Blink+: utilize links across tenant to increase bandwidth

Xiao Song
University of California, Berkeley
xiaosx@berkeley.edu

Yefan Zhou
University of California, Berkeley
yefan0726@berkeley.edu

Yibai Meng
University of California, Berkeley
mengyibai@berkeley.edu

CCS CONCEPTS

• **Software and its engineering → Massively parallel systems**.

KEYWORDS

Blink+, Collective Communication, Across Tenant

1 INTRODUCTION

In modern cloud environments, each physical machine (e.g. one DGX-1 box) may host multiple tenants, which only use part of the computing resources. For example, for 4 GPU machines, tenant one is allocated GPU 0, and tenant two GPU 1, 2, and 3.

Data parallel neural network training [4] [5] is widely used to reduce model training time. Machines need to communicate with each other to exchange data and states from time to time. Existing model synchronization protocols exclusively use links within their own allocated GPU group. In the previous example, tenant two only considers the communication links between GPU 1, 2, and 3 and leaves the connection between GPU 0 and GPU 1, 2, and 3 idle. The communication hardware operates independently alongside the computation cores for modern GPUs. Thus, we are underutilizing valuable bandwidth resources, which is the major bottleneck of distributed machine learning training.

Our project aims to fully utilize bandwidth across tenants by extending the NCCL package. We are able to increase the bandwidth in a mock multi-tenant setting by at most 200%.

This paper is organized into 5 Sections. In section 1, we briefly overview the Blink+ idea. In section 2, we go through prior work and how our project utilizes ideas from prior work. In section 3, we give a more detailed explanation of our project idea and our approach. In section 4, we show our Blink+ performance result and discuss some critical findings we have found. In section 5, we discuss the future work.

Our project is open-sourced and can be found on https://github.com/UCBerkeley-Spring2022-CS267-project/blinkplus.

2 PRIOR WORK

2.1 Blink's Spanning Tree based collective operations.

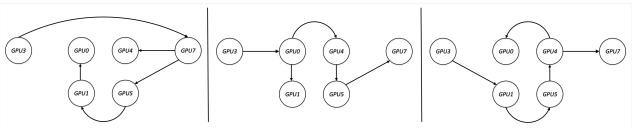


Figure 1: Spanning tree build by Blink [8]

Blink is a topology awared collective communication package where multiple spanning trees are packed together for communication. Blink shows that packing spanning tree will achieve theoretical lower bound for number of message send across node. A illustration of spanning tree build by Blink is shown in Figure 1. When building and packing spanning tree, Blink also assume each tree edge will fully utilize one NVLink bandwidth.

Our project follow Blink's [8] spanning tree approach and build spanning tree using NVLinks across tenant. We also follow Blink's assumption that each tree edge will fully utilize one NVLink bandwidth, this constrain how many spanning tree we can build across tenant. More detail explanation is given in the following section.

2.2 NCCL's Ring based collective operations

Like many collective communication package [1], [10], NCCL [7] 1.x support Ring based communication. The main idea of ring based collective operation is data pipeline. As illustrate in Figure 2, user input data are divided into multiple large chunk. Inside each large chunk, data is further divide into smaller chunk. Smaller data chunk are pipelined through the ring.

Our project do not utilize ring based collective ideas, and thus will not discuss it in further detail here.

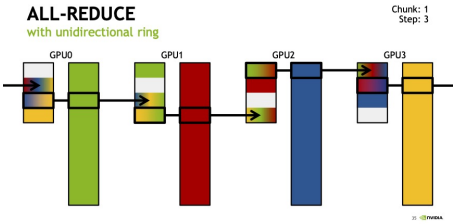


Figure 2: Ring based all-reduce using inside NCCL. [7]

2.3 NCCL's Tree based collective operations

Starting from version 2.x, NCCL also support the two tree algorithm collective operation based on [6]. User can explicitly enable the two tree algorithm by setting the environment variable `NCCL_ALGO=Tree`. In the original paper, each CPU or GPU node is considered as a tree node inside the two tree. The two tree structure ensures that leaf node inside one tree will always be the intermediate node in the other, this is illustrated in Figure 3. In the NCCL's implementation, all GPU node within a cluster (i.e. GPU 0-7 inside one DGX-1 box) are considered as a tree node in the two tree. For GPU nodes inside a cluster, they form a chain (not necessarily a ring) where the beginning and end of the chain connect to the other cluster (i.e. another DGX-1 box), which represent either parent or child tree node inside the two tree. Commonly the beginning and end of the chain are chosen so that those two GPU node share the same network card. User can also specify how the chain will be formed by providing a `graph.xml` file and set the environment variable `NCCL_GRAPH_FILE` point to that `graph.xml` file.

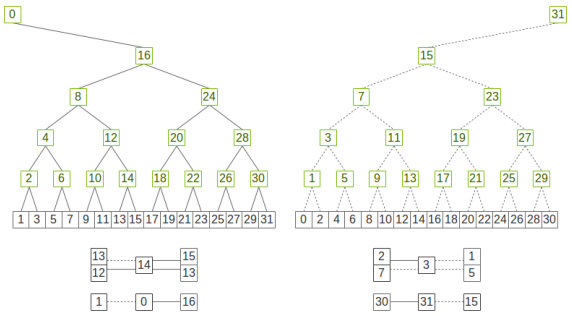


Figure 3: Two tree algorithm used inside NCCL[7]

Our project utilize the fact that NCCL's tree algorithm will use a chain inside the GPU cluster (i.e. inside one DGX-1) and user can specify how the chain will be build. More detail explanation is given in the following sections.

3 OUR APPROACH

3.1 Design Choice

In this project, we mainly focus on intra-node collectives operation for 2 GPU subset on 4 GPUs on a DGX-1 NVLink machine (e.g. User use GPU 0-1 and we build across tenant side channel using NVLinks from GPU 0 to 3). We make this design choice mainly due to time constrain of this project. The topology of GPU 0 to 3 on DGX-1 is shown on 4.

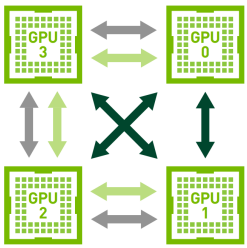


Figure 4: Topology of the first four GPUs in a NVIDIA DGX-1 cluster, connected by a interwoven net of NVLinks.

Similar to Blink, we build multiple spanning trees for the across tenant channel, partition data into chunks, and pipeline the chunked data on side channel spanning tree. Unlike Blink which only build spanning trees inside tenants, we utilize the idle link between tenants to build additional spanning trees involving non user-tenant GPUs ("side channels") to increase the collective communication bandwidth.

We implement the Blink+ idea by extending NCCL. Specifically, we build a wrapper around the NCCL public API, setting some NCCL-related variables, and call NCCL to build our side channel spanning tree and run collective operations. We choose to extend NCCL instead of building the package from scratch for the following reasons. First, after taking a deep insight into the source code of NCCL, we find it integrates sophisticated optimization on data transmission and pipelines, which is hard to surpass if we choose to build our package from scratch. Second, using NCCL-based implementation can potentially make our work more compatible with other software that uses NCCL like PyTorch [4] distributed training. Lastly, we have run a few experiments with NCCL and found that we can build side-channel spanning trees alongside the original user group and utilize NCCL's advanced implementation and optimization for development.

3.2 Implementation Detail

A general overview of our approach is shown in Algorithm 1. Figure 5 shows the Blink+ spanning tree given user group GPU 0-1. The filled line with black interconnection indicates GPU 0 and 1 in the user tenant. Each color line represents a non-user-tenant spanning tree that Blink+ builds to help increase the bandwidth. In the 2 GPU user tenant case, all the non-user-tenant spanning trees is equivalent to a chain structure (e.g., the red line in Figure 5a equivalent to a chain starting from 0, going through 3, and end at 1). As mentioned in Section 2.3, when setting `NCCL_ALGO=Tree` and provide a customized `graph.xml` file, we're able use NCCL to create chain within one DGX-1. We utilize this property of NCCL and build our non-user-tenant spanning tree by setting `NCCL_ALGO=Tree` and providing customized `graph.xml` files for each spanning tree we build. For example, in the case of user tenant GPU 0 and 1 (Figure 5a, there are 4 chains in total:

(1) GPU 0 to GPU 1 (black), created by NCCL
(2) GPU 0 to GPU 3 to GPU 1 (red), created by Blink+ through providing customized `graph.xml`
(3) GPU 0 to GPU 2 to GPU 1 (green), created by Blink+ through providing customized `graph.xml`
(4) GPU 0 to GPU 3 to GPU 2 to GPU 1 (blue), created by Blink+ through providing customized `graph.xml`

Recognize the extra-tenant GPUs;
for *subset* **in** *extra-tenant GPUs* **do**
  generate_tree(in-tenent GPUs + subset)
end
Decide on which trees to use, considering the link capacities;
Partition and assign the data to trees;
Initialize temporary buffers on extra-tenant GPUs;
for *tree* **in** *trees to use* **do**
  call NCCL collective;
end

Algorithm 1:

One thing to notice is that DGX-1's NVLink connection is a non-symmetric. As shown in Figure 4, GPU 0 to one are connected through one NVLink, while GPU 0 to 3 are connected through two NVLink. Blink+ is topology aware and thus would create different number of across tenant spanning tree given different user group. Sometimes the Blink+ spanning tree failed to utilize all the NVLink in topology. As shown in Figure 5a, one NVLink between GPU 3 and GPU 2 remain idle and there's no way to utilize it.

To build Blink+ by extending NCCL, we also need to allocate data buffers on non-user-tenant GPUs. For example, when the the user tenant involved GPU 0 and 1, we would allocate an data buffer on

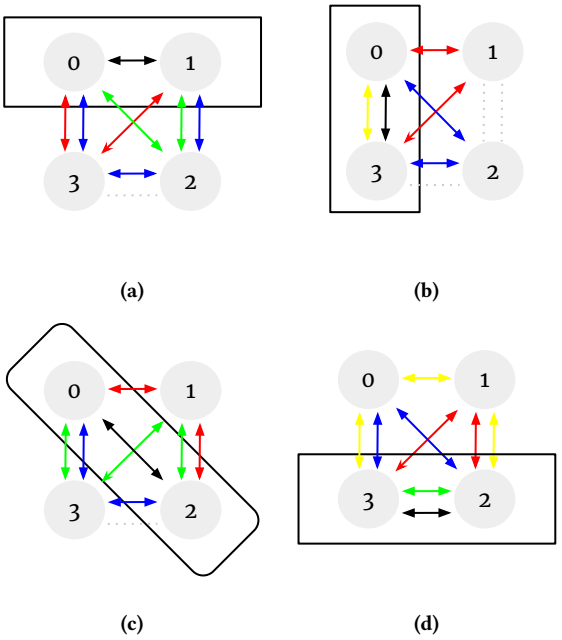


Figure 5: Spanning tree for collective communication. The two GPUs in the filled rectangles are the user tenant GPUs. Each color line signifies a spanning tree that Blink+ build to increase bandwidth. The gray dotted links represent the NVLinks that have to remain idle.

GPU 2 and 3 so that the data can be transferred through spanning tree of "0 -> 2 -> 1", "0 -> 3 -> 1" and "0 -> 2 -> 3 -> 1". The ideal option is to build a circular buffer [9] to save memory footprint. For ease of implementation, we allocate a large 1 GB buffer for each non-user-tenant GPU for across the tenant spanning tree.

In the `broadcast` case, a receive buffer is allocated on each non-user-tenant GPU. The initial value of these buffers is not relevant, as the data from the root device will overwrite them. The send buffer is also not relevant for `broadcast` operation. In `allreduce`, the data in send buffers of non-user-tenant GPUs are involved in the calculation. For example, in the user tenant GPU 0-1 case, part of the data would run `allreduce` on GPU 0 and 1 (original user tenant), and part of the data would run `allreduce` on GPU 0 3 and 1 (Blink+ spanning tree). We would use an all-zero buffer for GPU 3 if the all reduce operation is `sum` and an all one buffer for GPU 3 if the all reduce operation is `product`. The content of the received buffer is not relevant since it will always be overwritten.

4 RESULT

In this section, we first provide the experimental setup, then provide results to show the performance of our approach Blink+ in improving the bandwidth of collective communication.

4.1 Experimental Setup

For experimental setup, we use the GPU platform NVIDIA DGX-1 with NVLink Generation 2 and 4 V100 GPUs. We consider two operations `broadcast` and `allreduce`, two tenant user cases GPU 0-1 and GPU 0-3. We set our baseline to be NCCL's bandwidth and compare it with the bandwidth achieved by adding Blink+ on top. The `broadcast` operation means every node inside the group will receive x Megabyte (MB) of data copy from the defined root node. The `allreduce` operation means every node inside the group will have the reduced value from all node's input data of x Megabyte (MB). We test the bandwidth with multipling x values spanning from 1 MB to 1024 MB with a factor of 2.

We follow the protocol of the official `nccl-test` to do the bandwidth measurement and follow the definition of algorithm bandwidth (BW) inside `nccl-test`. We first do a warm-up by running the kernel for 5 iterations, and during timing, we repeat the operations for 20 iterations and get the final estimate T by diving the elapsed time by the number of iterations. The bandwidth is calculated by

$$\mathbf{BW} = \frac{x}{T}$$

4.2 Theoretical Performance Analysis

We analyze the performance of the two user cases (GPU 0-1 and GPU 0-3) separately because the topology of the two cases is different and Blink+ is topology-aware. As shown in Figure 4, GPU 0-1 are connected with one NVLink and GPU 0-3 are connected with two NVLink. NCCL is topology-aware and will build one channel for GPU 0-1 and two-channel for GPU 0-3. The bandwidth of one NVLink is theoretically about 20 GB/s. Thus, the baseline bandwidth using NCCL for GPU 0-1 is 20 GB/s and for GPU 0-3 is 40 GB/s.

Our approach is topology-aware and the optimal number of side channels is different for the two cases. For GPU 0-1 case, Blink+ will build three side channels to fully utilize DGX-1 topology. As shown in Figure 5a, the three side channels are marked with colors red, green, and blue. Each side channel would increase the total bandwidth by 20 GB/s and thus increase the final bandwidth to 80 GB/s. This means the theoretical maximum bandwidth is increased by 300 %. For GPU 0-3 case, Blink+ will build two side channel to fully utilize DGX-1 topology (Figure 5b red and blue channel). This will increase the final theoretical bandwidth to 80 GB/s and achieve a 100 % theoretical bandwidth increase.

4.3 Analysis of Suboptimal Actual performance

We show the `broadcast` results of the two experiments in Figure 6c (GPU 0-1) and Figure 6d (GPU 0-3). Contrary to our expectation, our experiment results show that our approach fails to reach the theoretical maximum bandwidth of 80 GB/s and can only increase the actual bandwidth to 60 GB/s (200% improvement for GPU 0-1 and 50% for GPU 0-3). This suboptimal performance appears across user GPU groups, and we found that the maximum bandwidth we can achieve is around 60 GB/s.

We dive deeper into the reasons behind this suboptimal performance by visualizing the kernel execution time with NVIDIA Visual Profiler [3]. We profile the `broadcast` operation runtime of GPU 0-1 when adding 3 side channels (a total of 4 channels including the original GPU 0-1 one channel). This is the case where we failed to achieve theoretical maximum bandwidth of 80 GB/s. We also profile the `broadcast` operation runtime of GPU 0-1 when adding 2 side

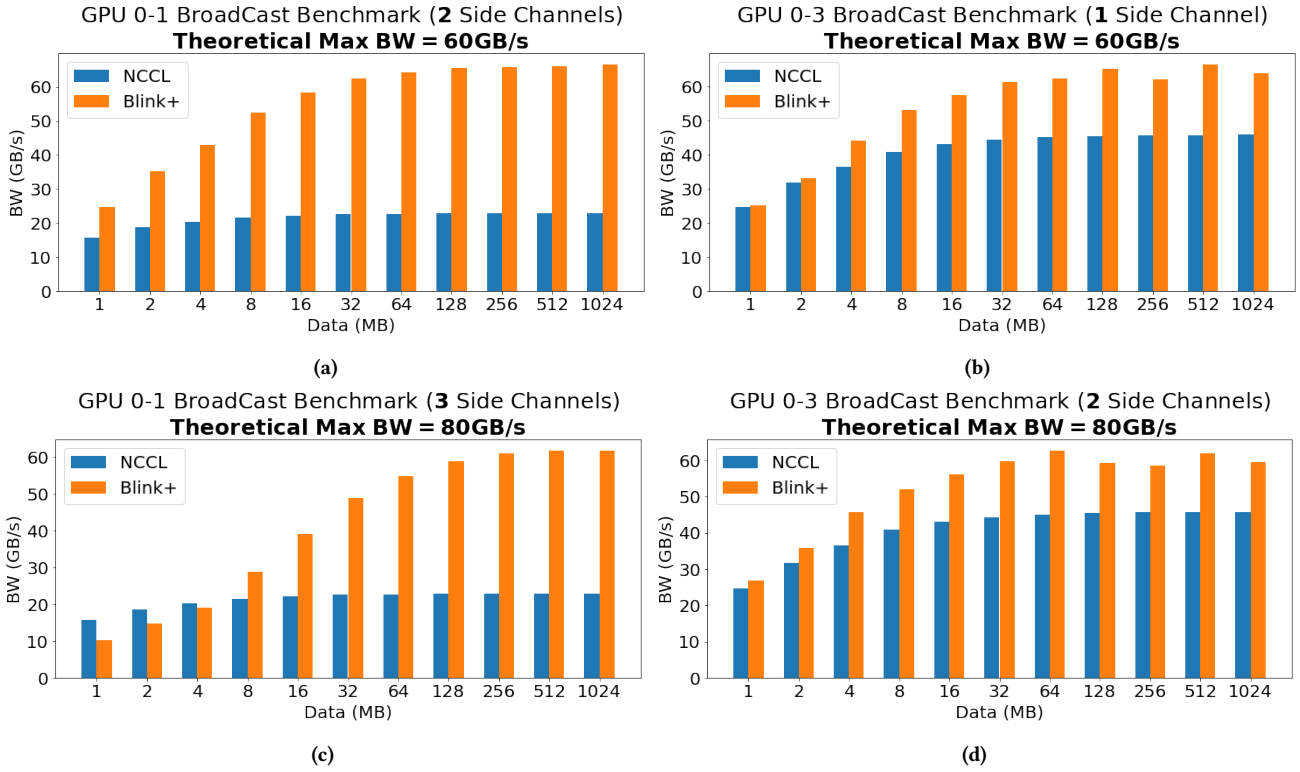


Figure 6: Comparing broadcast performance of Blink+ (ours) with NCCL (baseline) for two tenant cases GPU 0-1 (a) (c) and GPU 0-3 (b) (d).

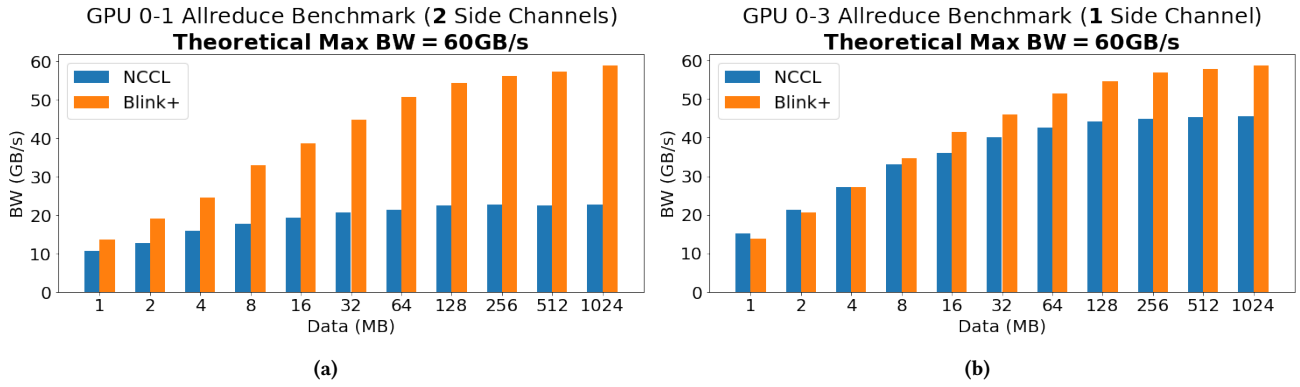
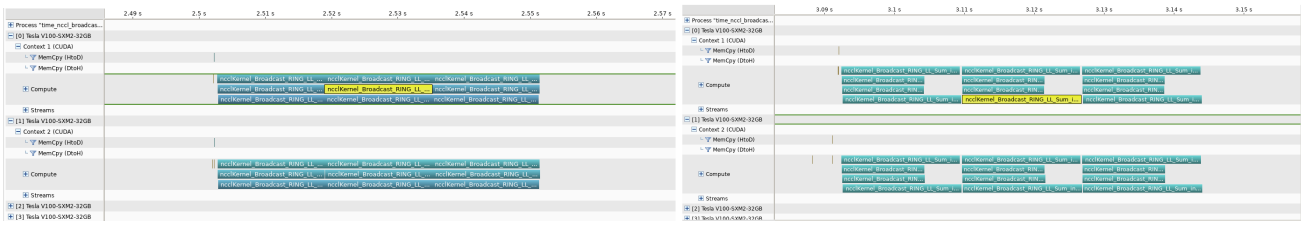


Figure 7: Comparing allreduce performance of Blink+ (ours) with NCCL (baseline) for two tenant cases GPU 0-1 (a) and GPU 0-3 (b).

channels (a total of 3 channels including the original GPU 0-1 one channel). The NVVP result is shown in Figure 8b for adding 3 side-channel and Figure 8a for adding 2 side channel. Each horizontal execution stream represents one channel. The `broadcast` kernel is executed 3 times, represented by three segments on one horizontal stream. We can tell from the profiling result that when a total of 4 channels (8b) is used (1 original channel and 3 additional side channels), the same kernel with same amount of work takes longer time to execute on channel 1 and channel 4. This causes the actual running time becomes longer and thus decreases the actual bandwidth. When a total of 3 channels is used (8a), the kernel in each channel takes roughly the same time. The reason behind the uneven kernel execution time is related to `CUDA run_time` [2] scheduling, and we do not have control over how kernels are scheduled.

This finding shows that when using 4 channels in total, either 1 original channel plus 3 side channels in GPU 0-1 or 2 original



(a) (b)

Figure 8: Visualization of broadcast kernel execution of GPU 0-1 using NVIDIA Visual Profiler. (a) Adding two side channels (total three channels). (b) Adding three side channels (total four channels).

channels plus 2 side channels in GPU 0-3, we are unable to reach the theoretical maximum bandwidth improvement due to CUDA runtime scheduling. The actual maximum bandwidth we can achieve is around 60 GB/s, which is equivalent of using 3 channels in total.

4.4 Preliminary Best Setting and Results

Based on the analysis above, we determine that the best setting for us to reach the optimal performance is to use three channels. This means adding two channels for GPU 0-1 (originally with one channel) and adding one channel for GPU 0-3 (originally with two channels). The results of the `broadcast` operation under the empirical best setting are shown in Figure 6a and 6b. The bandwidth improvement is 200% for GPU 0-1 and 50% for GPU 0-3, which achieve the theoretical maximum bandwidth of 60 GB/s with three channels. The results of `allreduce` operation are shown in Figure 7. Our approach improves the bandwidth of `allreduce` in the same percentage as the `broadcast` and reaches the theoretical maximum bandwidth of using three channels in total.

5 FUTURE WORK

In this semester, we mainly focus on the case of 2 GPU subset in 4 GPU case. We choose to tackle the 2 GPU subset case in this semester because it allow us to utilize NCCL's chain based collective operations and achieve higher theoretical bandwidth improvement. NVIDIA DGX-1 have 8 inter-connected GPUs. A further step include generalize the Blink+'s utilize bandwidth across tenant idea to all GPU subsets case in NVIDIA DGX-1's 8 GPU (e.g. using NVLink between GPU 0,1,2,3 and GPU 4,5,6,7 for user tenant group of GPU 0,1,2,3). We also maintain a large 1 GB buffer inside Blink+ for sending and receiving data on non-user tenant GPUs. A circular buffer can be used to reduce the memory overhead for non-user tenant GPUs.

REFERENCES

[1] baidu-research. 2017. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce.

[2] Nvidia. 2022. cuda-runtime. https://docs.nvidia.com/cuda/cuda-runtime-api/.

[3] Nvidia. 2022. nvidia-visual-profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035.

[5] Ganesan Ponnuswami, Sriram Kailasam, and Dileep Aroor Dinesh. 2022. Evaluating Data-Parallel Distributed Training Strategies. In *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*. 759–763. https://doi.org/10.1109/COMSNETS53615.2022.9668349

[6] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.* 35, 12 (dec 2009), 581–594. https://doi.org/10.1016/j.parco.2009.09.001

[7] Sylvain Jeaugey. 2021. NCCL. https://github.com/NVIDIA/nccl.

[8] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.

[9] Wikipedia contributors. 2022. Circular buffer — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Circular_buffer&oldid=1084776648. [Online; accessed 8-May-2022].

[10] Bingjing Zhang, Yang Ruan, and Judy Qiu. 2015. Harp: Collective Communication on Hadoop. In *2015 IEEE International Conference on Cloud Engineering*. 228–233. https://doi.org/10.1109/IC2E.2015.35