Comprehensive Deadlock Prevention for GPU Collective Communication

Lichen Pan
School of Computer Science, Peking University
plch368@pku.edu.cn

Juncheng Liu
OneFlow Research
liujuncheng@oneflow.org

Yongquan Fu
National Key Laboratory of Parallel and Distributed Computing, College of Computer Science and Technology, National University of Defense Technology
yongquanf@nudt.edu.cn

Jinhui Yuan
OneFlow Research
yuanjinhui@oneflow.org

Rongkai Zhang
School of Computer Science, Peking University
rkzhang@stu.pku.edu.cn

Pengze Li
School of Computer Science, Peking University
lipengze@pku.edu.cn

Zhen Xiao
School of Computer Science, Peking University
xiaozhen@pku.edu.cn

Abstract

Distributed deep neural network training necessitates efficient GPU collective communications, which are inherently susceptible to deadlocks. GPU collective deadlocks arise easily in distributed deep learning applications when multiple collectives circularly wait for each other. GPU collective deadlocks pose a significant challenge to the correct functioning and efficiency of distributed deep learning, and no general effective solutions are currently available. Only in specific scenarios, ad-hoc methods, making an application invoke collectives in a consistent order across GPUs, can be used to prevent circular collective dependency and deadlocks.

This paper presents *DFCCL*, a novel GPU collective communication library that provides a comprehensive approach for GPU collective deadlock prevention while maintaining high performance. *DFCCL* achieves preemption for GPU collectives at the bottom library level, effectively preventing deadlocks even if applications cause circular collective dependency. *DFCCL* ensures high performance with its execution and scheduling methods for collectives. Experiments show that *DFCCL* effectively prevents GPU collective deadlocks in various situations. Moreover, extensive evaluations demonstrate that *DFCCL* delivers performance comparable to or superior to NCCL, the state-of-the-art collective communication library highly optimized for NVIDIA GPUs.

CCS Concepts: • **Computer systems organization** → *Distributed architectures*; *Reliability*; • **Computing methodologies** → *Machine learning*.

Keywords: Collective Communication, GPU, Deadlock Prevention, Preemption

ACM Reference Format:
Lichen Pan, Juncheng Liu, Yongquan Fu, Jinhui Yuan, Rongkai Zhang, Pengze Li, and Zhen Xiao. 2025. Comprehensive Deadlock Prevention for GPU Collective Communication. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717466

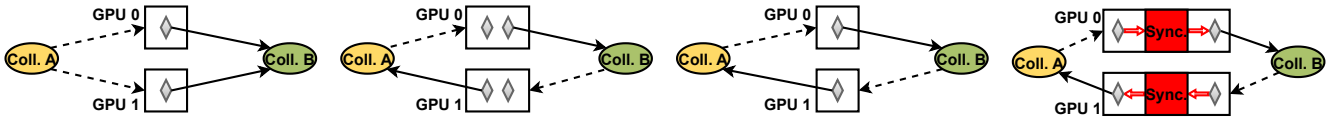
EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717466

1 Introduction

Recent years have witnessed the parameter count of deep neural network (DNN) models grow faster than the memory capacity and computational power of a single GPU [3, 21, 57]. This entails distributed DNN training, which includes various techniques such as data parallelism (DP) [35, 56], tensor parallelism (TP) [7, 59, 65], pipeline parallelism (PP) [25, 40, 41], and hybrid parallelism [7, 41, 55]. GPU collective communications are used to synchronize DNN status and play a critical role in distributed DNN training.

Widely used GPU collectives are vulnerable to deadlocks, because they work in a resource-holding, busy-waiting way, and preemption is ill-supported on GPUs. GPU collective



(a) **Legal**: consistent order.

(b) **Legal**: disorder with sufficient resources.

(c) **Deadlock**: disorder with single queue or resource-depletion.

(d) **Deadlock**: disorder with GPU synchronization, despite sufficient resources.

Figure 1. Legal Situations and Basic Deadlock Situations. In the figures, both collective A and collective B execute on GPU 0 and GPU 1. The diamonds within each GPU represent the resource "units" required to execute a collective on that GPU, assuming collective A and B require the same amount of resources. Solid arrows indicate that a resource unit has been allocated to a collective, which means the collective is *executing* on that GPU. Dashed arrows represent a collective applying for a resource unit from a GPU, signifying that the collective is *invoked* on that GPU but is not executing. Outlined red arrows depict the dependencies of idle resources on allocated resources introduced by GPU synchronization.

deadlocks occur when an application causes circular collective dependency. We summarize the *basic deadlock situations* of GPU collectives (Fig. 1) at the bottom library level of the distributed deep learning stack. The *disordered invocation of collectives* across different GPUs emerges as a necessary condition for circular collective dependency and deadlocks, while *GPU synchronization* (see Sec. 2.3) exacerbates the circular collective dependency.

We conduct simulation experiments to quantitatively analyze the susceptibility of GPU collectives to deadlocks. The simulation experiments (Table 1) show that low probabilities of collective disorder and GPU synchronization (both 0.004%) can cause high deadlock risks (6.94%), and the deadlock ratio is more sensitive to GPU synchronization than to disordered collective invocation.

GPU collective deadlocks pose significant threats to the correct functioning, training efficiency, and hardware utilization of distributed deep learning. GPU collective deadlocks can cause GPUs to show 100% utilization with no progress being made, leading to wasted resources [12–14, 16]. The deadlocks are challenging to identify and resolve with little error messages or logs clearly indicating the problem [30, 39].

Existing methods fail to deal comprehensively and effectively with GPU collective deadlocks. Current methods to prevent circular collective dependency and deadlocks are stopgap case-by-case solutions that make an application invoke collectives in a consistent order across GPUs, with GPU synchronization unmanaged. At present, manual hardcoding, which is expensive to develop and verify, and tightly coupled with applications, remains the sole practical approach when integrating PP with other parallel techniques. In more complex and irregular distributed DNN training scenarios [6, 11], applying the labor-intensive yet ad hoc manual collective orchestration method is quite challenging, and uncontrolled GPU synchronization further diminishes its effectiveness.

This paper presents *DFCCL* (Deadlock Free Collective Communication Library), which is, to the best of our knowledge, the first GPU collective communication library that provides a comprehensive approach for GPU collective deadlock prevention and maintains high performance.

DFCCL offers a general and effective method to prevent deadlocks in various scenarios via *preemption*, fundamentally breaking the inherent susceptibility of GPU collectives to deadlocks. *DFCCL* executes collectives in a *two-phase blocking* manner [20, 49] in the *daemon kernel*, preempting collectives deemed to be *stuck* via context switch. With preemptive support at the bottom library level, circular collective dependency from applications no longer causes deadlocks.

DFCCL maintains high performance with three key designs: ❶ *DFCCL* employs the on-GPU control logic and the busy-waiting [46] execution mode before preempting a collective to ensure high throughput. ❷ *DFCCL* makes each GPU independently perform collective preemption in a *decentralized dynamic* manner to avoid coordination overheads among GPUs. ❸ *DFCCL*'s adaptive scheduling scheme, which supports user-specified priority and achieves decentralized dynamic gang-scheduling for collectives, ensures efficient scheduling and execution of collectives.

We implement *DFCCL* based on NVIDIA GPUs. *DFCCL* can be seamlessly integrated with existing distributed deep learning frameworks [1, 51, 66] by appropriately substituting their NCCL API calls with *DFCCL* API calls.

Experimental results show that *DFCCL* effectively prevents GPU collective deadlocks. We compare collective bandwidth and latency between *DFCCL* and NCCL [46], the state-of-the-art (SOTA) collective communication library highly optimized for NVIDIA GPUs. We conduct experiments to compare the DNN training performance using collectives from *DFCCL* versus NCCL across various distributed training techniques and frameworks. Evaluation results show that *DFCCL* delivers performance comparable to NCCL, which

requires various scenario-dedicated CPU orchestration, and achieves performance gains under certain circumstances.

This paper makes the following contributions:

- We quantify the analysis of deadlocks and the influencing factors through simulation experiments.
- We identify the preemption chances for GPU collectives and introduce *DFCCL*, a novel GPU collective communication library that offers a comprehensive approach to preventing GPU collective deadlocks with performance guarantees.
- Experimental results show that *DFCCL* effectively prevents GPU collective deadlocks and achieves performance comparable to or superior to NCCL.

2 Background and Motivations

2.1 NCCL

NCCL provides GPU collectives efficiently utilizing inter-GPU bandwidth, and is commonly used in distributed DNN training. Experiments show that the throughput of NCCL all-reduce surpasses that of CUDA-aware MPI [31, 61] when the buffer size exceeds 32 KB. The increase in throughput reaches over 6.7× at most.

NCCL boosts throughput by shifting from the CPU-based control plane to the on-GPU, busy-waiting control logic, which, however, makes NCCL collectives deadlock-prone.

2.2 Community Discussions

The documentation of fundamental distributed DNN training tools merely alerts developers to the risk of NCCL deadlocks without providing practical solutions [2, 15, 45, 47, 48].

We investigate issues reporting NCCL-related deadlocks in Pytorch, DeepSpeed, and TensorFlow repositories. When a deadlock occurs, typical symptoms include the program hanging, GPU utilization stuck at 100%, NCCL logs failing to provide valuable information. Despite extensive discussions, no final solutions are proposed in many issues, while in others, only ad-hoc workarounds are offered, such as introducing fences or turning off certain hardware features.

The deadlocks reported in some issues conform to the GPU synchronization-related deadlock case (Sec. 2.3), and the improvised methods avoid the mutual interference between GPU synchronization and collectives. Issue #31095 reported in PyTorch [13] reports a scenario where after the invocation of GPU collectives, a process gets stuck at the code line responsible for page-locked host memory allocation. When a sleep interval is added between the GPU collectives and the memory allocation, this problem disappears. Issues reporting NCCL deadlocks from various repositories [12, 14, 16] suggest disabling IOMMU (Input-Output Memory Management Unit), which triggers CPU-initiated GPU memory operations, leading to implicit GPU synchronization.

2.3 Analysis of GPU Collective Deadlocks

NCCL collectives are vulnerable to deadlocks since they inherently satisfy three out of the four individually necessary and jointly sufficient conditions for a deadlock (except *circular waiting*): ❶ ***Mutual exclusion***: GPU resources, e.g., streaming multiprocessors and shared memory [43], occupied by one collective cannot be simultaneously used by others. ❷ ***Hold and wait***: In NCCL, the parts of a collective on different GPUs busy-wait until all peers are ready while holding allocated resources. ❸ ***No preemption***: There is no practical official preemption support for GPUs, and the GPU-preemption techniques in literature are not suitable for collectives (see Sec. 7). Therefore, once an application causes circular collective dependency, a deadlock occurs.

At the bottom library level, Fig. 1 shows the legal situations and the **basic deadlock situations** of GPU collectives.

When an application invokes collectives in a consistent order on each GPU, e.g., invoking collective B before A on both GPU 0 and 1 (Fig. 1(a)), these collectives can execute normally. CUDA stream [44] enables the parallel execution of multiple collective kernels when resources are sufficient. When an application invokes collectives in different orders on each participating GPU, if these collectives are issued to different streams, and the kernels in all these streams can be scheduled for execution due to sufficient resources (Fig. 1(b)), these collectives can execute normally.

Fig. 1(c) and 1(d) summarize three basic GPU collective deadlock situations with circular collective dependency at bottom library level. ❶ **Single Queue**: For the single queue programming model where collectives are issued in a single stream on each GPU, the disordered invocation of collectives on different GPUs leads to circular collective dependency and a deadlock (Fig. 1(c)). ❷ **Resource Depletion**: When collectives are issued to different streams, but resources are insufficient, the disordered invocation of collectives on different GPUs leads to the deadlock situation similar to Single Queue (Fig. 1(c)). ❸ **GPU Synchronization Related**: GPU synchronization is categorized into explicit and implicit types [43]. Explicit synchronization involves *cudaDeviceSynchronize()* calls, while implicit synchronization includes GPU default stream commands, page-locked host memory allocation, and CPU-initiated GPU memory operations. GPU synchronization operations block and suspend a GPU until all kernels in all streams of the GPU complete. Therefore, GPU synchronization introduces the dependency where idle resources cannot be allocated to collectives invoked after the GPU synchronization, until all allocated resources are released by prior collectives after their completion. As shown in Fig. 1(d), when two GPUs invoke two collectives in opposite orders, and both issue GPU synchronization after invoking one, the disordered collective invocation and the resource dependency introduced by GPU synchronization together lead to circular collective dependency and a deadlock.

Table 1. Configurations and Deadlock Ratios in Simulation Based Analysis.

	3D Grouping Policy							Free Grouping Policy					
	TP, DP, PP Group Size	#Group	#GPU	#Collective / TP, DP Group	Disorder Prob.	Sync. Prob.	**Deadlock Ratio**	#Group	#GPU	#Collective / Group	Disorder Prob.	Sync. Prob.	**Deadlock Ratio**
Single-Queue Model	4, 4, 4	32	64	400, 1200	1e-7	-	**1.10%**	1	8	161	1e-5	-	**1.21%**
					1e-6	-	**9.97%**	32	64	400 or 1200	1e-6	-	**0.98%**
	8, 6, 64	896	3072	400, 1200	1e-9	-	**0.47%**				1e-5	-	**9.45%**
					1e-8	-	**3.59%**	32	128	400 or 1200	1e-6	-	**1.72%**
Sync. Model	4, 4, 4	32	64	400, 1200	2e-3	4e-3	**0.68%**	32	64	400 or 1200	4e-6	4e-5	**0.81%**
					4e-3	4e-3	**1.38%**				4e-5	4e-5	**1.16%**
					4e-3	2e-3	**0.32%**				4e-5	8e-5	**6.56%**
				800, 2400	4e-3	4e-3	**2.56%**			800 or 2400	4e-5	4e-5	**6.94%**
	8, 6, 64	896	3072	400, 1200	8e-4	8e-4	**1.56%**	32	128	400 or 1200	4e-5	4e-5	**2.34%**

Disordered collective invocation across GPUs is a necessary condition for circular collective dependency and deadlocks, with GPU synchronization exacerbating the risk of circular waiting among collectives. The root cause of the disordered collective invocation and the issuance of GPU synchronization originates with the application. In applications where collectives lack data dependency, these collectives can be invoked in varying orders across GPUs. GPU synchronization, independent of these collectives, can also be issued whenever needed on corresponding GPUs.

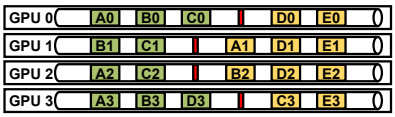


Figure 2. Example Collective States in the Synchronization Model. A, B, C, D, and E are five collectives. A*i* represents the part of collective A on GPU *i*. The left green box indicates a collective is *executing* on a GPU, and each right yellow box indicates an *invoked* collective. The middle red bar represents the synchronization that suspends a GPU.

2.4 Simulation Based Analysis

2.4.1 Deadlock Simulator We develop a simulator to quantitatively analyze how disordered collective invocation and GPU synchronization affect GPU collective deadlocks.

Our simulation is driven by real-world distributed DNN training practices and profiling [41, 59, 66]. GPUs are organized into different *groups*. Each group has a separate collective list for its GPUs to *invoke* and *execute*. A single GPU can belong to multiple groups, which means that the collectives a GPU will invoke and execute are the union of the collectives from all the groups the GPU belongs to. A collective has three states on each GPU: *invoked*, *executing*, and *successful*. A collective becomes successful when it reaches the executing state on all GPUs in its corresponding group. The condition for a collective's transition from invoked to executing state is related to the **deadlock decision model**.

We study two deadlock decision models in the simulator based on the basic deadlock situations discussed in Sec. 2.3:
• **Single-queue model.** A collective on a GPU transitions to the executing state if there is no executing or invoked collectives before it. Each GPU is restricted to having only one executing collective at a time.
• **Synchronization model.** A GPU may randomly initiates synchronization operations that suspends it. A collective on a GPU transitions to the executing state if it is invoked before the GPU is suspended. A GPU ends suspension when all executing collectives before the synchronization transition to successful, or if there are no executing collectives before the synchronization. Each GPU can maintain an unlimited number of executing collectives. Given the varying total resources of different GPU models and the diverse resource needs of collectives, the simulator employs an idealized infinite resource assumption to simplify these complexities.

In both deadlock decision models, the simulator checks for cycles in the *dependency graph* to determine the presence of a deadlock, after each collective invocation and synchronization issuance. The *nodes* of the dependency graph are collective parts on GPUs. The graph includes two types of *directed dependency edges*: ❶ an executing collective on one GPU points to all its invoked counterparts on other GPUs, e.g., D3->D0, D3->D1, and D3->D2 in Fig. 2; ❷ an invoked collective on a GPU points to all executing collectives on the same GPU, e.g., D0->A0, D0->B0, and D0->C0 in Fig. 2. One of the cycles in the dependency graph corresponding to Fig. 2 is A0->A1->B1->B2->C2->C3->D3->D0->A0.

The simulation models key behaviors related to GPU collective deadlocks: (1) the *disorder probability* specifies the probability of disordered collective invocations by GPUs; (2) the *synchronization probability* determines the probability of issuing synchronization operations.

We study two typical **GPU grouping policies** for distributed DNN training:
• **3D grouping policy**. As shown in Fig. 3, GPUs are organized according to the 3D grouping scheme in 3D-hybrid parallel distributed DNN training [41, 59], where GPUs holding the same DNN model part in different TP groups form a DP group within each PP group. The configuration file

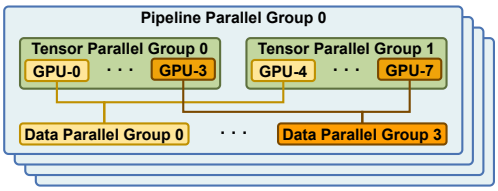


Figure 3. Example Grouping of GPUs in 4-way Tensor, 2-way Data, and 4-way Pipeline Hybrid Parallelism [41, 59].

specifies the sizes of the TP, DP, and PP groups, along with the number of collectives in the TP and DP groups.
• **Free grouping policy.** The configuration file directly specifies the total number of groups, as well as the GPU lists and the number of collectives of each group.

The simulator's input is synthesized event sequences for each GPU. These sequences, consisting of collective invocation events and GPU synchronization events, are generated based on disorder probability, synchronization probability, and GPU grouping. The simulator transitions the collective state according to currently submitted events and determines if deadlocks occur using the deadlock decision model.

2.4.2 Simulation Setup Table 1 summarizes the configurations and deadlock ratios of the simulation experiments.
• The (8, 6, 64)-3D grouping case is inspired by the training configuration of GPT-3 [41]. The (1, 8)-free grouping case simulates a data parallel scenario.
• In 3D grouping, each GPU invokes collectives from two groups. In contrast, in the (32, 64)-free grouping case, which mirrors the (4, 4, 4)-3D grouping in total groups and GPUs, GPUs variably receive collectives from one to five groups.
• In the (32, 64)-free grouping case, 28 groups have three GPUs each, and four groups have eight GPUs each. The (32, 128)-free grouping case increases each group by two GPUs.
• In the free grouping policy, "400 or 1200" means 50% of groups have 400 collectives and the other 50% have 1200 collectives. "800 or 2400" preserves this distribution.
• A *round* is defined as all collectives are successful or the simulation runs until a deadlock arises. For all the configurations, deadlock ratios are calculated from 32,000 rounds.

2.4.3 Result Analysis From Table 1, we can conclude:
❶ Extremely low disorder and synchronization probabilities can lead to prohibitively high deadlock risks. In the single-queue model, the deadlock ratio is up to six orders of magnitude higher than the disorder probability (0.49% vs. 1e-9). In the synchronization model, the deadlock ratio is up to three orders of magnitude higher than the disorder and synchronization probability (6.94% vs. 4e-5).
❷ The deadlock ratio is positively correlated with the disorder probability and the synchronization probability.
❸ The synchronization model is more sensitive to the synchronization probability than to the disorder probability. For example, in the (32, 64)-free grouping case, while increasing the disorder probability tenfold (from 4e-6 to 4e-5) raises the deadlock ratio by 42%, doubling the synchronization probability (from 4e-5 to 8e-5) dramatically increases the deadlock ratio by 468%. This is because GPU synchronization introduces the dependency of idle resources on allocated resources, which facilitates the emergence of circular waiting among disordered collectives, and results in deadlocks even with ample resources (Fig.1(d) vs. Fig.1(b)).
❹ The deadlock ratio correlates positively with the total number of GPUs and the total number of planned collectives.
❺ In the synchronization model, the deadlock ratio is positively correlated with the group overlapping degree, i.e., the number of groups a GPU belongs to, which indicates the complexity of distributed DNN training scenarios. For example, the (32, 64)-free case employs disorder and synchronization probabilities two orders of magnitude lower than the (4, 4, 4)-3D case (4e-5 vs. 4e-3), yet their reported deadlock ratios are similar (1.16% vs. 1.38%).

Sec. 6.1 demonstrates that in a real-world environment, when both disorder and synchronization probabilities are engineered to reach 100%, NCCL exhibits a 100% deadlock ratio, whereas *DFCCL* does not encounter any deadlocks.

2.5 Existing Methods to Deal With Deadlocks

Existing methods prevent circular collective dependency by ensuring that collectives are invoked in a consistent order across GPUs, without managing GPU synchronization. Most approaches achieve this through additional CPU orchestration tightly coupled with specific parallel training techniques. Manual hardcoding is currently the only viable solution when combining PP with other parallel techniques.

Different CPU coordination strategies are employed for data parallelism. ❶ Horovod [56] presents the dynamic centralized coordinating approach. The Horovod central coordinator gathers collectives' readiness from each GPU during runtime and broadcasts a list of collectives ready on all GPUs, allowing GPUs to start the all-reduces in the list order. ❷ BytePS [29] requires centralized coordination prior to invoking collectives among intra-node GPUs. ❸ KungFu [38] determines the predominant GPU collective calling order in the initial training step via gather and broadcast operations. Subsequently, decentralized schedulers enforce this order across all GPUs. ❹ OneFlow [66] introduces a static-sorting based scheduling approach. Its compiler automatically constructs task graphs for all GPUs, sorting collectives based on each graph's topological order. During runtime, GPUs initiate collectives following these pre-sorted sequences.

Horovod, BytePS, and KungFu are incapable of orchestrating all collectives in 3D-hybrid parallelism. ❶ Megatron-LM [41, 59] introduces manual hardcoding for hybrid parallelism by manually and meticulously arranging each GPU's collectives related to different groups (Fig. 3). ❷ OneFlow

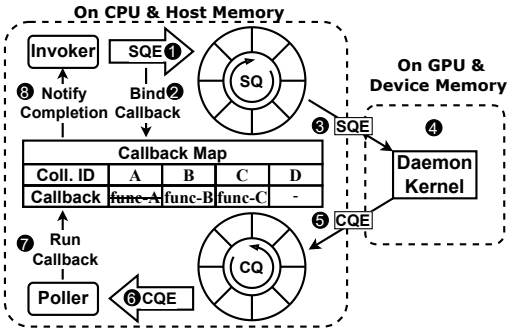


Figure 4. Overview of *DFCCL*. A, B, C, and D denote four registered collectives. A is complete; B and C are invoked; and D has not been invoked yet.

follows the manual hardcoding scheme when combining PP with other parallel techniques.

2.5.1 Limitations of Existing Methods Case-by-case collective orchestration at the application level is neither effective nor general in preventing GPU collective deadlocks.

Manual hardcoding is indeed an expensive ad hoc method. The existing manual orchestration implementation demands extensive development and verification by experienced engineers, and is closely tied to hybrid parallelism characteristics.

As distributed training patterns grow more complex and dynamic, it is becoming increasingly challenging for engineers to manually orchestrate collectives at the application level, ensuring that GPUs invoke them in the consistent order under all runtime circumstances. Pathways [6, 11] presents a heterogeneous and dynamic distributed training paradigm less symmetrical and more irregular than 3D-hybrid parallelism[1]. This paradigm is conceptually similar to the (32, 64)-free grouping case in Sec. 2.4.2. Table 1 shows, in the (32, 64)-free grouping case, disorder and synchronization probabilities of no more than 0.004% yield a deadlock risk near 7%. Besides, Sec. 2.4.3 shows deadlocks are more sensitive to GPU synchronization than to disordered collective invocation. This implies that in dynamic, complex scenarios, as long as the disorder probability is not zero, laborious collective orchestration can be futile in preventing deadlocks due to uncontrolled synchronization fluctuations, e.g., more frequent GPU memory allocations.

3 Overview

DFCCL is a GPU collective communication library that prevents deadlock through preemption, and maintains high performance with efficient execution and scheduling.

3.1 *DFCCL* Components and Collective Life Cycle

DFCCL consists of GPU and CPU parts, as shown in Fig. 4. On each GPU, the *daemon kernel* (Sec. 4), *DFCCL*'s core component, handles execution, preemption, and scheduling for multiple collectives. *DFCCL*'s CPU component provides user-friendly APIs and manages asynchronous, non-blocking request submitting and completion notifying based on the *submission queue (SQ)* and *completion queue (CQ)*. Each GPU's daemon kernel corresponds to a separate SQ & CQ.

Fig. 4 identifies a collective's life cycle in *DFCCL* through numerical labels. ❶, ❷: The collective invoker inserts a *submission queue element (SQE)* into the SQ, and records a (collective ID, callback) pair in the callback map. ❸, ❹: The daemon kernel periodically checks the SQ, fetches and parses SQEs, and executes the requested collectives. ❺: The daemon kernel inserts a *completion queue entry (CQE)* for each completed collective into the CQ. ❻, ❼: The *poller* thread monitors the CQ. Once the poller finds a CQE, it executes the callback tied to the collective, which notifies the invoker of the collective's completion in the user-defined way.

Using callbacks for collectives' asynchronous invocation is a common programming practice [51, 66]. *DFCCL*'s design offers a flat learning curve.

3.2 APIs of *DFCCL*

Listing 1 provides a list of *DFCCL*'s APIs. ***dfcclInit*** initializes the rank context of a GPU. ***dfcclDestroy*** destroys the rank context and releases resources. ***dfcclRegister**** ("*" represents a specific collective) registers a collective and prepares corresponding data structures for a collective on the specified GPU. Every registered collective has a unique ID. ***dfcclRun**** invokes a registered collective based on the ID and records a user-defined callback corresponded to the collective.

In *DFCCL*, a collective is registered once with *dfcclRegister** and can then be invoked repeatedly as needed using *dfcclRun**. *DFCCL* also allows dynamic registration of new collectives during runtime.

The *communicator* manages the resources for inter-GPU data transfer during collective execution. *DFCCL* maintains a communicator pool transparent to users, automatically creating and allocating communicators for collectives.

[1]Pathways relies on the centralized scheduler and a closed-source dataflow system called PLAQUE to ensure that all participating devices invoke collectives in a consistent order. The specific implementation details are not publicly available. However, as described in its paper, Pathways also relies on additional CPU orchestration to prevent deadlocks in device collectives, making it fundamentally indistinguishable from existing approaches.

```
ret_t dfcclInit(rankCtx_t* rankCtx, int rank);
ret_t dfcclRegisterAllReduce(size_t count, type_t
    type, redOp_t op, int collId, devSet_t devSet,
     int priority, rankCtx_t rankCtx);
ret_t dfcclRunAllReduce(const void* sendbuff, void
    * recvbuff, int collId, func_t callback, void*
     callbackArgs, rankCtx_t rankCtx);
ret_t dfcclDestroy(rankCtx_t rankCtx);
```

Listing 1. APIs of *DFCCL*. Take all-reduce as an example.

DFCCL can be seamlessly integrated with existing distributed deep learning frameworks, e.g, PyTorch [51], OneFlow [66], TensorFlow [1], by appropriately substituting their calls to the NCCL APIs with calls to the *DFCCL* APIs.

3.3 Benefits of *DFCCL*

DFCCL's daemon kernel integrates preemptive scheduling on GPU, fundamentally breaking GPU collectives' inherent susceptibility to deadlocks. This eliminates the need for CPU-based coordination between user-facing interfaces and DFCCL's library-level APIs (e.g., *dfcclRunAllReduce*). Embedding scheduling logic in a daemon kernel rather than relying on CPU orchestration is novel and has the potential to be extrapolated to other CPU-GPU paradigms.

DFCCL achieves three key objectives simultaneously: 1) the APIs of the underlying collective communication library can be directly called concurrently and asynchronously; 2) guaranteeing high performance; and 3) maintaining independence from specific parallel training techniques, thus ensuring wide applicability. The combined realization of these three objectives represents an open research challenge that existing work has not addressed.

***DFCCL* vs. NCCL** The methodology difference between *DFCCL* and NCCL includes two main aspects.

❶ *DFCCL* manages the execution, preemption, and scheduling for an arbitrary number of collectives submitted dynamically in a single daemon kernel. In contrast, each NCCL kernel is dedicated to one or a few predetermined collectives, relying entirely on CUDA's underlying scheduling.

❷ *DFCCL* offers SQ, CQ, and callback management for asynchronous request submitting and completion notifying, unlike NCCL, which requires additional mechanisms, e.g., cudaEvent [43], to verify collective completion asynchronously.

4 Daemon Kernel

In this section, we first present the preemption chances for GPU collectives. Next, we present collective preemption and scheduling in the daemon kernel. Then, we present the daemon kernel's voluntary quitting and event-driven starting. Finally, we analyze its correctness and performance.

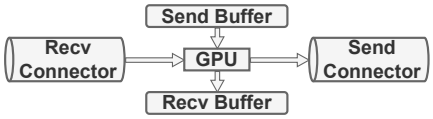


Figure 5. Buffers Used in GPU Collectives.

4.1 Preemption Chance of GPU Collectives

The preemption opportunity for common GPU collectives (all-reduce, all-gather, reduce-scatter, reduce, and broadcast) arises because they are all composed of a subset of the same group of *primitives* [17, 46]. In a collective, GPUs are organized into a specific logical topology, with each GPU assigned a primitive sequence based on its position within this topology. To facilitate processing, input data for a collective are divided into regular *chunks*. GPUs execute a collective by performing its primitive sequence a certain number of times to process all the data chunks.

Every primitive is a fusion of basic actions, i.e., *send*, *recv*, *reduce*, and *copy*, which describe the basic operations on the four buffers used in collectives as shown in Fig. 5. *send/recv buffers* are local buffers for input and output. *send/recv connectors* contain lock-free ring buffers used for inter-GPU data transfer, managed by the *communicator*. The *send* action writes data to the *send connector*, while *recv* reads data from the *recv connector*. The *reduce* action reduces data from the *send buffer* and the *recv connector* with a specified *reducing function*. The *copy* action puts data into the *recv buffer*. Each primitive includes one or both of the *send* or *recv* actions. Based on the presence of *send* and *recv* actions, a primitive busy-waits until the *send connector* is writable and/or the *recv connector* is readable before progressing.

By limiting a primitive's wait time, we can abort its execution, thereby *preempting* the associated collective. Moreover, the *persistent visibility* of written data enables individual GPUs to independently preempt collectives in a *decentralized dynamic* manner, without explicit coordination among GPUs. Once a GPU writes data to the *send connector* for a collective's primitive, the data remain visible to the peer GPU. The visibility persists even if the collective part on this GPU is preempted after writing, or the corresponding collective part on the peer GPU is preempted before writing.

4.2 Collective Execution and Preemption

Fig. 6 demonstrates key components of the daemon kernel and identifies operations related to collective execution and preemption through numerical labels.

❶: The daemon kernel fetches and parses SQEs, and maintains collectives in the *task queue* in the shared memory [43].

❷: The daemon kernel traverses the task queue and schedules a collective, details of scheduling are discussed in Sec. 4.3.

❸: The daemon kernel executes the primitive sequence of the scheduled collective in a *two-phase blocking* manner [20, 49].

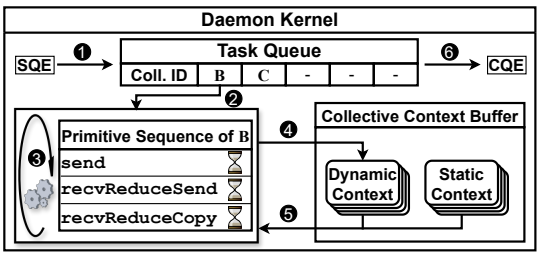


Figure 6. Daemon Kernel of *DFCCL*. Collective B and C are invoked, and B is scheduled currently.

The daemon kernel assigns *spin thresholds* to the collective's primitives to limit the busy-waiting time. During execution, a primitive first polls up to spin-threshold times to check if the condition required by the *send* and/or *recv* action is met. If the primitive cannot execute after polling spin-threshold times, it is aborted, and the associated collective is deemed to be *stuck* and then preempted on this GPU.

❹, ❺: The *context* of the preempted collective is saved in the *collective context buffer* in the global memory [43], and the daemon kernel loads the next scheduled collective's context into the *active context slot* in the shared memory. The context of collectives consists of *dynamic context* and *static context*. The dynamic context includes changing states during collective execution, e.g., the current data chunk ID and the aborted primitive's ID. The static context of a collective contains its constant configuration, such as local and *connector* buffers' addresses shown in Fig.5, and its meta information including the number of GPUs executing the collective, the rank of the current GPU among participants, and the composition of the collective's primitive sequence, etc. Before executing a collective, the daemon kernel loads both its dynamic and static context, yet only saving its dynamic context after preemption, as the static context remains unchanged during collective execution. However, the collective's static context can change across multiple calls, e.g., the addresses of *send buffer* and *recv buffer* may vary.

❻: The daemon kernel inserts a CQE into the CQ for the completed collective.

4.3 Adaptive Collective Scheduling

Algorithm 1 shows *DFCCL*'s scheduling process. *DFCCL*'s daemon kernel schedules collectives via the *adaptive stickiness adjustment scheme*, which enables priority assigning and decentralized dynamic gang-scheduling for collectives.

The *stickiness* of a collective indicates the daemon kernel's willingness to wait for its progress. A collective's stickiness is reflected in its position in the task queue and the spin thresholds assigned to its primitives. The unified stickiness adjustment mechanism independently controls both aspects on each GPU, supporting various policies. All the GPUs

Algorithm 1 Scheduling Process of *DFCCL*.

with *Pointer2SQ*, *SpinThreshold*, *PrimitiveExecuteStatus*, and *Pointer2CQ* **in** *shared memory*

```
1:  while not FinallyExit() do
2:      FetchSQE&SortTaskQueueByPriority()
3:      SetInitialSpinThreshold()
4:      for collective ∈ task queue do
5:          LoadCollectiveContext()
6:          for primitive ∈ collective do
7:              ExecutePrimitive()
8:              if PrimitiveSuccess() then
9:                  AdaptivelyAdjustSpinThreshold()
10:             else
11:                 PreemptCollective()
12:                 SaveCollectiveContext()
13:                 break
14:         if CollectiveSuccess() then
15:             SendCQE()
```

adopt the same stickiness adjustment policy, which is further divided into an *order adjusting policy* and a *spin threshold adjusting policy*. The order adjusting policy controls the frequency of fetching SQEs from the SQ and the ordering of collectives in the task queue, according to user-specified priorities (line 2). The spin threshold adjusting policy enables each GPU to negotiate in a decentralized dynamic manner to execute the same collective (line 3 and line 9).

When no specific priority is assigned to collectives, the daemon kernel adopts a FIFO-ordering policy; otherwise, it employs a priority-based ordering policy.

• **FIFO Ordering.** This policy aims to empty the task queue quickly. Under this policy, the daemon kernel fetches an SQE from the SQ when the task queue is empty or all collectives in the queue cannot progress for a while. The newly fetched collective is put at the end of the task queue.

• **Priority-based Ordering.** This policy optimizes performance for applications' special scheduling needs. Under this policy, the daemon kernel checks the SQ more frequently and sorts the task queue by priority. A practical priority scheme is assigning higher priority to collectives arriving later to enable the overlap of communication and computation in data parallelism [5, 35, 52].

When a collective is preempted, it remains at its original task queue position, and the daemon kernel schedules the next collective in the task queue for execution.

DFCCL's automated, adaptive spin threshold adjusting policy, which enables GPUs to negotiate in a decentralized dynamic manner to achieve gang-scheduling, plays a pivotal role in forestalling inter-GPU conflicts. This policy instructs the daemon kernel to assign the largest initial spin threshold to the collective at the front of the task queue. Each subsequent collective receives a progressively lower initial spin threshold (line 3). During execution, if the daemon kernel detects that a primitive of a collective executes successfully, it

raises the spin threshold of the succeeding primitives of that collective (line 9), which increases the probability that all GPUs simultaneously execute or wait for the same collective, resulting in de facto gang-scheduling.

DFCCL uses automated profiling to set suitable parameters, e.g., the initial spin threshold and voluntary quitting period (Sec. 4.4), to achieve the Pareto-optimal (Sec. 4.5).

4.4 Voluntary Quitting and Event-driven Starting

The daemon kernel voluntarily quits when it cannot fetch new SQEs for a certain period and the task queue is either empty or contains only collectives that cannot progress. The daemon kernel voluntarily quits for two reasons:

- To release GPU resources for other tasks when it's idle.
- To prevent deadlocks related to GPU synchronization. Once the daemon kernel quits, the blocking GPU synchronization can complete, thus allowing stuck collectives to proceed.

DFCCL tries to start the daemon kernel upon SQE insertion to the SQ or when the inserted CQEs are fewer than SQEs. The daemon kernel is initially launched upon the insertion of the first SQE. *dfcclDestroy* inserts an *exiting SQE* into the SQ, making the daemon kernel *finally exit* after reading it.

4.5 Correctness and Performance Analysis

Correctness. The daemon kernel ensures the correctness of dynamic, decentralized preempting and restoring of collectives, by preserving the context integrity of preempted, uncompleted collectives. Saving and loading the preempted collective's dynamic context ensures it restarts from the previous stopping point, without under- or re-transmitting data. The daemon kernel prevents other collectives from using preempted, uncompleted collective's *connectors*, ensuring the correct exploitation of the data visibility (Sec. 4.1). Besides, the daemon kernel's voluntary quitting and restarting do not corrupt preempted collectives' context in global memory.

Performance Modeling. The overheads T in collective execution includes the busy-waiting time ($t[spin]$), the context switch time ($t[switch]$), and the waiting time for scheduling influenced by task queue length ($t[q\_len]$).

$$T = t[spin] + t[switch] + t[q\_len] \tag{1}$$

$t[spin]$ correlates positively with the spin threshold (N_{spin}). Both $t[switch]$ and $t[q\_len]$ are negatively correlated with N_{spin}: a larger N_{spin} increases the probability that a collective successfully waits for the same collective's scheduling on peer GPUs, so it experiences fewer preemptions and context switches, and completes faster, thereby reducing the task queue length. Therefore, we can assess the correlation between the overheads T and the spin threshold N_{spin} via expression 2. In *DFCCL*, adaptively adjusting the spin threshold, as a uniform approach, is used to approximate a Pareto-optimal [18] for overheads in various scenarios.

$$T \sim N_{spin} + \frac{1}{N_{spin}} \tag{2}$$

4.6 Discussions

DFCCL introduces at least two innovations for deadlock-free GPU collectives: ❶ A daemon kernel that achieves two-phase blocking execution of GPU collectives on the hardware platform without preemption support, without altering the underlying GPU task scheduling mechanism. ❷ A collective preemption and scheduling co-design that enables decentralized, dynamic collaboration of multiple GPUs without explicit coordination among them, enhancing traditional timer-based context switch in preemptive scheduling [4].

During the two-phase blocking execution, *DFCCL* can preempt a collective at any time by interrupting any primitive. The specific method of interrupting a primitive involves assigning an appropriate spin threshold before its execution, causing it to yield if no progress is made within the spin threshold. In contrast, in NCCL, a primitive busy-waits indefinitely while holding resources.

5 Implementation and Optimizations

Implementation Details of the Daemon Kernel We tailor the daemon kernel to the block-thread programming model of CUDA [43]. A CUDA kernel comprises multiple parallel threads executing the same code. The threads are grouped into blocks. All threads of a block reside in the same streaming multiprocessor (SM) and have access to the SM's limited shared memory. Equally-shaped blocks further form a grid. Each collective is assigned a specific grid and block sizes. The daemon kernel is launched using the largest grid and block sizes among all registered collectives.

Threads within the same block synchronize easily, while those in different blocks typically run asynchronously. When the daemon kernel executes collectives requiring different numbers of blocks simultaneously, its higher-index blocks can execute a different collective than lower-index blocks. The number of active threads executing a collective's primitives inside a block depends on the block size assigned to the collective. The daemon kernel makes extra threads wait. Each block independently decides when to quit voluntarily.

The daemon kernel's reads from the SQ and writes to the CQ are implemented to accommodate asynchronous block execution. The SQ is a single-producer-multi-consumer (SPMC) ring buffer, allowing only one CPU thread to write an SQE at a time. All blocks of the daemon kernel read the SQE. When a block reads a new SQE, it atomically increases a counter inside the SQE. If a block finds the increased counter of an SQE equals the daemon kernel's grid size, it marks the corresponding SQ slot as writable. A block executes a

collective only if its index is lower than the collective's assigned grid size. The CQ is a multi-producer-single-consumer (MPSC) ring buffer. Only one poller thread on the CPU reads CQEs. The daemon kernel maintains a completion counter for each collective in global memory. A block atomically increases the collective's completion counter when completing its part of a collective. If a block finds the increased completion counter of a collective equals its assigned grid size, the block writes a CQE to the CQ and resets the collective's completion counter. Multiple blocks can concurrently write CQEs into the CQ for different completed collectives.

The CPU & GPU cache-coherence mechanism is transparent to SQ/CQ management, and we use memory fences for memory consistency when necessary. We use CUDA's atomic APIs that ensure transparent atomic read/write operations between GPUs and DRAM, as well as within a GPU.

Optimizing CQ. The SQ and CQ reside in page-locked host memory. We reduce host-memory-related reads, writes and memory fences when writing CQEs to decrease CQE-writing latency by leveraging CUDA's 64-bit atomic operations. The **vanilla ring-buffer-based CQ** requires at least five host-memory-related operations to prevent blocks from inserting CQEs for different collectives into the same CQ slot. The vanilla ring-buffer-based CQ also requires a memory fence between writing the CQE and updating the CQ's *tail* to ensure memory consistency. The **optimized ring-buffer-based CQ** uses exactly four host-memory-related operations without fences by encapsulating the complete collective's ID and current *tail* in a single 64-bit atomic write. The poller validates a CQE by comparing CQ's *head* and the *tail* from the 64-bit bitmap. We further develop an **optimized CQ** that only requires at least a single CUDA *atomicCAS_system* operation to write a CQE, abandoning the ring buffer semantics. This optimized CQ is based on the observation that the CQE only carries the complete collective's ID. A block atomically writes the ID into a writable slot in the CQ. The poller scans the CQ, checking whether a slot contains a valid collective ID and marking the slot writable after reading the collective ID. SQ is implemented as a vanilla ring buffer because an SQE contains more information than a 64-bit word can hold.

Reducing the Overheads of Context Switching. We employ three methods to minimize context-switching overheads. ❶ The daemon kernel loads and saves the context in parallel with multiple threads. We encapsulate the dynamic and static context into 16-byte aligned *structs* to utilize the 16-byte *load/store* instructions. ❷ The daemon kernel uses multiple active context slots in shared memory, managed with a direct-mapped cache approach. ❸ The daemon kernel employs a lazy-saving strategy, only saving the dynamic context of a collective that has progressed before preemption.

Integrating* DFCCL *with Frameworks. We extend OneFlow [66] as well as PyTorch [51] & Megatron-LM [30, 59] to use *DFCCL*-based collectives. The integration with each framework requires approximately 1,000 lines of C++ code to invoke proper *DFCCL* APIs.

Table 2. Specifications of the Experimental Platforms.

	Specification
Processor	Intel Xeon Silver 4314 @ 2.40GHz (16 cores × 2 sockets)
DRAM	512GB @ 2666 MT/s
GPU	NVIDIA GeForce RTX 3080 Ti 12GB × 8
NIC	Mellanox MT28908 @ 56Gb/s
Processor	Intel Xeon Silver 4314 @ 2.40GHz (16 cores × 2 sockets)
DRAM	512GB @ 2666 MT/s
GPU	NVIDIA GeForce RTX 3090 24GB × 8
NIC	Mellanox MT28908 @ 56Gb/s
Switch	Mellanox SX6036 (36 full-duplex 56Gb/s ports)

6 Evaluation

In this section, we conduct experiments to verify *DFCCL*'s deadlock prevention capability and measure its performance.

• **Testbed.** We conduct experiments on the platforms detailed in Table 2, using Ubuntu 20.04 and CUDA 11.7. On the dual-socket servers, GPUs 0-3 and GPUs 4-7 belong to two separate *PIX* domains, and these two device groups reside within the *SYS* domain. GPUs within each machine communicate via the Shared Memory (SHM) transports, while inter-machine communication utilizes RDMA networking. These machines are hereafter referred to as the 3080ti-server and the 3090-server. The primitive sequences for collectives are generated with Simple protocol and Ring algorithm [46].

• **Benchmarks.** ❶ Verifying *DFCCL*'s deadlock-prevention capability. ❷ Measuring *DFCCL*'s workload-independent overheads. ❸ Evaluating the bandwidth and latency of common GPU collectives based on NCCL Tests [42]. ❹ Evaluating the performance of DNN training.

• **Comparing Targets.** We compare the bandwidth and latency of collectives from *DFCCL* with those from NCCL [46]. To evaluate training performance, we conducted three sets of comparative experiments: ❶ In data-parallel scenarios, we compare ResNet50 [24] training throughput (#samples consumed per second) with *DFCCL* versus that with NCCL orchestrated by different CPU-based methods. The comparing targets include Horovod v0.28.1, KungFu v0.2.5, and static sorting from OneFlow v0.8.1. ❷ To demonstrate *DFCCL*'s applicability and performance under various distributed training methods, we compare the training throughput of Vision Transformer (ViT) [19] in OneFlow v0.8.1 at different scales and with different distributed training techniques. ❸ To showcase *DFCCL*'s performance in more popular and recent scenarios, we compare the GPT-2 [26, 53] training performance with *DFCCL* against that with manually orchestrated NCCL in PyTorch v2.2.1 & Megatron-LM 23.06.

6.1 *DFCCL*'s Deadlock-preventing Capability

We develop testing programs to demonstrate the deadlock-preventing capability of *DFCCL*. The testing programs directly invoke GPU collectives and GPU synchronization operations according to the basic GPU deadlock situations at bottom library level discussed in Sec. 2.3.

• In the first program, eight GPUs, each using a unique random launch order, invoke the same set of eight all-reduces with buffer sizes from 256B to 1MB. Results on the 3090-server show that, without a dedicated stickiness adjustment policy, all GPUs successfully execute the eight *DFCCL*-based all-reduces for 200 iterations. Approximately 18,000 preemptions occur for each block on average.

• In the second program, we insert *cudaDeviceSynchronize()* calls as GPU synchronization between all-reduces invoked in different orders on eight GPUs. Results on the 3090-server show that over 200 iterations, the daemon kernel on each GPU voluntarily quits for 360 times on average, ensuring successful execution of the *DFCCL*-based all-reduces.

NCCL's deadlock ratio is 100% in the testing programs.

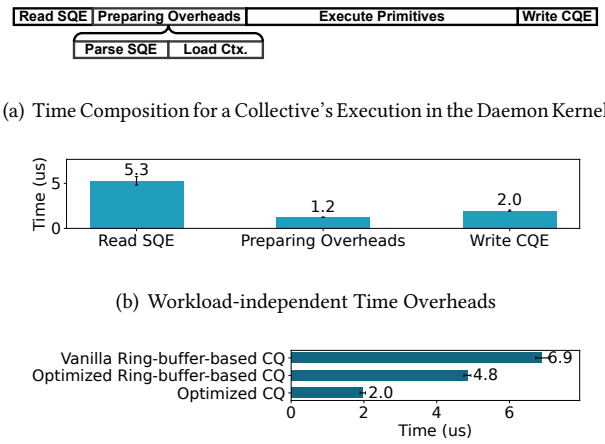
The testing programs are representative of GPU collective deadlock situations that can happen during distributed DNN training, because different characteristics at application level, e.g., model type, training scale, and parallelism-type, do not introduce other basic GPU collective deadlock situations at bottom library level.

6.2 Workload-independent Overheads

Workload-independent overheads, which do not increase linearly with workload (buffer size), are divided into **memory overheads** and **time overheads**. ❶ Workload-independent memory overheads include shared memory for each block's task queue and active context slot, and global memory for the collective context buffer and other related data structures. ❷ Workload-independent time overheads in *DFCCL* include the time required for executing multiple collectives within the daemon kernel, including loading and saving context.

DFCCL requires 13KB of shared memory and 4MB of global memory per block to maintain the block-dedicated task queue and collective context buffer for 1,000 collectives. Another 11KB of global memory is needed to keep the completion counters of collectives and other information accessible to all blocks.

Fig. 7(a) illustrates the timeline of executing a collective in the daemon kernel without preemptions. Fig. 7(b) illustrates the SQE read time, the preparing overheads, and the CQE write time when executing all-reduce on the 3090-server's eight GPUs. Fig. 7(c) shows the optimized CQ, detailed in Sec. 5, reduces CQE write time from about 6.9 μs to about 2.0 μs. The context loading takes approximately 0.45 μs, while saving context requires about 0.05 μs.



(a) Time Composition for a Collective's Execution in the Daemon Kernel

(b) Workload-independent Time Overheads

(c) Time Taken to Write CQE to Different Versions of CQ

Figure 7. Workload-independent Time Overheads Analysis.

6.3 Bandwidth and Latency of Collectives

We rewrite the NCCL Tests [42] based on commit 8274cb4 to evaluate the bandwidth and latency of *DFCCL*-based collectives, and compare *DFCCL* with NCCL 2.12.12. Fig. 8 presents selected results. These results are obtained by averaging three experiments, with five iterations each. *DFCCL* achieves comparable algorithm bandwidth and latency to NCCL across various scales and buffer sizes.

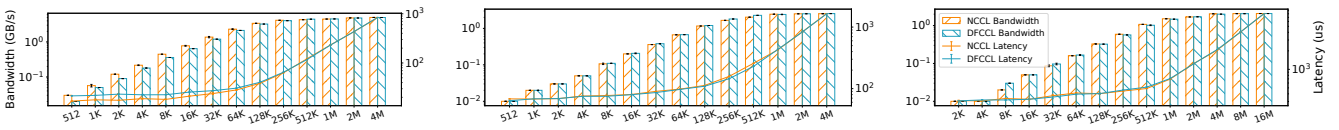
To further analyze *DFCCL*'s performance gains and overheads, we measure and compare the end-to-end latency and core execution time of collectives in *DFCCL* and NCCL with different buffer sizes. The results are shown in Fig. 9.

For *DFCCL*, the core execution time includes "preparing overheads" and "execute primitives" time, as shown in Fig. 7(a). For NCCL, it is the execution time of the dedicated kernel for a collective.

Fig. 9(a) indicates that with a small buffer, the end-to-end latency of the collective from *DFCCL* is about 4 μs higher than that from NCCL, yet *DFCCL*'s core execution time is shorter. Meanwhile, when dealing with a larger buffer, as illustrated in Fig. 9(b), *DFCCL*'s end-to-end latency is about 3 μs lower than NCCL's, and the core execution time of *DFCCL* is roughly 20 μs shorter than that of NCCL.

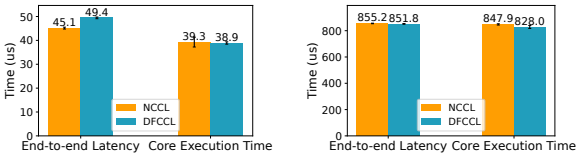
The reduced core execution time of collectives in *DFCCL* results from the fusion of collectives within the daemon kernel, both spatially and temporally. In *DFCCL*, the daemon kernel fuses concurrently invoked collectives and multiple iterations of a single collective.

When a collective requires fewer primitives, its core execution time is shorter. This makes I/O latency a more significant contributor to the collective's end-to-end latency, and renders the core execution time savings in *DFCCL* insufficient to offset its I/O latency, e.g., in Fig. 9(a) and when the buffer size



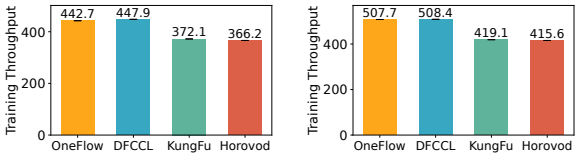
(a) 3080ti-server, 8 GPUs, Broadcast (b) 3090-server, 8 GPUs, All-reduce (c) 2×3080ti-&2×3090-server, 32 GPUs, All-reduce

Figure 8. Algorithm Bandwidth and End-to-end Latency of Collectives with Different Buffer Sizes.



(a) All-gather with **4KB** Buffer (b) All-gather with **4MB** Buffer

Figure 9. Case Study: Comparing End-to-end Latency and Core Execution Time of NCCL and *DFCCL* with Small and Large Buffer Sizes on eight GPUs of the 3090-server.



(a) 3080ti-server (b) 3090-server

Figure 10. Average Throughput of Training ResNet50 with Data Parallelism for 200 Iterations.

is less than 64KB in Fig. 8(a). We will prioritize optimizing *DFCCL*'s I/O handling scheme in future work.

Nonetheless, as the workload increases, the reduction in core execution time achieved by *DFCCL* gradually compensates for its I/O latency, ultimately leading to a decrease in the overall end-to-end latency as shown in Fig. 8.

6.4 DNN Training Performance

Results in this section show that *DFCCL*, using a *unified* on-GPU adaptive scheduling scheme, achieves training performance on par with or exceeding NCCL, which requires *different* CPU orchestration methods tailored to specific scenarios. *DFCCL*'s efficiency also relates to its ability to execute deadlock-free collectives concurrently and asynchronously without extra CPU coordination, and low overheads for deadlock identification and context switch. In the experiments, we observe different GPUs launching collectives in various orders when using *DFCCL*, and *no deadlocks* arise. The loss convergence rate with *DFCCL* remains similar to NCCL.

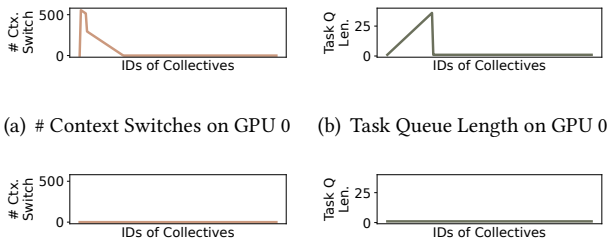
6.4.1 Data Parallel ResNet50 Training Experiments are conducted using eight GPUs on each of the 3080ti-server and 3090-server. Per-GPU batch sizes are set to 48 and 96 for the 3080ti- and 3090-server, respectively. We report the average training throughput over 200 iterations.

Fig. 10 illustrates that, the training throughput achieved with *DFCCL* is comparable to that attained with statically sorted NCCL in OneFlow, with improvements up to 1.2%. *DFCCL* outperforms KungFu and Horovod by 20.4%-22.3%.

Case Study: Assessing the Impact of the Adaptive Scheduling on Performance. Fig. 11 displays two sets of statistical data from a ResNet50 training iteration with four GPUs on the 3090-server: the number of context switches and task queue lengths on different GPUs. The X-axis of the figures in Fig. 11 represents IDs of the collectives used in ResNet50 training. These IDs are displayed in the invocation order on each GPU. Fig. 11(a) and Fig. 11(c) display the number of context switches of each collective, i.e., the number of times the collective is preempted before its completion in the iteration. Meanwhile, Fig. 11(b) and Fig. 11(d) illustrate the task queue length after the daemon kernel reads the SQE of the corresponding collective. The context switch and task queue length on GPU 1 & 3 is similar to that on GPU 0.

The spikes in Fig. 11 are due to a naive spin threshold adjustment policy, where each collective's spin threshold is fixed at 10,000 and does not adaptively change. During the backward pass of DP, collectives are invoked in bursts. The scenario in Fig. 11 occurs when GPU 2 slightly delays issuing collectives, while the other three GPUs aggressively fetch new SQEs because all fetched collectives fail to progress. As a result, collectives accumulate in the task queues of the three GPUs, and collectives fetched earlier are preempted multiple times. Such a situation causes the training throughput to drop from over 500 to less than 100.

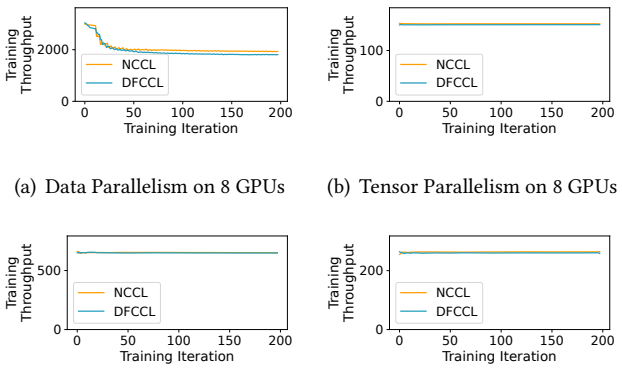
By applying the adaptive spin threshold adjusting policy, which allocates a larger initial spin threshold (100,000) via profiling for the collective at the front of the task queue, and utilizes a twentyfold-greater spin threshold after primitive success, *DFCCL* eliminates the spike in Fig. 11, and maintains high training throughput by achieving gang-scheduling.



(a) # Context Switches on GPU 0 (b) Task Queue Length on GPU 0

(c) # Context Switches on GPU 2 (d) Task Queue Length on GPU 2

Figure 11. Statistical Data for Assessing the Impact of the Adaptive Scheduling on Performance.



(a) Data Parallelism on 8 GPUs (b) Tensor Parallelism on 8 GPUs

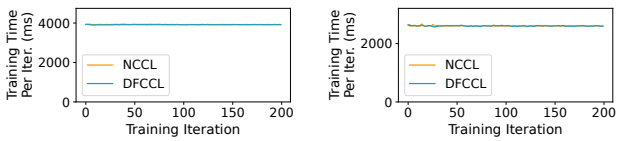
(c) 3D Hybrid on 16 GPUs (Base) (d) 3D Hybrid on 16 GPUs (Large)

Figure 12. Throughput for Training ViT Using OneFlow on 3090-Servers (Higher is Better).

6.4.2 Distributed ViT Training Fig. 12 shows the ViT model's training throughput under various distributed training techniques. Each iteration reports the average throughput from the start to the current iteration [64]. Fig.12(a)-12(c) employ the base ViT configuration, while Fig.12(d) utilizes the large ViT configuration. The microbatch size is 128. Experiments use eight GPUs on a single 3090-server (Fig. 12(a)-12(b)), and 16 GPUs across two 3090-servers (Fig. 12(c)-12(d)).

Results in Fig. 12 show that *DFCCL* efficiently provides the required collectives for various parallel DNN training methods. Compared to NCCL statically sorted or manually orchestrated by OneFlow, *DFCCL* delivers comparable performance. In Fig. 12(a), *DFCCL* exceeds NCCL by up to 8.6% and maintains a gap within 7.4%. In other experiments, the difference between *DFCCL* and NCCL falls within ±3%.

6.4.3 Distributed GPT-2 Training Fig. 13 shows the per-iteration training time of GPT-2 [26, 53] under 3D-hybrid parallelism with Megatron-LM & PyTorch. The microbatch size is 18. Results demonstrate that *DFCCL* provides comparable training performance to NCCL that is manually orchestrated by Megatron-LM & PyTorch, exhibiting performance differences within ±4%. Furthermore, *DFCCL* achieves training stability comparable to NCCL. On a single 3090-server, *DFCCL* exhibits a 1.4% coefficient of variation in per-iteration training time over 200 iterations, compared to NCCL's 1.5%. When scaling to 16 GPUs across two 3090-servers, the coefficient of variation for per-iteration training time remains comparable, with 4.3% for *DFCCL* and 3.9% for NCCL.



(a) 3D Hybrid on 8 GPUs (b) 3D Hybrid on 16 GPUs

Figure 13. Time Per Iteration for Training GPT-2 Using PyTorch & Megatron-LM on 3090-Servers (Lower is Better).

7 Related Work

Collective Scheduling. Some collective scheduling work aims to overlap communication and computation. Poseidon [67], TicTac [23], P3 [28] are proposed in the context of parameter-server-based [34] data parallelism, and the core insights are applied to all-reduce-based data parallelism. Poseidon [67] proposes wait-free backpropagation, synchronizing gradients after each layer rather than a whole iteration. TicTac [23] adjusts gradient synchronization order so that the next iteration can launch earlier. P3 [28] (priority-based parameter propagation) slices parameters into finer granularity, allowing synchronization of higher-priority parameters to preempt that of lower priority. Bytescheduler [52] applies a similar idea to both parameter server and all-reduce architectures, and introduces an approach to auto-tuning slice size. PACE [5] preemptively schedules segmented all-reduce kernels based on the directed acyclic graph (DAG) of DNN. PyTorch Distributed [35] initiates all-reduce earlier than the end of the local backward pass to overlap computation with communication. CoCoNet [27] fuses split all-reduce with the computation consuming its results and overlaps the fused kernels. *DFCCL* can support the above scheduling policies.

Collective Optimization. Blueconnect [10] decomposes an all-reduce to many parallelizable reduce-scatters and all-gathers to adapt to network topology. BLink [62] proposes a heuristic spanning tree packing algorithm to optimize collective primitives. PLink [37] exploits the hierarchical network topology to construct a logical topology for collectives. SCCL [8] solves an integer programming encoding

to achieve the Pareto-frontier of latency- and bandwidth-optimal algorithms in a single machine. TACCL [58] synthesizes collective algorithms for multi-node topologies utilizing bandwidth and latency probes. MSCCLang [17] provides a domain-specific language describing the chunk-oriented dataflow of collectives. Themis [54] dynamically assigns unique pipeline schedules to data chunks in collectives to maximize utilization of all network dimensions. Existing collective optimization techniques utilize underlying physical topology to adjust data segmentation, chunk routing, aggregation hierarchy, etc., complementing *DFCCL* orthogonally.

GPU Preemption. Prior work in literature aims to support preemption for GPU to lower the end-to-end latency of high-priority tasks. Hardware solutions [36, 50, 60] enhance the hardware to support preemption, managing context such as registers, shared memory, barrier states, etc., during runtime. Software methods [9, 22, 32, 33, 63, 68] can be applied directly on commodity GPUs. Wait-based preemption approaches [9, 63, 68] modify user kernels to insert scheduling points so that user kernels quit more frequently and expose more scheduling chances. Lee et al. [32, 33] and REEF [22] kill the preempted kernel directly to decrease scheduling delay. Preempted idempotent kernels can be aborted and then restarted without affecting correctness [22, 32], while preempted non-idempotent kernels are rolled back and then relaunched [32, 33]. *DFCCL* supports preemption without hardware modification. Wait-based preemption still cannot prevent deadlocks. Collectives are non-idempotent, and rolling back collectives introduces considerable overheads and complicated synchronization issues among GPUs. Existing GPU preemptive scheduling methods are limited to single-GPU kernels. *DFCCL* achieves decentralized dynamic preemption and adaptive scheduling for collectives across multiple GPUs.

8 Conclusion

GPU collective deadlocks pose threats to distributed deep learning. We present *DFCCL*, a novel GPU collective communication library that provides a comprehensive approach to prevent GPU collective deadlocks by supporting preemption, and maintaining high performance via adaptive scheduling. Experimental results show that *DFCCL* effectively prevents GPU collective deadlocks and achieves performance comparable to or superior to NCCL. The code of *DFCCL* and the simulator is publicly available at https://github.com/Oneflow-Inc/dfccl.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was supported by the National Key R&D Program of China under Grant 2023YFB2703800. The contact author is Zhen Xiao.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] AMD. 2023. IOMMU Advisory for Multi-GPU Environments. [Online]. https://community.amd.com/t5/knowledge-base/iommu-advisory-for-multi-gpu-environments/ta-p/477468.

[3] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. 2018. AI and Compute. [Online]. https://openai.com/blog/ai-and-compute/.

[4] S Tanenbaum Andrew and Bos Herbert. 2015. *Modern operating systems*. Pearson Education.

[5] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. 2020. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 626–635.

[6] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ML. *arXiv preprint arXiv:2203.12533* (2022).

[7] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. 2021. Maximizing Parallelism in Distributed Training for Huge Neural Networks. *CoRR* abs/2105.14450 (2021). arXiv:2105.14450 https://arxiv.org/abs/2105.14450

[8] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 62–75.

[9] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/3018743.3018748

[10] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems* 1 (2019), 241–251.

[11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[12] Pytorch Community. 2019. Multi-gpu example freeze and is not killable. [Online]. https://github.com/pytorch/pytorch/issues/24081.

[13] Pytorch Community. 2019. pin_memory stuck in DDP/Reducer constructor. [Online]. https://github.com/pytorch/pytorch/issues/31095.

[14] Pytorch Community. 2021. Deadlock in a single machine multi-gpu using dataparlel when cpu is AMD. [Online]. https://github.com/pytorch/pytorch/issues/52142.

[15] PyTorch Community. 2023. Waring in Process Group. [Online]. https://pytorch.org/docs/stable/distributed.html.

[16] TensorFlow Community. 2020. tf.distribute.MirroredStrategy is stuck. [Online]. https://github.com/tensorflow/tensorflow/issues/44976.

[17] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft Collective Communication Language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 502–514. https://doi.org/10.1145/3575693.3575724

[18] Kalyanmoy Deb. 2011. Multi-objective optimisation using evolutionary algorithms: an introduction. In *Multi-objective evolutionary optimisation for product design and manufacturing*. Springer, 3–34.

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YicbFdNTTy

[20] Dror G Feitelson and Larry Rudolph. 1992. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing* 16, 4 (1992), 306–318.

[21] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, and Kurt Keutzer. 2021. AI and Memory Wall. *RiseLab Medium Post* (2021). Retrieved Feb. 2, 2023 from https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

[22] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[23] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems* 1 (2019), 418–430.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[26] Hugging Face. 2021. CodeParrot. [Online]. https://huggingface.co/codeparrot/codeparrot.

[27] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.

[28] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *Proceedings of Machine Learning and Systems* 1 (2019), 132–145.

[29] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous {GPU/CPU} Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[30] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling Large Language Model Training to More Than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[31] Jiri Kraus. 2013. An Introduction to CUDA-Aware MPI. [Online]. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

[32] Hyeonsu Lee, Hyunjun Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. 2021. Idempotence-Based Preemptive GPU Kernel Scheduling for Embedded Systems. *IEEE Trans. Comput.* 70, 3 (2021), 332–346. https://doi.org/10.1109/TC.2020.2988251

[33] Hyeonsu Lee, Jaehun Roh, and Euiseong Seo. 2018. A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 202–213. https://doi.org/10.1109/RTAS.2018.00029

[34] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[36] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, John West and Cherri M. Pancake (Eds.). IEEE Computer Society, 898–908. https://doi.org/10.1109/SC.2016.76

[37] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems* 2 (2020), 82–97.

[38] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 937–954.

[39] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216.

[40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[41] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[42] NVIDIA. 2021. NCCL Tests. [Online]. https://github.com/NVIDIA/nccl-tests.

[43] NVIDIA. 2023. CUDA C++ Programming Guide. [Online]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[44] NVIDIA. 2023. CUDA Stream Management. [Online]. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[45] NVIDIA. 2023. Inter-GPU Communication with CUDA-aware MPI. [Online]. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/mpi.html.

[46] NVIDIA. 2023. NVIDIA NCCL. [Online]. https://github.com/nvidia/nccl.

[47] NVIDIA. 2023. PCI Access Control Services (ACS). [Online]. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/troubleshooting.html.

[48] NVIDIA. 2023. Using Multiple NCCL Communicators Concurrently. [Online]. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/

usage/communicators.html.

[49] John K Ousterhout et al. 1982. Scheduling Techniques for Concurrent Systems.. In *ICDCS*, Vol. 82. 22–30.

[50] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 593–606. https://doi.org/10.1145/2694344.2694346

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[52] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[54] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 581–596. https://doi.org/10.1145/3470496.3527382

[55] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[56] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[57] Jaime Sevilla and Pablo Villalobos. 2021. Parameter Counts in Machine Learning. [Online]. https://www.alignmentforum.org/posts/GzoWcYibWYwJva8aL/parametercounts-in-machine-learning.

[58] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 593–612.

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[60] Ivan Tanasić, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 193–204. https://doi.org/10.1109/ISCA.2014.6853208

[61] MVAPICH Team. 2023. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot. [Online]. https://mvapich.cse.ohio-state.edu/.

[62] 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.

[63] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 483–496. https://doi.org/10.1145/3037697.3037742

[64] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. 2019. Detectron2. [Online]. https://github.com/facebookresearch/detectron2.

[65] Qifan Xu, Shenggui Li, Chaoyu Gong, and Yang You. 2021. An Efficient 2D Method for Training Super-Large Deep Learning Models. *CoRR* abs/2104.05343 (2021). arXiv:2104.05343 https://arxiv.org/abs/2104.05343

[66] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. 2021. OneFlow: Redesign the Distributed Deep Learning Framework from Scratch. *arXiv preprint arXiv:2110.15032* (2021).

[67] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.

[68] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97. https://doi.org/10.1109/RTAS.2015.7108420

A Artifact Appendix

The artifact for the EuroSys 2025 paper, "Comprehensive Deadlock Prevention for GPU Collective Communication", provides the code of *DFCCL* and the simulator mentioned in Sec. 2.4. The DOI of the artifact is at https://doi.org/10.5281/zenodo.14871978.

The code of *DFCCL* and the simulator is also publicly available at https://github.com/Oneflow-Inc/dfccl.