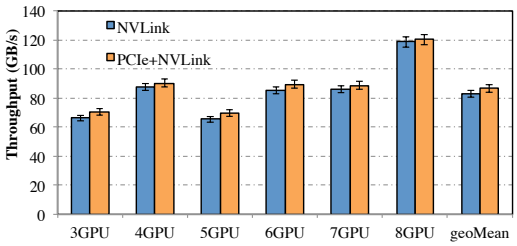


Figure 26. Hybrid and NVLink-only broadcast throughput comparison with varied number of GPUs.

mal data split can be achieved by making $T_{PCIe} = T_{NVL}$.

$$\begin{aligned} &\text{Objective } T_{PCIe} + T_{dpa} = T_{NVL} \\ \implies D_{PCIe} =& \frac{D_{total} \times BW_{PCIe}}{BW_{PCIe} + BW_{NVL}} - \\ & \frac{T_{dpa} \times BW_{PCIe} \times BW_{NVL}}{BW_{PCIe} + BW_{NVL}} \\ D_{NVL} =& D_{total} - D_{PCIe} \end{aligned} \tag{8}$$

The optimal data splits are shown in Equation 8. Note that in Equation 8, T_{dpa} is empirically measured and may vary depending on number of GPUs. We measure this during the initial few calls into our library.

We evaluate hybrid (or combined) data transfers over both PCIe *and* NVLink. For brevity, we only show broadcast results for 3-8 GPUs on the AWS DGX-1V server. Figure 26, highlights the additional 2-5 GB/s performance gain over NVLink-only transfers when `Blink` combines transfers over both NVLink and PCIe. The time to switch communication channels from NVLink to PCIe increases as the number of GPUs grow. For 3 and 4 GPU settings, compared with NVLink-only Broadcast, hybrid transfers can achieve around 5GB/s boost; with 7 and 8 GPUs this boost is only around 2GB/s. This is because the total time spent on enabling and disabling peer-access, i.e. switching between PCIe and NVLink, is proportional to the number of GPU in use.

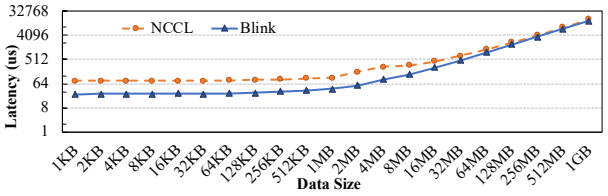


Figure 27. Allreduce Latency in μs (`Blink` and NCCL2) on a 16-GPU DGX-2.

A.4 DGX-2 Allreduce

We present above results comparing latency for AllReduce operations when using 16 GPUs on a DGX-2 machine. As described in Section 3.4, Blink uses a number of single-hop trees to perform AllReduce when GPUs are connected using NVSwitch. One of the main advantages of a single-hop tree is that this reduces latency compared to using a ring across the GPUs. To validate this we measure the latency of AllReduce and vary the dataset size from 1KB to 1GB as shown in Figure 27. We find that Blink is especially effective for smaller data sizes offering up to 3.32× lower latency compared to NCCL's double-binary trees and rings.