HongTu: Scalable Full-Graph GNN Training on Multiple GPUs (via communication-optimized CPU data offloading)

QIANGE WANG, National University of Singapore, Singapore
YAO CHEN, National University of Singapore, Singapore
WENG-FAI WONG, National University of Singapore, Singapore
BINGSHENG HE, National University of Singapore, Singapore

Full-graph training on graph neural networks (GNN) has emerged as a promising training method for its effectiveness. Full-graph training requires extensive memory and computation resources. To accelerate this training process, researchers have proposed employing multi-GPU processing. However the scalability of existing frameworks is limited as they necessitate maintaining the training data for every layer in GPU memory. To efficiently train on large graphs, we present HongTu, a scalable full-graph GNN training system running on GPU-accelerated platforms. HongTu stores vertex data in CPU memory and offloads training to GPUs. HongTu employs a memory-efficient full-graph training framework that reduces runtime memory consumption by using partition-based training and recomputation-caching-hybrid intermediate data management. To address the issue of increased host-GPU communication caused by duplicated neighbor access among partitions, HongTu employs a deduplicated communication framework that converts the redundant host-GPU communication to efficient inter/intra-GPU data access. Further, HongTu uses a cost model-guided graph reorganization method to minimize communication overhead. Experimental results on a 4×A100 GPU server show that HongTu effectively supports billion-scale full-graph GNN training while reducing host-GPU data communication by 25%-71%. Compared to the full-graph GNN system DistGNN running on 16 CPU nodes, HongTu achieves speedups ranging from 7.8× to 20.2×. For small graphs where the training data fits into the GPUs, HongTu achieves performance comparable to existing GPU-based GNN systems.

CCS Concepts: • **Information systems → Data management systems**; • **Computing methodologies → Parallel computing methodologies**.

Additional Key Words and Phrases: Graph neural networks; GNN training; GPU; CPU data offloading

ACM Reference Format:
Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs (via communication-optimized CPU data offloading). *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 246 (December 2023), 28 pages. https://doi.org/10.1145/3626733

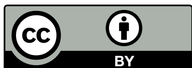
1 INTRODUCTION

Graph neural networks (GNNs) have gained increasing popularity for their effectiveness in modeling graph data [7, 12, 21, 24, 28, 29, 31, 52, 57–59]. By iteratively aggregating parameterized neighbor representations through graph propagation, GNNs can capture the topology and feature information at the same time and generate more informative representations for the downstream tasks.

Recently, full-graph GNN training that trains on the entire graph, has emerged as a promising GNN training method for its effectiveness brought by full-neighbor aggregation semantic and

246

Authors' addresses: Qiange Wang, National University of Singapore, Singapore, wangqg@comp.nus.edu.sg; Yao Chen, National University of Singapore, Singapore, yaochen@comp.nus.edu.sg; Weng-Fai Wong, National University of Singapore, Singapore, wongwf@comp.nus.edu.sg; Bingsheng He, National University of Singapore, Singapore, hebs@comp.nus.edu.sg.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).
2836-6573/2023/12-ART246
https://doi.org/10.1145/3626733

Table 1. Memory consumption of graph topology, vertex (**Vtx**) data, and intermediate (**Intr**) data for 3-layer full-graph GCN training on three billion-scale graphs.

Dataset	Model Config	Topology	Vtx Data	Intr Data
it-2004	256-128-128-64	12.8GB	177.2GB	108.3GB
ogbn-paper	200-128-128-172	18.0GB	519.4GB	425.3GB
friendster	256-128-128-64	28.9GB	293.3GB	179.3GB

full-batch gradient descent [18, 32, 56]. However, full-graph training requires high computing power. The training involves random vertex data access and neural network computation, requiring high memory bandwidth and massive parallel computation. Considering the increasing sizes of real-world graphs, research efforts have been made toward extending GNN training to GPU platforms [4, 18, 30, 42, 51, 53, 56]. These frameworks partition the input graphs, parallelize the computation across multiple GPUs, and handle remote neighbor aggregations through inter-GPU communication.

Despite the significant performance improvement achieved through massive parallel processing, scaling GPU-based GNN training frameworks to large graphs remains challenging due to the limited capacity of GPU memory. In GNN training, the core data that need to be maintained in GPU memory includes vertex data and intermediate data. Vertex data consist of the vertex representations (feature) and vertex gradients of every layer, while intermediate data are the intermediate results of neural network models, which are generated in the forward computation and consumed in the gradient computation in the backward pass. For real-world graphs, the data often exceed the device memory capacity even with multiple GPUs. As illustrated in Table 1, both the vertex data and intermediate data can occupy hundreds of gigabytes of GPU memory[1]. Moreover, in practice, vertex data and intermediate data are only the basic data for model training, and additional memory must be reserved for auxiliary data such as graph topology, communication buffers, and neighbor replicas. The experiments in DistGNN [32] demonstrate that running a 3-layer GraphSAGE model on the ogbn-paper graph [16], with 111 million vertices and 1.6 billion edges, requires 6 terabytes of memory in a cluster with 16 shared-nothing CPU nodes [32]. Building a GPU memory pool of comparable size requires significant monetary cost and engineering effort, which limits the graph scale that existing full-graph GNN systems can handle.

Recently, mini-batch GNN training has been proposed as an alternative approach to training large graphs on memory-constrained platforms. This method enables users to train on batched and sampled subgraphs, thus reducing the memory requirements [9, 27, 50, 61, 62, 64]. However, mini-batch training may suffer from low accuracy caused by information loss [18, 32, 50, 56].

In this work, we aim to accelerate large-scale full-graph GNN training beyond GPU memory capacity by leveraging partition-based CPU-GPU heterogeneous processing, which partitions and stores data in CPU memory and offloads computation to GPUs. This approach has been widely adopted in GPU applications that manage data beyond GPU memory capacity, including database analytical processing [11, 23] and graph data analytics [34, 48]. However, traditional partition-based processing approaches face two significant challenges when handling neural network computation and high-dimensional graph propagation in full-graph GNN training.

Firstly, partition-based processing cannot effectively reduce memory consumption of intermediate data. In DNN training, data samples can be randomly partitioned into disjoint subsets for parallel training, which reduces the memory consumption of both vertex and intermediate data since there are no dependencies among the data samples. However, GNN training is distinct

[1]The memory consumption of intermediate data varies across GNN models and can be much larger in GNNs involving complex edge computation [25, 52].

from it because the graph propagation computation creates cross-partition data dependencies. To compute the gradients of a partitioned subgraph, gradients of all its dependent partitions from downstream layers must also be computed [56], which in turn requires storing intermediate data for these partitions. As a result, although partition-based processing enables loading the vertex data of a small partitioned slice at a time, a significant amount of GPU memory still needs to be reserved for maintaining the intermediate data.

Secondly, partition-based processing leads to increased host-GPU communication. In GNN training, graph propagation involves aggregating features from neighboring vertices. This requires loading the data of the entire neighbor set onto GPUs when processing each partitioned subgraph. However, when a large graph is split into multiple partitions, vertices with multiple outgoing edges may be replicated to multiple partitions as neighbors. As a result, it is necessary to transfer them multiple times during training, which increases host-GPU communication. Moreover, since high-dimensional vertex attributes can consume a substantial amount of memory (as shown in Figure 1), it is not feasible to store the frequently accessed vertex data entirely in GPU, as is done in GPU-accelerated graph analytical frameworks [34, 47, 63].

We present HongTu, a GPU-accelerated full-graph GNN training system that addresses the challenges of traditional partition-based processing through two critical functions. Firstly, HongTu employs a **memory-efficient GNN training framework** that reduces runtime memory consumption of both vertex and intermediate data. This framework integrates a GNN-friendly partition method and a cost-effective recomputation-caching-hybrid intermediate data management method. Inspired by the recomputation-based DNN training method that avoids storing intermediate data by releasing intermediate data in the forward pass and recomputing it in the backward pass[5]. Based on the original method, our recomputation-caching-hybrid method further combines GPU-based recomputation and CPU-based data caching to reduce the recomputation overhead in GNNs. Secondly, HongTu employs a **deduplicated communication framework** that reduces host-GPU communication for duplicated neighbor access among partitions. We observe that duplicated neighbors access between sequentially and concurrently scheduled subgraphs can be efficiently handled through a single host-GPU communication and multiple inter/intra-GPU data accesses, rather than communicating them individually between CPU and GPUs. We leverage this observation to develop a communication deduplication method, and we also propose a subgraph reorganization method that enhances the effect of communication deduplication to improve performance.

In summary, we make the following contributions.

- We propose a memory-efficient GNN training framework that reduces runtime memory consumption by integrating a partition-based GNN training method and a recomputation-caching-hybrid intermediate data management method.
- We propose a deduplicated communication framework that reduces host-GPU data communication by optimizing the duplicated neighbor accesses between sequentially and concurrently scheduled subgraphs.
- We develop HongTu, a GPU-accelerated system for full-graph GNN training that overcomes the memory limitation of GPUs and integrates an efficient communication implementation to achieve high performance.

Experimental results on four NVIDIA A100 GPUs show that HongTu reduces host-GPU communication by 38%-78% and achieves 1.3×-3.4× performance improvement over the vanilla approach that transfers the entire neighbor set for each partition. When compared to DistGNN [32] running on 16 CPU nodes, HongTu achieves speedups ranging from 7.8× to 20.2×. Furthermore, for small graphs that can fit into GPUs, HongTu achieves performance comparable to existing multi-GPU systems.

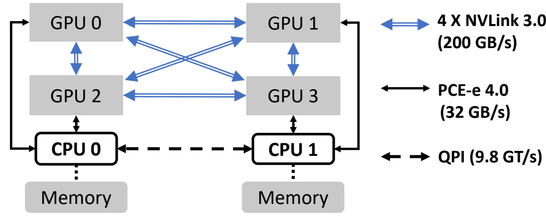


Fig. 1. An example of multi-GPU architecture.

The rest of the work is organized as follows, §2 describes the background and motivations. §3 gives an overview of HongTu. §4 describes the memory-efficient GNN training framework. §5 describes the deduplicated communication framework. §6 describes system implementation. §7 presents results. §8 concludes.

2 BACKGROUND AND MOTIVATIONS

2.1 Multi-GPU Architecture

Modern GPUs are equipped with high bandwidth memory and massive streaming multiprocessors (SMs), making them suitable for memory- and computing-intensive applications. However, the limited device memory capacity, typically ranging from several to tens of gigabytes [38], poses a constraint on the size of data that can be efficiently processed. To address this issue, hardware researchers have developed fast interconnects to connect multiple GPUs, such as AMD Infinity Fabric [2] and NVIDIA NVLink [40]. Figure 1 provides an example of a 4×A100 GPU server, where the four GPUs are interconnected through 4×NVLink-3.0 with 200GB/s inter-GPU communication bandwidth, enabling low latency and high throughput inter-GPU data access. Every two GPUs are connected to a single CPU via PCIe 4.0 interconnect. Although CPUs are generally equipped with hundreds to thousands of gigabytes of host memory, the slow CPU-GPU communication bandwidth (up to 32GB/s in PCIe 4.0) often creates a performance bottleneck for GPU access to CPU memory. Moreover, the two CPUs are linked through a QPI bus, forming a two-socket Non-Uniform Memory Access (NUMA) architecture, where GPUs accessing remote CPU memory via QPI experience slower speeds than those accessing local CPU memory. Therefore, building high-performance multi-GPU applications requires careful optimization of heterogeneous communication, especially in reducing CPU-GPU data transfer.

2.2 GNN Basis

A GNN takes a graph and the vertex-associated property (feature) of all vertices as input and learns a representation vector for each vertex by stacking multiple GNN layers. In each layer, GNN models generally follow an aggregate-update computation pattern.

$$\mathbf{h}_v^l = \mathbf{UPDATE}\Big(\mathbf{AGGREGATE}(\{\mathbf{h}_u^{l-1} | u \in N(v)\}), \mathbf{h}_v^{l-1}\Big), \tag{1}$$

$\mathbf{h}_v^l$ is the representation of v in the l-th layer and $\mathbf{h}_v^0$ is the input vertex feature. The **AGGREGATE** function collects the $l-1$-th layer representations of v's in neighbors, i.e., $\{\mathbf{h}_u^{l-1}) | u \in N(v)\}$, to compute the neighbor representation of v. The **UPDATE** function utilizes the neighbor aggregation result and v's representation in the $l-1$-th layer to calculate v's vertex representation in the l-th layer. Both the aggregate and update functions can be neural networks, which are updated during training. To illustrate, we present two examples: the graph convolutional network (GCN) [21] and the graph attention network (GAT) [52].

GCN is a simple yet effective model that has neural network computation on vertices.

Table 2. Summary of existing full-graph GNN systems. The 'VD' represents the vertex data and the 'ID' represents the intermediate data.

Hardware	Systems	Full Nbr. Agg.	Partially storing VD	Partially storing ID	Major contributions
CPU	DistGNN[32]	✔	-	-	CPU-aggregation optimization& Staleness communication
	Graphite[13]	✔	-	-	Hardware-assisted node property aggregation
GPU	DGL[8]	✔	✗	✗	single-GPU system
	PyG[10]	✔	✗	✗	single-GPU system
	CAGNET[51]	✔	✗	✗	1.5D/2D/3D Graph Partitioning
	DGCL[4]	✔	✗	✗	Cost-based communication routine
	PipeGCN[53]	✗	✗	✗	Staleness-communication&Pipelining
	Sancus[42]	✔	✗	✗	Staleness-communication
	NeuGraph[30]	✗	✔	✗	SAGA abstraction & Partition-based training
	NeutronStar[30]	✗	✔	✗	Hybrid dependency management&Partition-based training
	ROC[18]	✔	✗	✔	Learned graph partitioning cost-based intermediate data management
	HongTu	✔	✔	✔	Recomputation-caching-hybrid intermediate data management Deduplicated communication framework

$$\mathbf{h}_v^l = \sigma(W^l \otimes (\sum_{u \in N(v)} d_{uv} \mathbf{h}_u^{l-1})) \tag{2}$$

The aggregate function is a simple weighted neighbor convolution, where d_{uv} is the normalized edge weight of edge $<u, v>$. The update function involves a linear transformation and a non-linear activation function (e.g., **ReLU**).

GAT introduces a self-attention mechanism, which assigns different attention parameters on edges to distinguish which neighbors are more important.

$$\mathbf{h}_v^l = \sigma\Big(\sum_{u \in N(v)} \frac{exp(\hat{\sigma}(a^l [W^l \mathbf{h}_v^{l-1} || W^l \mathbf{h}_u^{l-1}]))}{\sum_{u \in N(v)} exp(\hat{\sigma}(a^l [W^l \mathbf{h}_v^{l-1} || W^l \mathbf{h}_u^{l-1}]))} \mathbf{h}_u^{l-1} \Big) \tag{3}$$

The aggregate function first concatenates the parameterized representations of source u and destination v, and applies a^l to compute the edge-wise attention coefficient, i.e., $a^l [W^l \mathbf{h}_v^{l-1} || W^l \mathbf{h}_u^{l-1}]$. Then, it feeds the attention coefficients to a **LeakyReLU** activation (i.e., $\hat{\sigma}$) and uses a neighbor-oriented **softmax** function (i.e., $\frac{exp(\cdot)}{\sum_{u \in N(v)} exp(\cdot)}$) to compute the edge weight for the neighbor aggregation. The update function is usually a simple non-linear activation (e.g., **ReLU**).

2.3 Full-Graph GNN Training

Full-graph GNN training uses the full-neighbor aggregation semantic and global gradient descent algorithm. It runs epochs repeatedly on the entire graph until reaching the target accuracy or epoch. Each training epoch consists of a forward and a backward pass, followed by parameter update, which uses the gradients computed in the backward pass to update the trainable parameters in every layer. In the forward pass, vertex representations are computed layer-by-layer using the **AGGREGATE** and **UPDATE** operations presented in Section 2.2. At every layer, each vertex aggregates the representations of the incoming neighbors and calculates the vertex representation by applying the learnable model parameters. The final layer's vertex representations are then sent to the downstream task where the loss value is calculated based on the ground truth labels. In the backward pass, GNN's computation starts from the last layer and proceeds back to the first layer, calculating the gradient of loss with respect to the model across all layers. In each layer, the gradients of vertex representations are computed using the chain rule, facilitating both intra-layer model gradient calculation and cross-layer gradient transmission [50, 56].

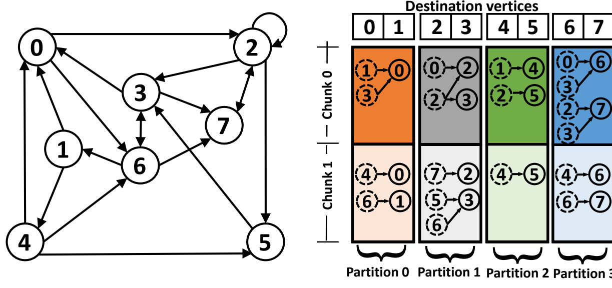


Fig. 2. 2D graph partitioning on a 8 vertices toy graph. Each partitioned subgraph is represented by a colored box. Solid and dashed circles denote the master and mirror vertices, respectively.

GNN models are distinct from traditional DNN models because the link relationship between vertices creates complex and non-uniform data dependencies. In a CNN, the convolution kernels are fixed and treat all pixels in the same way. However, in a GCN, the **AGGREGATE** and its backward operation handle data dependencies by gathering data along edges. This not only entails random data accesses but also introduces complexities in workload partition due to its irregular nature. Generally, achieving efficient vertex data access necessitates accommodating all vertex data within GPU memory. In GNN training, intermediate data generated in the forward pass needs to be reserved for gradient computation in the backward pass. For example, the update function in GCN [21] involves `linear+Relu` computations in the forward pass, i.e., $\mathbf{h} = ReLU(\mathbf{a} \times W)$. Its backward pass computes the gradients of parameter W using the formula: $\nabla W{=}(\mathbf{a})^t \times ReLU^{-1}(\mathbf{a} \times W) * \nabla \mathbf{h}$. Here, $ReLU^{-1}$ is the derivative function of $ReLU$, which returns 1 for positive inputs and 0 otherwise. $(\cdot)^t$ represents the transpose operation. $\mathbf{a} \times W$ is the intermediate data that needs to be reserved for gradient computation. While the effectiveness and high accuracy of full-graph GNN training have been widely demonstrated by academic studies [18, 21, 30, 50], its practical application in industry is limited due to the significant memory requirements for maintaining large-scale vertex and intermediate data.

2.4 Existing Systems and limitations

Table 2 summarizes existing full-graph GNN systems and their major contributions. Early systems, such as DGL [55] and PyG [10] use full-graph training on a single GPU, and thus their efficiency and scalability are constrained by the limited GPU resource. To meet the high computation and memory requirements of full-graph GNN training, distributed- and multi-GPU-based systems have been proposed. CAGNET [51], DGCL [4], PipeGCN[53], and Sancus [42] are four multi-GPU GNN systems that maintain both vertex and intermediate data in GPU memory. In these systems, inter-GPU communication emerges as a performance bottleneck [4, 42]. CAGNET [51] proposes 1.5D, 2D, and 3D graph partitioning to optimize the data distribution among GPUs. DGCL [4] analyses the speeds of heterogeneous communication among devices and proposes an automatic routine algorithm to improve communication efficiency. PipeGCN [53] and Sancus [42] investigate staleness-communication in GNN training, which reduces communication times while sustaining a reasonable level of accuracy. Despite the high performance of these frameworks, they can hardly scale to large input graphs. As illustrated in Table 1, accommodating the data for training ogbn-paper graph needs at least 77 NVIDIA A100 GPUs (80GB), which is expensive and requires sophisticated design for managing communication and fault-tolerance on a distributed GPU cluster. Moreover, the relatively slow inter-node communication can also become a critical performance bottleneck [4].

Recently, some research work [18, 30, 56] try to relax the memory constraint of GPU by partially loading vertex or intermediate data during training. As illustrated in Table 2, NeuGraph [30] and

NeutronStar [56] employ 2-D graph partitioning to split a large graph into multiple chunks, where each chunk contains a specific range of destination and source vertices (as shown in Figure 2). During training, the two frameworks store the vertex data in the CPU and sequentially load the vertex data of partitioned chunks to the GPU for training. On the other hand, ROC [18] utilize CPU-memory to manage the intermediate data. ROC includes a cost model to represent the host-GPU data transfer overhead, and utilizes dynamic programming to find the optimal communication plan. By doing so, ROC allows the GPU to store only part of the intermediate data.

In addition to GPU-based systems, researchers have also explored distributed CPU-based systems [13, 32] to leverage the large memory capacity of CPU platforms. However, these systems generally exhibit inferior performance when compared to GPU-based solutions, and the monetary cost of using high-end CPU clusters is also high. Therefore, building a CPU-GPU heterogeneous system that fully utilizes the memory and computation resources of a single-node-multi-GPU architecture becomes a cost-effective option.

However, we observe that the existing out-of-GPU-memory processing systems still face two limitations that hinder their effectiveness and efficiency in handling large-scale GNN training.

Limitation 1: Existing systems still suffer from the high memory consumption of either vertex or intermediate data. While NeuGraph [30] and NeutronStar [56] decrease the memory consumption of vertex data, they still require intermediate data to be stored entirely in the GPU. Conversely, ROC reduces the memory consumption of intermediate data, but still necessitates completed storage of vertex data in the GPU. More importantly, several critical limitations hinder the direct combination of these memory reduction methods. Firstly, the 2D partitioning in NeuGraph and NeutronStar separates a vertex's neighbors into multiple slices, making implementing full-neighbor aggregation challenging for complex GNNs like GAT [52] model, which involves a `softmax()` computation on the entire neighbor set. In these workloads, loading all neighbor-containing partitions is still necessary. This renders existing systems ineffective on training large-scale GAT-like models. Secondly, ROC's caching-based method is inefficient on complex GNNs with large-scale intermediate data [25, 26, 52], as swapping large-scale intermediate data significantly increases host-GPU communication. Moreover, since the intermediate data are swapped at a whole-graph granularity, ROC's approach may fail if a single intermediate tensor is excessively large.

Table 3. Neighbor replication factor α under different partitions.

Partitions	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
it-2004	1.23	1.35	1.46	1.52	1.60	1.63	1.71	1.76	1.85
ogbn-paper	1.25	1.52	2.13	3.02	4.46	6.34	8.50	10.6	12.3
friendster	1.32	1.77	2.68	3.86	5.48	7.70	10.70	14.4	18.1

Limitation 2: Existing systems suffer from increased host-GPU communication caused by neighbor replication. When a graph is partitioned into multiple subsets, vertices with multiple outgoing edges are replicated across partitions to serve as incoming neighbors for remote neighbor aggregation. These vertices, which we refer to as duplicate neighbors, need to be transferred individually among partitions during computation, leading to increased host-GPU communication. The communication volume is quantified by the neighbor replication factor α, which is defined as the average number of replicas per vertex. Compared to the ideal case where each vertex is transferred only once, transferring neighbor data for each chunk individually results in α times communication volume, which makes the PCIe-based host-GPU communication a critical performance bottleneck. Table 3 presents the replication factor of the three large graphs used in our evaluation, split from 2 to 512 partitions. We can observe that increasing the number of partitions leads to an increase in data transfer volume. However, existing partition-based systems do not account for this factor. NeuGraph

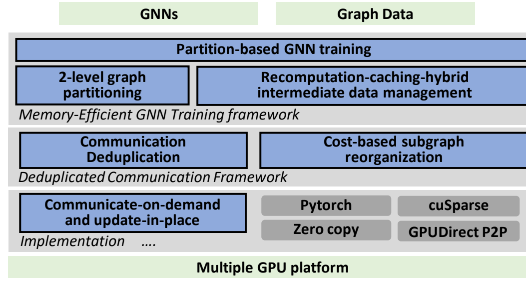


Fig. 3. HongTu system overview.

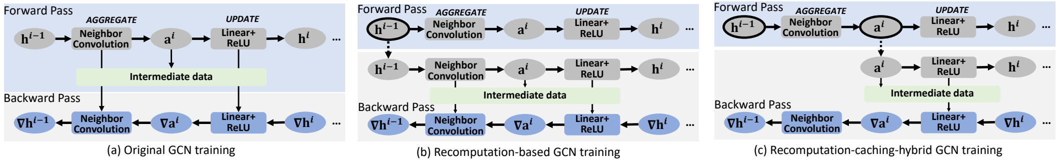


Fig. 4. A Graphical illustration of the original, recomputation-based, and recomputation-caching-hybrid training methods on the GCN model. Here we give the example of a single layer, while other layers have the same calculation mode. A box represents an operation, a circle represents a tensor, and a tensor surrounded by a black frame indicates that the it needs to be cached in the CPU memory (checkpoint). Solid arrows indicate dependencies between tensors and operations and dash arrows indicate host-GPU communication.

and NeutronStar [30, 56] use host-side filters to remove unnecessary data before communicating a partition, but still need to transfer the neighbor data entirely for each of them.

3 THE HongTu FRAMEWORK

We present HongTu, a GPU-accelerated full-graph GNN system that addresses the limitations outlined in Section 2.4 through two critical system components. First, HongTu provides a memory-efficient training framework that reduces the memory consumption of both vertex data and intermediate data. Second, HongTu provides a deduplicated communication framework that effectively reduces host-GPU communication for duplicated neighbor access among subgraphs. Figure 3 provides an architectural overview of HongTu.

Memory-efficient GNN training framework. HongTu adopts a graph partitioning method that groups edges incident on the same destination into a single chunk. This design facilitates full neighbor aggregation on each chunk individually, enabling HongTu to support complex GNNs (such as GAT [52] and GGCN [25]) efficiently while reducing memory usage. Moreover, HongTu extends the recomputation-based DNN training method [5] to GNN training, which avoids storing intermediate data by recomputing it in the backward pass. Taking the advantages that some graph operations involve only simple edge computation and do not generate intermediate data, we hybrid GPU-based recomputation and CPU-based data caching to reduce the additional processing overhead.

Deduplicated communication framework. We observe that the duplicated data access between concurrently scheduled subgraphs on multiple GPUs and the duplicated data access between sequentially scheduled subgraphs on the same GPU can benefit from inter-GPU and intra-GPU data communication, both of which have higher speeds compared to PCIe-based host-GPU communication. We propose a deduplicated communication method that transfers the data of each duplicated neighbor only once between CPU and GPU, and converts redundant host-GPU communication

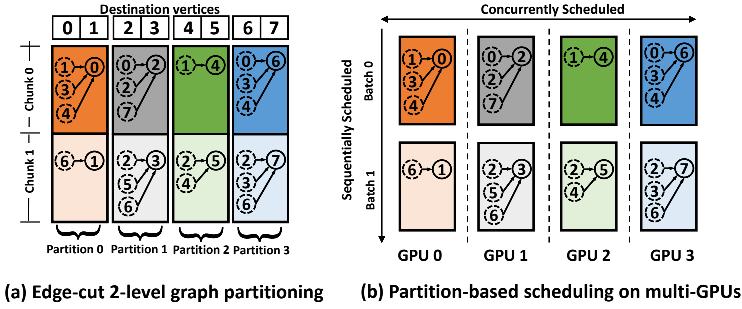


Fig. 5. An illustration of 2-level graph partitioning and the task scheduling on multiple GPUs.

into more efficient inter-GPU communication or intra-GPU data reuse. Moreover, considering the impact of vertex distribution on communication deduplication effectiveness, HongTu incorporates a cost-model guided subgraph reorganization method to minimize communication overhead.

4 MEMORY-EFFICIENT GNN TRAINING FRAMEWORK

4.1 Edge-Cut 2-Level Graph Partitioning

HongTu employs an edge-cut partitioning to split the graph into small execution units suitable for processing by a single GPU, as shown in Figure 5 (a). Initially, the input graph is split into m (the number of GPUs) partitions through Metis partitioning [20] to improve load balancing and group closely linked vertices into one partition. Each partition is subsequently divided into computation-balanced chunks through range-based partitioning [65], with each chunk containing a unique set of destination vertices and their associated edges. This partitioning method enables the full-neighbor aggregation to be implemented on each chunk individually. It is worth mentioning that only the in-edges of destinations need to be grouped, as the complex aggregations are executed only in the forward pass. In backward propagation, source vertices accumulate the gradient along the out-edges through summation. Leveraging the associativity of the sum operation, multiple source replicas in different chunks can independently calculate gradients and subsequently aggregate them. During GNN training, partitioned subgraphs are scheduled in a fixed order as shown in Figure 5 (b). Chunks belong to the same partition are sequentially scheduled on one GPU, and chunks with the same local position from different partitions are scheduled concurrently on different GPUs. For brevity, we use the term *batch* to refer to a group of concurrently scheduled chunks from different partitions.

4.2 Recomputation-Caching-Hybrid Intermediate Data Management

To reduce the memory consumption of DNN training, researchers have proposed a recomputation-based strategy that eliminates the need to store intermediate data for every layer by recalculating an additional forward pass in the backward computation [5]. However, this method is designed for DNN training and assumes the training data of all layers can be entirely stored in GPU memory as the checkpoint. This makes it unsuitable for full-graph GNN training, where the training data of the entire graph can occupy a significant amount of memory. We generalize the recomputation-based approach to the CPU-GPU heterogeneous platform. Figure 4 (b) shows a graphical illustration of the recomputation-based method on a single layer of GCN training, where the **AGGREGATE** operation is the neighbor convolution and the **UPDATE** operation is the `Linear+ReLU` calculation. In the forward computation of each GNN layer, HongTu copies the output representations to CPU memory as checkpoint and releases the intermediate data to make room for training the next *batch*.

In the backward pass, HongTu loads the checkpoint from CPU, recomputes the forward pass, and computes the gradients based on the regenerated intermediate data. This method allows HongTu to store the training data of only one layer, thereby reducing the overall GPU memory consumption. Importantly, the recomputation-based approach maintains the accuracy of the original training method [5] as shown in Figure 4 (a), because the regenerated intermediate data are identical to that produced in the forward computation.

Recomputation-based training reduces memory consumption but entails an additional forward pass. However, not all recomputation is necessary. In the case of GNNs with simple arithmetic edge computation, where the **AGGREGATE** operation does not yield intermediate results required for gradient computation, caching the output of the **AGGREGATE** operation in CPU can eliminate the need for recomputation. For instance, the GCN model [21] in Equation 2 employs a weighted neighbor summation as the **AGGREGATE** operation. Recomputing the **AGGREGATE** requires loading representations of all neighbors from the CPU and redoing neighbor convolution on GPUs, resulting in $O(\alpha|V|)$ CPU-GPU communication and $O(|E|)$ GPU computation. Alternatively, caching the output neighbor representations of **AGGREGATE** in CPU memory and transferring them back when needed achieves the same functionality with only $O(|V|)$ host-GPU communication. Based on this observation, we propose a recomputation-caching-hybrid method shown in Figure 4 (c). In the forward pass, HongTu caches the neighbor representation ($\mathbf{a}^i$) in the CPU as the recomputation checkpoint. In the backward pass, HongTu skips the **AGGREGATE** step, loads the cached neighbor representations from the CPU, and recomputes only the **UPDATE** stage. This hybrid design can benefit a broad range of popularly used GNNs, such as GCN [21], GraphSage [14], GIN [59], and CommNet [49]. However, for GNNs with neural network computation on edges (e.g., GAT [52] and GGCN [25]), the overhead of caching the $O(|E|)$ intermediate data can be higher than that of recomputation. In such cases, HongTu falls back to the recomputation-based method as depicted in Figure 4 (b).

4.3 Overall Execution Flow in HongTu

Algorithm 1 outlines the overall execution flow. To begin with, HongTu partitions the graph with 2-level partitioning (line 1). Each subgraph, represented by G_{ij}, consists of a set of disjointly split vertices V_{ij} and their incoming edges E_{ij}, where i is the partition id and j is the chunk id. After graph partitioning, the communication deduplication module reorganizes the partitioned subgraphs, deduplicates the neighbor accesses, and generates the new partitions $\{\mathcal{G}_{ij}|0\leq i<m, 0\leq j<n\}$ for parallel training (line 2). Since the initial partitioned graph G_{ij} is no longer used in the subsequent computation, we use V_{ij} and E_{ij} to represent the vertices and edges of the new subgraph $\mathcal{G}_{ij}$ and denotes its in-neighbor set by N_{ij}. After preprocessing, HongTu initializes the vertex representation buffer $\mathbf{h}^l$ and gradient buffer $\nabla\mathbf{h}^l$ in the CPU memory (line 3).

In the training process, *batches* are scheduled sequentially, and subgraphs in each *batch* are processed in parallel. In the forward pass of each *batch*, neighbor representations of all subgraphs, i.e., $\{\mathbf{h}^l_{N_{ij}}|0\leq i\leq m\}$ are first loaded from CPU to GPUs through the deduplicated communication framework (line 6) which will be discussed in Section 5. Following this, each GPU performs the forward computation (lines 7-8), copies the newly computed vertex representation $\mathbf{h}^{l+1}_{V_{ij}}$ to CPU (line 9), and releases intermediate data to make room for training the next *batch*. After completing the forward pass, the downstream task takes the final layer output $\mathbf{h}^L$ as input, computes the `loss` and the gradient of `loss` to the final layer representation, i.e., $\nabla\mathbf{h}^L$ (lines 10-11). In the backward pass, computation is scheduled from the last layer to the first layer (line 12). In each *batch*, HongTu reloads the checkpoint to GPUs (line 14), loads the gradients of destinations from CPU memory (line 16), recomputes the forward pass of the current layer (line 17), and computes the gradients (line

Algorithm 1 Workflow of HongTu for a single epoch

Input: Graph $G(V, E)$, feature $\{\mathbf{h}_v^0 \mid v \in V\}$, Initial parameterized GNN layers $\{\mathbf{GNN}^0, \mathbf{GNN}^1 \ldots \mathbf{GNN}^{L-1}\}$
Output: Updated parameterized GNN layers $\{\mathbf{GNN}^0, \mathbf{GNN}^1 \ldots \mathbf{GNN}^{L-1}\}$

1: $\{G_{ij}|0{\le}i{<}m, 0{\le}j{<}n\}$=**two_level_partition**$(G, m, n)$
2: $\{\mathcal{G}_{ij}|0{\le}i{<}m, 0{\le}j{<}n\}$= **deduplication**$(\{G_{ij}|0{\le}i{<}m, 0{\le}j{<}n\})$
3: allocate $\{\mathbf{h}^l, \nabla\mathbf{h}^l \mid 0 \le l \le L\}$ in the CPU memory.
4: **for** layer $l = 0$ to $L-1$ **do**
5: **for** *batch* with id $j = 0$ to $n-1$ **do**
6: $\{\mathbf{h}^l_{N_{ij}}|0{\le}i{<}m\} \leftarrow$ **dedup_comm_fwd**$(\mathbf{h}^l, \{\mathcal{G}_{ij}|0{\le}i{<}m\}$, HtoD)
7: **for** GPU $i = 0$ to $m-1$ **do in parallel**
8: $\mathbf{h}^{l+1}_{V_{ij}}$ = GPU(i).forward($\mathbf{GNN}^i, \mathbf{h}^l_{N_{ij}}, \mathcal{G}_{ij}$)
9: $\mathbf{h}^{l+1} \leftarrow$ GPU(i).mem_copy($\mathbf{h}^{l+1}_{V_{ij}}, V_{ij}$, DtoH)
10: **loss**= downstream_task($\mathbf{h}^L$)
11: $\nabla\mathbf{h}^L$ =**loss**.backward()
12: **for** layer $l = L-1$ to 0 **do**
13: **for** *batch* with id $j = 0$ to $n-1$ **do**
14: $\{\mathbf{chkpt}^l_{ij}|0{\le}i{<}m\} \leftarrow$**load_recomp_chkpt**(. . ., HtoD)
15: **for** GPU $i = 0, 1, \ldots, m-1$ **do in parallel**
16: $\nabla\mathbf{h}^{l+1}_{V_{ij}} \leftarrow$ GPU(i).mem_copy($\nabla\mathbf{h}^{l+1}, V_{ij}$, HtoD)
17: GPU(i).reforward($\mathbf{GNN}^l$, $\mathbf{chkpt}^l_{ij}$ $\mathcal{G}_{ij}$)
18: $\nabla\mathbf{h}^l_{N_{ij}}$ = GPU(i).backward($\mathbf{GNN}^l, \nabla\mathbf{h}^{l+1}_{V_{ij}}, \mathcal{G}_{ij}$)
19: $\nabla\mathbf{h}^l \overset{\oplus}{\leftarrow}$**dedup_comm_bwd**$(\{\nabla\mathbf{h}^l_{N_{ij}}, \mathcal{G}_{ij}|0{\le}i{<}m\}$,DtoH)
20: **for** layer $l = 0$ to $L-1$ **do**
21: sync_and_update ($\mathbf{GNN}^l$) //parameter update

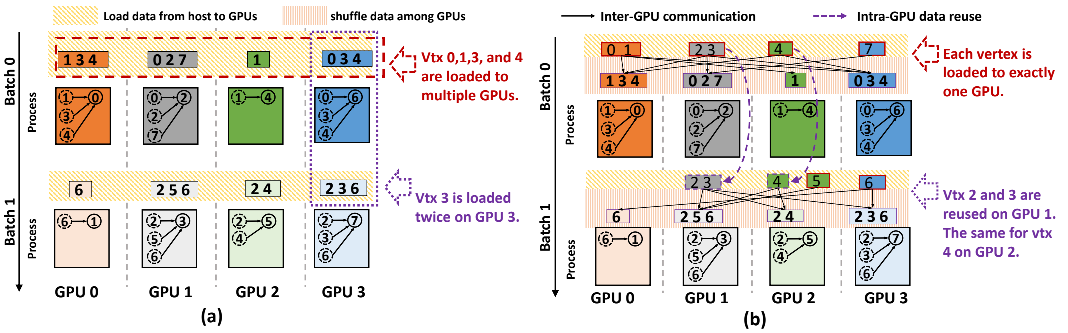


Fig. 6. Graphical illustrations of (a) increased host-GPU communication caused by duplicated neighbor accesses and (b) the proposed communication deduplication method.

18) based on the regenerated intermediate data. Finally, the neighbor gradients $\{\nabla\mathbf{h}^l_{N_{ij}} \mid 0{\le}i{\le}m\}$ are transferred back to CPU and accumulated into the gradient buffer $\nabla\mathbf{h}^l$ through deduplicated communication (line 19).

After completing a forward-backward pass, HongTu updates the model parameters according to the gradients (line 19). Since the model parameters in GNN are often small, HongTu replicates them among GPUs and uses `all_reduce()` function to synchronize.

The Effectiveness of memory reduction. By combining 2-level graph partitioning and recomputation-based training, HongTu can maintain the training data of one GNN layer for a single subgraph in each GPU. In an ideal case where the graph is evenly partitioned, the vertex data volume of one subgraph can be formalized as $(1 + \alpha_{m*n})|V|/(m*n)$, where $m*n$ is the number of subgraphs, $|V|$ is the number of vertices, and $\alpha_{(m*n)}$ is the neighbor replication factor given $m*n$. As shown in Table 3, every doubling of the number of partitions results in a 47%, 35%, and 32% reduction in the memory consumption of vertex data for the three graphs, respectively. The memory consumption of intermediate data varies, depending on the GNN model in use. Some models are dominated by the number of vertices [14, 21, 49, 59], while others are dominated by the number of edges [25, 26, 52], and both decrease linearly as $m*n$ increases. In practical training, memory consumption can be adjusted by tuning the number of partitions to adapt to different GPUs.

5 DEDUPLICATED COMMUNICATION FRAMEWORK

In this section, we present the design and implementation of deduplicated communication framework.

5.1 Basic Design

Inter-GPU duplicated neighbor access. Duplicated neighbors between concurrently-scheduled subgraphs cause the same vertex to be transferred to multiple GPUs. As indicated by the red dashed box in Figure 6 (a), vertex 0, 1, 3, and 4 are transferred to multiple GPUs in *batch* 0. The data of these vertices are redundantly communicated between CPU and GPUs. Instead, we can transfer the duplicated vertex to one GPU and handle the access requests from other GPUs through inter-GPU communication. Benefiting from the high communication bandwidth between GPUs (as described in Section 2), converting host-GPU communication to inter-GPU communication can significantly improve performance.

Intra-GPU duplicated neighbor access. Duplicated neighbors between sequentially-scheduled subgraphs cause the same vertex to be transferred multiple times to the same GPU. As indicated by the purple dotted boxes in Figure 6 (a), vertex 2 and 5 in GPU 1 and vertex 3 in GPU 3 are loaded in both batch 0 and batch 1. For those adjacently-scheduled subgraphs, neighbor access to duplicated neighbors from the successor subgraph can directly reuse the already transferred data in GPU, converting host-GPU communication to intra-GPU data access.

Communication Deduplication. We stack these two techniques to cooperatively reduce the communication for duplicated neighbor accesses, as illustrated in Figure 6 (b). Our method involves two steps. In the first step, it computes the union of neighbors in each *batch* (i.e., a group of concurrently scheduled subgraphs). This union is then deduplicated and stored in a transition vertex set denoted by $\mathbb{N}_j^{\cup} = \cup_{i=0}^{m} N_{ij}$, where $0 \leq j < m$ indicates the batch id. During computation, each vertex in the transition vertex set is transferred to exactly one GPU and shared among GPUs through inter-GPU communication. Figure 6 (b) provides a graphical example. The deduplicated vertex sets, $\mathbb{N}_0^{\cup}$, i.e., $\{0, 1, 2, 3, 4, 7\}$, and $\mathbb{N}_1^{\cup}$, i.e., $\{2, 3, 4, 5, 6\}$ are transferred only once, reducing host-GPU communication times from 19 to 11. To share communication workload among GPUs, we divide $\mathbb{N}_j^{\cup}$ into m subsets $\{\mathbb{N}_{0j}, \dots \mathbb{N}_{m-1j}\}$, and assign the communication of $\mathbb{N}_{ij}$ to GPU i, where $\mathbb{N}_{ij}$ is the subset of $\mathbb{N}_j^{\cup}$ belonging to partition i, as shown in Figure 6 (b). In the second step, we perform the intra-GPU deduplication on the transition vertex set for each pair of adjacently-scheduled subgraphs, e.g., $\mathbb{N}_{ij-1}$ and $\mathbb{N}_{ij}$. We divide the successor transition vertex set $\mathbb{N}_{ij}$ into two disjoint subsets $\mathbb{N}_{ij}^{gpu}$ and $\mathbb{N}_{ij}^{cpu}$, where $\mathbb{N}_{ij}^{gpu}$ represents the duplicated vertices, i.e., $\mathbb{N}_{ij} \cap \mathbb{N}_{ij-1}$, and $\mathbb{N}_{ij}^{cpu}$ represents the remaining vertices $\mathbb{N}_{ij} \setminus \mathbb{N}_{ij-1}$. When loading the data of $\mathbb{N}_{ij}$ from the CPU, vertices in $\mathbb{N}_{ij}^{gpu}$ are directly reused from the GPU, and vertices in $\mathbb{N}_{ij}^{cpu}$ are loaded from the CPU memory.

Figure 6 (b) shows a graphical illustration with purple dashed arrows. When processing batch 1, the data of $\mathbb{N}_{11}^{gpu}$ ($\{2,3\}$) and $\mathbb{N}_{12}^{gpu}$ ($\{4\}$) are directly reused from *batch* 0, and the data of $\mathbb{N}_{12}^{cpu}$ ($\{5\}$) and $\mathbb{N}_{13}^{cpu}$ ($\{6\}$) are loaded from CPU memory. This step further reduces host-GPU communication times from 11 to 8.

5.2 Workflow of Deduplicated Communication

HongTu uses a transition data buffer $\mathbf{h}_{\mathbb{N}_{ij}}$ on each GPU to manage the data of transition vertices $\mathbb{N}_{ij}$, based on which we can decouple host-GPU communication and inter-GPU communication.

Algorithm 2 `dedup_comm_fwd`

Input: $\{N_{ij}, \mathbb{N}_{ij}, \mathbb{N}_{ij}^{gpu}, \mathbb{N}_{ij}^{cpu} \mid 0 \le i < m\}$, $\mathbf{h}^l$ in the CPU.
Output: load $\{\mathbf{h}_{N_{ij}} \mid 0 \le i < m\}$ to the m GPUs separately
1: **for** GPU $i = 0$ to $m-1$ **do in parallel** //host-to-GPU
2: $\quad \mathbf{h}_{\mathbb{N}_{ij}} \leftarrow$ `GPU(`i`).reuse(`$\mathbf{h}_{\mathbb{N}_{ij-1}}, \mathbb{N}_{ij}^{gpu}$`)`
3: $\quad \mathbf{h}^l_{\mathbb{N}_{ij}} \leftarrow$ `GPU(`i`).mem_copy_sparse(`$\mathbf{h}^l, \mathbb{N}_{ij}^{cpu}$`, HtoD)`
4: `synchronize()`
5: **for** GPU $i = 0$ to $m-1$ **do in parallel** //GPU-to-GPU
6: $\quad$ **for** GPU $k = i+1$ to $(m+i)$ mod m **do**
7: $\quad\quad \mathbf{h}_{N_{ij}} \leftarrow$ `GPU(`i`).fetch_from_gpu(`$k, \mathbf{h}_{\mathbb{N}_{ik}}, N_{ij} \cap \mathbb{N}_{ik}$`, DtoD)`
8: `synchronize()`

Algorithm 2 outlines the workflow of deduplicated communication in the forward pass. It loads the neighbor representations $\mathbf{h}^l_{N_{ij}}$ from the CPU data buffer $\mathbf{h}^l$ to the m GPUs separately, based on four distinct vertex sets: $\{N_{ij}, \mathbb{N}_{ij}, \mathbb{N}_{ij}^{gpu}, \mathbb{N}_{ij}^{cpu} | 0 \le i < m\}$. In the first step, each GPU i loads the data of transition vertices to the transition data buffer $\mathbf{h}_{\mathbb{N}_{ij}}$, by reusing the data of $\mathbb{N}_{ij}^{gpu}$ from $\mathbf{h}_{\mathbb{N}_{ij-1}}$ and loading the data of $\mathbb{N}_{ij}^{cpu}$ from the CPU (lines 2-3). In the second step, GPUs communicate with each other to fetch the data of each $N_{ij} \cap \mathbb{N}_{ik}$ from remote transition data buffers and assemble the neighbor data $\mathbf{h}^l_{N_{ij}}$ in local memory (lines 5-7).

Algorithm 3 `dedup_comm_bwd`

Input: $\{N_{ij}, \mathbb{N}_{ij}, \mathbb{N}_{ij}^{gpu}, \mathbb{N}_{ij}^{cpu} | 0 \le i < m\}$, $\{\nabla \mathbf{h}^l_{N_{ij}} | 0 \le i < m\}$ on the m GPUs
Output: Accumulate $\{\mathbf{h}_{N_{ij}} | 0 \le i < m\}$ to the CPU gradient buffer $\nabla \mathbf{h}^l$.
1: **for** GPU $i = 0$ to $m-1$ **do in parallel**
2: $\quad$ **for** GPU $k = i+1$ to $(m+i)$ mod m **do**
3: $\quad\quad \nabla \mathbf{h}_{\mathbb{N}_{ik}} \overset{\oplus}{\leftarrow}$ `GPU(`i`).accum_to_gpu(`$k, \nabla \mathbf{h}^l_{N_{ij}}, N_{ij} \cap \mathbb{N}_{ik}$`, DtoD)`
4: `synchronize()`
5: **for** GPU $i = 0$ to $m-1$ **do in parallel**
6: $\quad \nabla \hat{\mathbf{h}}_{\mathbb{N}_{ij}^{cpu}} \leftarrow$ `GPU(`i`).mem_copy_sparse(`$\nabla \mathbf{h}_{\mathbb{N}_{ij}}, \mathbb{N}_{ij}^{cpu}$`, DtoH)`
7: $\quad \nabla \mathbf{h}^l \overset{\oplus}{\leftarrow} \nabla \hat{\mathbf{h}}_{\mathbb{N}_{ij}^{cpu}}$ //CPU computation
8: `synchronize()`

In the backward pass, deduplicated communication function copies the neighbor gradients $\nabla \mathbf{h}^l_{N_{ij}}$ back to the CPU and accumulates them to the gradient buffer $\nabla \mathbf{h}^l$. Algorithm 3 outlines this process. In the first step, GPUs accumulate the neighbor gradients $\nabla \mathbf{h}_{N_{ij}}$ back to the gradient buffer of transition vertices (lines 1-3). Subsequently, each GPU moves the gradients of $\mathbb{N}_{ij}^{cpu}$ out to CPU memory (line 6) and reserves the gradient of $\mathbb{N}_{ij}^{gpu}$ in GPU to accumulate the gradients of the next

batch. One the CPU side, the gradients of $\mathbb{N}_{ij}^{cpu}$ are accumulated to $\nabla\mathbf{h}_i^l$ with CPUs (line 7). Since the gradient accumulation only involves simple arithmetic addition, utilizing CPUs is faster than copying the data to the GPU computation, in which the data movement involves bidirectional host-GPU communication.

5.3 Cost-Effective Subgraph Reorganization

The effectiveness of communication deduplication is affected by the distribution of duplicated neighbors. To enhance communication efficiency, we quantify the deduplicated communication overhead and propose a subgraph reorganization method to minimize it.

Cost of deduplicated communication. Initially, the neighbor set of each subgraph is transferred entirely. The host-GPU communication has a volume of $\mathbf{V}_{ori} = \sum_{j=0}^{n}\sum_{i=0}^{m}|N_{ij}|$, where $|N_{ij}|$ represents the number of neighbors of partition i chunk j. By utilizing inter-GPU communication duplication, the host-GPU communication volume is reduced to $\mathbf{V}_{+p2p} = \sum_{j=0}^{n}|\cup_{i=0}^{m} N_{ij}|$, where $\cup_{i=0}^{m}N_{ij}$ is the transition vertex set of batch j. By further applying intra-GPU communication duplication, the duplicated transition vertices of each pair of adjacent subgraphs, i.e., $\cup_{i=0}^{m}N_{ij} \cap \cup_{i=0}^{m}N_{ij-1}$, are no longer required to be transferred. Consequently, the host-GPU communication is further reduced to $\mathbf{V}_{+ru} = |\cup_{i=0}^{m} N_{i0}| + \sum_{j=1}^{n}|\cup_{i=0}^{m} N_{ij} \setminus \cup_{i=0}^{m}N_{ij-1}|$. Finally, the total communication overhead can be formalized as

$$\mathbf{C} = \mathbf{V}_{+ru}/\mathbf{T}_{hd} + (\mathbf{V}_{ori} - \mathbf{V}_{+p2p})/\mathbf{T}_{dd} + (\mathbf{V}_{+p2p} - \mathbf{V}_{+ru})/\mathbf{T}_{ru}, \quad (4)$$

where $\mathbf{T}_{hd}$, $\mathbf{T}_{dd}$, and $\mathbf{T}_{ru}$ represent the throughput of host-GPU communication, inter-GPU communication, and intra-GPU data reusing, respectively. These parameters are environment-specific and depend on the used GPU platform.

We can observe that the communication cost $\mathbf{C}$ is affected by the number of duplicated neighbors among subgraphs. Obtaining the minimal $\mathbf{C}$ requires careful adjustments of vertex distribution in the partitions. However, optimizing this goal in the partitioning stage is challenging because it involves a vast search space at the vertex granularity and couples with several constraints, such as load balancing and communication reduction. To simplify this problem, we propose a subgraph granularity optimization approach. Specifically, given an initialized load-balancing optimized graph partition $\{G_{ij} \mid 0 \le i < m, 0 \le j < n\}$, the objective is to find a reorganized partition $\{\mathcal{G}_{ij} \mid 0 \le i < m, 0 \le j < n\}$ that minimizes the cost $\mathbf{C}$ in Equation 4, where each $\mathcal{G}_{ij}$ is a subgraph from the initial partition, e.g., G_{kl}. This combinatorial optimization problem is NP-hard as it can be reduced to a variant of the classical NP-hard traveling salesman problem [19], which aims to find a Hamiltonian circuit in a weighted undirected complete graph that minimizes the total weight of the circuit. Therefore, it is infeasible to obtain an optimal solution in polynomial time using an exact algorithm. Next, we propose a 2-phase heuristic to reorganize the partition.

Partition reorganization. We propose a 2-phase, greedy-based heuristic that optimizes communication overhead by maximizing the number of inter- and intra-GPU duplicated neighbors. The goal is to fully leverage the effect of communication deduplication. Algorithm 4 outlines the workflow of our approach. In the first phase, we reorganize subgraphs within each partition to group subgraphs with the maximum number of duplicate neighbors into the same *batch*. The objective is to maximize the number of inter-GPU duplicated neighbors while preserving the locality achieved by the Metis graph partitioning. The algorithm initializes the intermediate partition G^t and the transition vertex set $\mathbb{N}^{\cup}$ for every *batch* with subgraphs in partition 0 (lines 1-2), and then reorganizes other partitions in turn. Specifically, it iterates over the transition vertex set of all *batches* (line 5) and retrieves for each *batch* the subgraph that has the maximum number of duplicate neighbors in the currently-processed partition (line 6). The algorithm then writes the found subgraph to the

Algorithm 4 Partition reorganization

Input: initial partitions $\{G_{ij} \mid 0 \le i < m, 0 \le j < n\}$
Output: reorganized partitions $\{\mathcal{G}_{ij} \mid 0 \le i < m, 0 \le j < n\}$

Phase 1: Reorganization for maximizing inter-GPU duplication

1: **for** $j = 0$ to $n-1$ **do**
2: $\quad G^t_{0j} \leftarrow G_{0j}$; $\mathbb{N}^{\cup}_j \leftarrow N_{0j}$
3: **for** $i = 1$ to $m-1$ **do**
4: $\quad \mathcal{K} \leftarrow \{0, 1, \ldots, n-1\}$ //subgraphs that have not been processed
5: $\quad$ **for** $j = 0$ to $n-1$ **do**
6: $\quad\quad$ `find` k `from` $\mathcal{K}$, `s.t.,` $\forall a \in \mathcal{K} : |N_{ik} \cap \mathbb{N}^{\cup}_j| \ge |N_{ia} \cap \mathbb{N}^{\cup}_j|$
7: $\quad\quad G^t_{ij} \leftarrow G_{ik}$; $\mathbb{N}^{\cup}_j \leftarrow \mathbb{N}^{\cup}_j \cup N_{ik}$; $\mathcal{K} \leftarrow \mathcal{K} \setminus k$

Phase 2: Reorganization for maximizing intra-GPU duplication

8: **for** $i = 0$ to $m-1$ **do**
9: $\quad \mathcal{G}_{i0} \leftarrow G^t_{i0}$
10: $\mathcal{K} \leftarrow \{1, 2, \ldots, n-1\}$ //*batches* that have not been processed
11: **for** $j = 1$ to $n-1$ **do**
12: $\quad$ `find` k `from` $\mathcal{K}$, `s.t.,` $\forall a \in \mathcal{K} : |\mathbb{N}^{\cup}_k \cap \mathbb{N}^{\cup}_{j-1}| \ge |\mathbb{N}^{\cup}_a \cap \mathbb{N}^{\cup}_{j-1}|$
13: $\quad$ **for** $i = 1$ to $m-1$ **do**
14: $\quad\quad \mathcal{G}_{ij} \leftarrow G^t_{ik}$
15: $\quad \mathcal{K} \leftarrow \mathcal{K} \setminus k$

corresponding *batch* in G^t (line 7), and updates $\mathbb{N}^{\cup}_j$ and $\mathcal{K}$ accordingly. In the second phase, we reorganize the partition at the *batch* granularity to maximize the number of intra-GPU duplicated transition vertices. The algorithm initializes $\mathcal{G}$ with *batch* 0 and records other *batches* in $\mathcal{K}$ (lines 8-10). During execution, it iteratively searches in $\mathcal{K}$ to find the *batch* that has the maximum number of duplicated transition vertices with the current *batch* (line 12), and writes the found *batch* to $\mathcal{G}$. After processing all batches, we obtain the communication-efficient reorganized partition $\mathcal{G}$.

Effectiveness with various interconnects. The proposed deduplicated communication framework offers benefits to GPU servers equipped with various interconnects. As discussed in Section 5.3, The vertex data to be transferred are divided into three subsets and handled with CPU-GPU communication, inter-GPU communication, and intra-GPU data reuse. Intra-GPU reuse consistently delivers benefits as its bandwidth T_{ru} is associated with the GPU memory bandwidth and often much higher than T_{hd}, which is associated with the GPU-CPU interconnect bandwidth (typically using PCIe). The effectiveness of inter-GPU data sharing depends on the bandwidth of inter-GPU interconnects. Fast interconnects such as NVIDIA NVLink [40] and AMD Infinity Fabric [2], inter-GPU communication provide substantial performance improvements through high-speed inter-GPU communication. Conversely, if GPUs are interconnected via slow PCIe, resulting in T_{hd} being equal to T_{dd}, inter-GPU communication does not bring enhancements. Nevertheless, employing the intra-GPU reuse optimization alone still yield considerable reductions in data transfer. As shown in Table 8, the intra-GPU duplication accounts for 36%-84% of the total duplication volume.

6 IMPLEMENTATION

The use of deduplicated communication can significantly reduce the volume of host-GPU communication, but achieving high performance requires careful implementation, particularly for irregular memory access during communication. First, transferring the data of $\mathbb{N}_{ij}$ and N_{ij} among CPU and GPUs involves non-continuous memory access. Conventional communication methods, such as NCCL [39] and `cudaMemcpy`, are unsuitable for our task as they are designed to operate on contiguous memory. Designing additional data compaction modules can increase the CPU overhead [30, 35]. Second, switching data from $\mathbf{h}_{\mathbb{N}_{ij-1}}$ to $\mathbf{h}_{\mathbb{N}_{ij}}$ needs to reserve the data of $\mathbb{N}^{gpu}_{ij}$ from $\mathbf{h}_{\mathbb{N}_{ij-1}}$

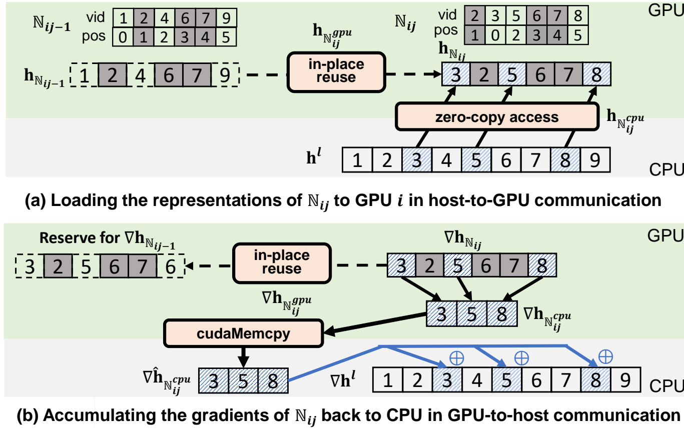


Fig. 7. Implementation of host-GPU communication.

and load the data of $\mathbb{N}_{ij}^{cpu}$ from CPU to $\mathbf{h}_{\mathbb{N}_{ij}}$. This process causes random memory manipulation on the two data buffers. To address these issues, HongTu provides a high performance communication implementation with two features: **communicate-on-demand** and **-update-in-place**.

On-demand communication. HongTu employs zero-copy memory access [34] and GPUDirect P2P access [60], which allow GPUs to directly access the memory of CPUs and other GPUs within the CUDA kernel by mapping them to the same memory address. Moreover, we implement the coalesced-and-aligned memory access optimization [34, 35], which optimize the PCIe bandwidth utilization by enabling each warp of threads to access the contiguous dimension of data. In this way, irregular and non-continous data communication among CPU and GPUs can be performed efficiently.

In-place transition data management. HongTu uses a single data buffer to maintain the transition vertex data for all subgraphs in a partition, and n position indices for maintaining the write position of transition vertices $\{\mathbb{N}_{ij}|0\leq j<n\}$ in the buffer. When scheduling a new *batch*, the data of newly scheduled transition vertices ($\mathbb{N}_{ij}$) are write to the buffer according to the indices. In the preprocessing, we process the transition indices for all subgraphs, making the duplicated vertices of each pair of adjacently-scheduled subgraphs have the same write positions. This allows the data of $\mathbb{N}_{ij}^{gpu}$ to be reused in-place. The data of $\mathbb{N}_{ij}^{cpu}$, which are loaded from the CPU, are inserted into the buffer based on their write positions in the indices. Figure 7 (a) shows a example of data loading in the host-to-GPU communication. Duplicated vertices between $\mathbb{N}_{ij-1}$ and $\mathbb{N}_{ij}$, i.e., $\{2, 6, 7\}$ have the same positions in the transition data buffer. When updating $\mathbf{h}_{\mathbb{N}_{ij-1}}$ to $\mathbf{h}_{\mathbb{N}_{ij}}$, the data of these vertices are reused in-place. In contrast, the data loaded from CPU, $\{3, 5, 8\}$, are inserted into the positions of discarded vertices $\{1, 4, 9\}$.

In-place neighbor data management. HongTu uses a single data buffer to maintain the neighbor data for all subgraphs in a partition. It uses a neighbor index for each neighbor set N_{ij} to track their read positions in the local/remote transition data buffer and the write positions in the local neighbor data buffer. When switching the neighbor data between subgraphs, data of N_{ij} are exchanged between the transition data buffer and neighbor data buffer according to the indices. We notice that duplicated neighbors between each pair of adjacently-scheduled subgraphs (i.e., $N_{ij} \cap N_{ij-1}$) are redundantly transferred. To address this issue, we extend the data reuse technique to inter-GPU communication. HongTu reorders the neighbor vertices of all subgraphs, making the duplicated

vertices ($N_{ij} \cap N_{ij-1}$) have the same positions in the neighbor data buffer, and reuses the data of them during communication.

Host-GPU communication implementation. HongTu allocates both $\mathbf{h}^l$ and $\nabla \mathbf{h}^l$ on pinned memory with `cudaMallocHost()` to support zero-copy memory access in the host-to-GPU communication. In the GPU-to-host communication, HongTu accumulates the gradients of $\mathbb{N}_{ij}^{cpu}$ to $\nabla \mathbf{h}^l$ in CPU and reserves the gradients of $\mathbb{N}_{ij}^{gpu}$ in GPU. To leverage GPU's high memory bandwidth, HongTu implements a GPU-based compaction module as shown in Figure 7 (b). The gradients to be moved out are first collected in GPU memory and then transferred back to CPU using `cudaMemcpyAsync()`.

Inter-GPU communication implementation. HongTu enables GPUDirect P2P access through the `cudaDeviceEnablePeerAccess()` function, which facilitates direct memory access between GPUs. In the forward pass, HongTu uses pull-based communication, where each vertex in N_{ij} reads its representation from the corresponding GPU. In the backward pass, HongTu employs a push-based communication scheme to accumulate the gradients of N_{ij} back to the transition data buffer in the corresponding GPUs, utilizing the `atomicAdd_system()` function [40]. To avoid resource contention caused by multiple GPUs accessing the data from the same GPU, we implement interleaved communication optimization [65] that avoids different GPUs accessing one GPU at the same time slot, as shown in Algorithm 2 (line 6) and Algorithm 3 (line 2).

Data buffer deduplication. Maintaining the data buffer of transition vertices $\mathbb{N}_{ij}$ and neighbor vertices N_{ij} separately leads to doubled data storage overhead on storing the data of $\mathbb{N}_{ij} \cap N_{ij}$. To avoid this issue, HongTu merges $\mathbb{N}_{ij}$ and N_{ij} and maintains the data of $\mathbb{N}_{ij} \cup N_{ij}$ with a single data buffer. Additionally, HongTu regenerates the position indices and modifies the topology of each subgraph to ensure that the computation engine can read and write the merged data buffer directly.

Computation engine. HongTu 's computation engine is based on cuSparse and Pytorch [41], operating independently from the communication engine, as shown in Algorithm 1 (lines 8, 17, and 18). Following existing frameworks such as Sancus [42] and DGL [55], HongTu organizes the topology of each subgraph chunk into the compressed sparse row/column (CSR/CSC) formats. These subgraph chunks are abstracted as `blocks` in the computation engine, facilitating GNN computations at each layer. Graph operations are implemented using cuSparse, while Pytorch serves as the backend for neural network computation. HongTu provides a GNN layer definition class with `__init__` and `forward` methods, enabling users to specify the model configuration and forward computation using built-in graph operations and Pytorch functions. Users also have the option to train their self-implemented GNN models in Pytorch or DGL by overloading these functions with their single-process codes. In case a different graph input format is used, users are required to convert the partitioned subgraphs into their preferred format in the preprocessing stage. The dataflow graph and autograd libraries of Pytorch are used for gradient computation, relieving users from explicitly managing the gradient calculations.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Environments. The multi-GPU experiments are conducted on a GPU server equipped with 4 AMD EPYC 7543 CPUs, 512GB DRAM, and 4 NVIDIA A100 (80GB) GPU. Each GPU is connected to a CPU via PCIe 4.0 link, and each CPU contains 128GB of local memory. The four GPUs are connected through NVLINK-3.0, providing 200GB/s inter-GPU bandwidth. The server runs Ubuntu 18.04 OS with GCC-7.5, CUDA 11.2 and PyTorch v1.9 backend [43]. The single-node CPU experiments are conducted on a Server contains two Intel Xeon 6246R CPU @3.40 GHz with a total of 32 cores

Table 4. Dataset description. $|V|$, $|E|$, $\#\mathbb{F}$, and $\#\mathbb{L}$ represent the number of vertices, edges, features, and labels, respectively.

Dataset	|V|	|E|	#𝔽	#𝕃	Type
reddit [15] (RDT)	0.23M	114M	602	41	post-to-post
ogbn-products [16] (OPT)	2.4M	62M	100	47	co-purchasing
it-2004 [3] (IT)	41M	1.2B	256	64	web graph
ogbn-paper [16] (OPR)	111M	1.6B	200	172	citation network
friendster [22](FDS)	65.6M	2.5B	256	64	social network

and 768 GB of memory. The distributed CPU experiments are conducted on a 16-node Aliyun ECS cluster. Each node (ecs.r5.16xlarge instance) is equipped with 56 vCPUs and 512GB DRAM. The network bandwidth is 20 Gbps. All these machines run Ubuntu 20.04.

Datasets and GNN algorithms. Table 4 presents the major parameters of the real-world graphs used in our experiments. For graphs without ground-truth properties (it-2004 and friendster), we use randomly generated features, labels, training (25%), test (25%) and validation (50%) set division. We use two popular GNN models with different computation patterns, GCN [21] has heavy-weight vertex computation and light-weight edge computation. GAT [52] has heavy-weight edge computation and light-weight vertex computation. The hidden layer dimensions for reddit and ogbn-products are set to 256, while for it-2004, ogbn-paper, and friendster, they are set to 128. In our evaluation, the number of partitions is set to 4. Since reddit and ogbn-products are small, their partitions are not additionally split. Each partition of it-2004, ogbn-paper, and friendster is divided into 8, 32, and 32 (resp. 16, 64, 64) chunks in GCN (resp. GAT) training, respectively.

The systems for comparison. We compare HongTu with three full-graph GNN systems: single-GPU DGL v0.9 [55], multi-GPU-based Sancus [42], and CPU-based DistGNN [32], as well as a GPU-based mini-batch GNN system DistDGL[62]. In DistDGL, the fan-out of neighbor sampling per layer is set to 10, and the batch size is set to 1024. We provide an in-memory version (HongTu-IM) that places all the training data in GPU to demonstrate the effectiveness of the GPU computation engine. We also provide a single-GPU HongTu with the inter-GPU communication disabled for comparison with DGL and single-CPU-based DistGNN.

Comparison with CPU data offloading techniques in DNN training. Certain DNN frameworks also employ CPU data offloading to mitigate GPU memory overhead. DeepSpeed [44–46] is a representative system that stores model parameters in CPU and offload computation to GPUs. However, these frameworks are designed for DNN training with large models and lack GNN-specific consideration, limiting their effectiveness in supporting GNN training (Section 8). To illustrate this, we compare HongTu with DeepSpeed [33] in Section 7.3. Since DeepSpeed does not support GNN training, we implement its data offloading method in HongTu as our baseline (denoted by **Baseline** in Figure 9), which transfers the neighbor data for each subgraph entirely. The host-GPU on-demand access optimization (Section 6) is enabled in the baseline to enhance CPU-GPU communication. Both DeepSpeed and HongTu employ recomputation-based training [5]. For a fair comparison, we enable recomputation-cache hybrid intermediate data management (Section 4.2) in both frameworks, even though DeepSpeed does not have this optimization.

Accuracy and evaluation metric. Full-graph GNN can achieve theoretical accuracy in HongTu because its training semantic is not changed. Figure 8 shows the validation and test accuracy of HongTu and DGL in full-graph training for a GCN model. After 100 epochs, the validation accuracy reached a stable state, HongTu and DGL-FG almost achieved the same validation and test accuracy. Since the accuracy gap between DGL and HongTu is almost negligible, we report the per-epoch runtime, i.e., the time to conduct a forward and backward pass over the entire graph. Shorter

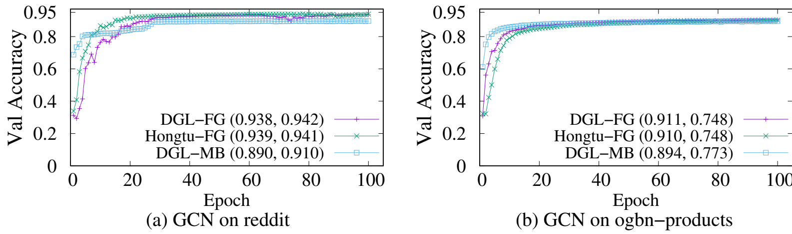


Fig. 8. Validation accuracy curves of DGL (full-graph), DistDGL (mini-batch) and HongTu for GCN with 100 epochs. The values in (#val, #test) are the final validation and test accuracy, respectively.

per-epoch time indicates better time-to-accuracy performance, and all the results are averaged over 20 epochs to ensure consistency. In comparison with mini-batch training-based DGL, HongTu achieves higher test and validation accuracy on reddit, while mini-batch training performs better on ogbn-products. Both the mini-batch and full-graph training methods possess distinct merits. However, assessing their effectiveness requires a comprehensive analysis and consideration of various factors, including batch size, sampling fan-out, and characteristics of input graphs, which is out of the scope of this work.

Table 5. Comparison with DGL and DistGNN on two small datasets.

Layers	System	Runtime of GCN (s)		Runtime of GAT (s)	
		RDT	**OPT**	**RDT**	**OPT**
2	DistGNN	4.2	10.1	40.7	49.9
	DGL	0.19 (21×)	0.27 (37×)	0.86 (47×)	1.22 (41×)
	HongTu-IM	0.20 (21×)	0.32 (31×)	0.77 (53×)	1.14 (43×)
	HongTu	0.33 (12×)	0.84 (12×)	1.15 (35×)	1.93 (26×)
4	DistGNN	7.78	22.9	77.9	220.8
	DGL	0.39 (20×)	0.82 (28×)	1.35 (58×)	2.19 (101×)
	HongTu-IM	0.39 (20×)	0.81 (28×)	1.21 (64×)	2.01 (109 ×)
	HongTu	0.62 (13×)	2.09 (11×)	2.21 (35×)	3.82 (58×)
8	DistGNN	15.1	46.2	148.4	418.6
	DGL	0.78 (19×)	1.92 (24×)	2.77 (54×)	OOM
	HongTu-IM	0.69 (22×)	1.76 (13×)	2.43 (26×)	OOM
	HongTu	1.15 (13×)	4.14 (11×)	4.50 (32×)	8.23 (50×)

7.2 Overall Comparison

First, we compare HongTu with single-CPU and single-GPU systems on small graphs to show the efficiency of GPU computation engine. Then we compare HongTu with multi-GPU systems on all graphs to evaluate its processing scale with limited GPU resources. Finally, we compare HongTu with a distributed CPU system on large graphs, evaluating its efficiency and low monetary cost.

Comparison with single-GPU and single-CPU systems. We compare HongTu and HongTu-IM with DGL [62] and single-CPU DistGNN by running GCN and GAT on the two small graphs (reddit and ogbn-products). Table 5 shows the runtime results and the speedups normalized to DistGNN. We observe that all three GPU-based solutions achieve more than one order-of-magnitude speedup over the CPU-based solution. HongTu-IM achieves performance similar to, or slightly better than, DGL. HongTu is 1.3×-3.8× slower than DGL due to additional overhead on host-GPU communication and CPU-based gradient accumulation. Although the performance is slightly behind, only HongTu is capable of training complex GNN models with large-scale intermediate data (e.g., GAT).

Table 6. Comparison with Multi-GPU systems on 4 A100 GPUs.

Layers	System	Runtime of GCN (s)				
		RDT	**OPT**	**IT**	**OPR**	**FDS**
2/2	Sancus	0.09	0.37	OOM	OOM	OOM
	HongTu-IM	0.09(1.0×)	0.29(1.27×)	OOM	OOM	OOM
	HongTu	0.13(0.72×)	0.45(0.81×)	3.6	25.1	15.5
	DistDGL	0.61(0.15×)	0.58(0.64×)	33.8	1.95	52.3
4/3	Sancus	0.15	0.65	OOM	OOM	OOM
	HongTu-IM	0.16(0.93×)	0.55(1.18×)	OOM	OOM	OOM
	HongTu	0.23(0.63×)	0.95(0.68×)	6.0	40.0	25.1
	DistDGL	1.49(0.10×)	4.13(0.16×)	281.5	6.95	397.2
8/4	Sancus	0.27	1.16	OOM	OOM	OOM
	HongTu-IM	0.25(1.08×)	1.04(1.15×)	OOM	OOM	OOM
	HongTu	0.45(0.65×)	2.03(0.57×)	8.3	59.4	34.6
	DistDGL	22.4(0.01×)	OOM	OOM	19.2	OOM

Table 7. Comparison with DistGNN on a 16-node ECS cluster.

Layers	Dataset	Runtime of GCN (s)		Runtime of GAT (s)	
		DistGNN	HongTu	**DistGNN**	HongTu
2	IT	38.9	3.6 (10.8×)	151.3	7.5 (20.2×)
	OPR	213.5	25.1 (8.5×)	OOM	42.4
	FDS	183.0	15.5 (11.8×)	OOM	27.5
3	IT	59.5	6.0 (9.9×)	OOM	12.7
	OPR	312.6	40.0 (7.8×)	OOM	65.6
	FDS	277.5	25.1 (11.1×)	OOM	42.8
4	IT	85.7	8.3 (10.3×)	OOM	17.5
	OPR	OOM	59.4	OOM	95.0
	FDS	369.4	34.6 (10.7×)	OOM	58.0

Comparison with multi-GPU system. We compare HongTu with Sancus [42] and DistDGL [62] by running GCN on all five graphs. For the two small graphs, we employ the model configurations with 2, 4, and 8 layers, while for the three large graphs, we employ model configurations with 2, 3, and 4 layers. The results are reported in Table 6. In comparison with Sancus, HongTu-IM delivers comparable performance to Sancus and is 1.2×-1.9× faster than HongTu on the two small graphs. However, both Sancus and HongTu-IM run out of memory on the three large graphs. In contrast, HongTu can effectively process them. DistDGL runs out of memory on ogbn-products, it-2004, and friendster when configured with 8, 4, and 4 layers, respectively. Furthermore, in cases where DistDGL successfully runs, the runtime exhibits exponential growth as the number of layers increases. These challenges arise from the neighbor explosion problem [18], where the computation and memory requirements for mini-batch GNN training increase exponentially with the number of layers. On successfully runs, HongTu outperforms DistDGL on reddit, ogbn-products, it-2004, and friendster. On ogbn-paper, DistDGL achieves a better performance due to its usage of only 1.2M vertices (1.1%) for the training, resulting in significantly lower computation volume compared to HongTu. In summary, HongTu exhibits advantages when training deep GNNs or when the input graph includes a large proportion of training vertices. However, when the training set and the number of model layers are small, DistDGL still holds certain advantages.

Comparison with distributed-CPU system. We compare HongTu with DistGNN [42] by running GCN and GAT on the three large graphs. The results are reported in Table 6. DistGNN runs out-of-memory for 4-layer GCN on ogbn-paper and all GAT workloads except the two-layer GAT training on it-2004. We can observe that training large-scale GNNs remains challenging, even with extensive host memory provided by multiple CPU nodes. Besides large-scale vertex and

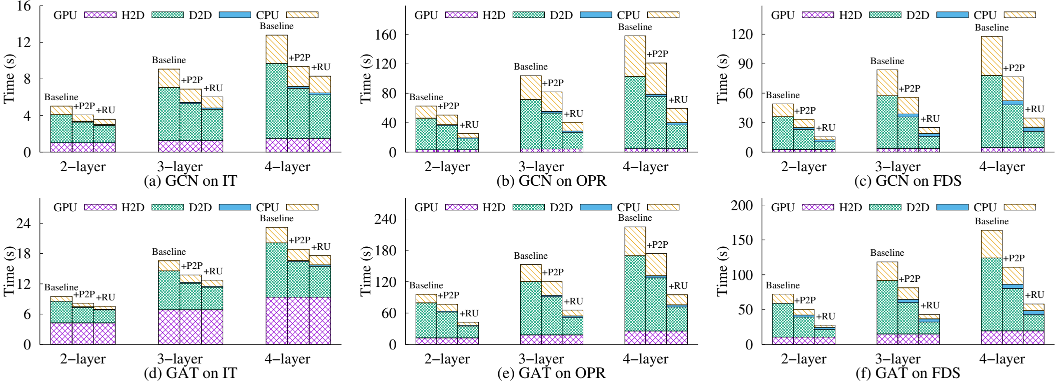


Fig. 9. Performance breakdown of HongTu on GCN and GAT with different hidden layers, where 'Baseline' for the baseline approach, 'P2P' for the inter-GPU communication, and 'RU' for the intra-GPU data reusing. 'GPU' represents the GPU computations, 'H2D' represents the host-GPU communication, 'D2D' represents the inter-GPU communication, and 'CPU' represents the CPU-based gradient accumulation.

Table 8. The proportion of the two types of duplication access on the three billion-scale graphs.

Dataset	Chunks	V_{ori}	$(V_{ori} - V_{+p2p})$	$(V_{+p2p} - V_{+ru})$
it-2004	32	1.6	0.26 (16.2%)	0.15 (9.2%)
ogbn-paper	128	8.5	0.77 (9.0%)	4.1 (48.3%)
friendster	128	10.7	2.50 (23.3%)	5.09 (47.6%)

intermediate data, DistGNN also needs to maintain the data of neighbor replicas and communication buffers for distributed processing. In other cases, HongTu outperforms CPU-based solution. On average, HongTu achieves 10.1× and 20.2× speedups over DistGNN for GCN and GAT models, respectively. Moreover, the per-hour monetary cost for 16 CPU nodes (ecs.r5.16xlarge, 5.24 USD per-nodes) is 4.16 times higher than that for a 4×A100 GPU node (ecs.gn7e-c16g1.16xlarge, 20.14 USD per-node), which has a similar configuration to our private GPU server. Therefore, HongTu provides a cost-effective solution for processing large-scale GNNs.

7.3 Communication Reduction Analysis

We enable inter-GPU and intra-GPU communication deduplication one-by-one to reveal how much HongTu can benefit from each of them. Table 8 illustrates the communication reduction volume normalized to the number of vertices ($|V|$).The results show that these two optimizations reduce host-GPU communication by 25%-71% on the three graphs. Although it-2004 originally has less redundant communication (0.6 times $|V|$), our proposed method still reduces 68% of the total redundant transfers (from $0.6|V|$ to $0.2|V|$). Ogbn-paper benefits more from intra-GPU deduplication due to its co-author graph structure and exhibits good locality.

To demonstrate the practical improvement of communication deduplication, we conducted experiments on GCN and GAT models with 2-, 3-, and 4-layer configurations on the three large graphs. We start from the baseline approach (**Baseline**) that transfers the neighbor data for each subgraph entirely, then enable inter-GPU (denoted by **+P2P**) and intra-GPU communication deduplication (denoted by **+RU**) one-by-one. Figure 9 reports the results. Even with on-demand access optimization (Section 6), the performance of the baseline approach remains inferior, because

Table 9. Analysis of cost of communication deduplication.

Engine	Runtime of 100 epochs on a 2-layer GCN(s)		
	it-2004	**ogbn-paper**	**friendster**
HongTu w/o CD	502.8	6260.2	4907.5
HongTu w/ CD	359.6	2513.0	1554.1
Preprocessing	+4.5	+33.9	+22.7

it suffers from the duplicated neighbor data communication and cross-partition remote host-GPU data access. The inter-GPU data sharing reduces communication time (including host-GPU communication and inter-GPU communication) by 23%- 26%, 23%-27%, and 39%- 42%, on the three graphs respectively. The reduction in transfer time is greater than the reduction in transfer volume because it eliminates the remote neighbor access across CPUs. The intra-GPU data reusing further reduces transfer time by 9%-12%, 39%-42%, and 36%-37% for the three graphs, respectively. Overall, HongTu that uses deduplicated communication can achieve speedups ranging from 1.3× to 3.4× compared to the baseline approach.

Overhead of communication deduplicaton. As communication deduplication has the cost to preprocess the input graph after graph partitioning, we evaluate the overhead and show in Table 9, where the preprocessing time is denoted as "Preprocessing". We compare it to the execution time of running GCN for 100 epochs with and without communication deduplication (CD). We observe that communication deduplication brings up to 1.5% overhead into HongTu while significantly improving performance over the baseline. The low overhead of communication deduplication comes from two folds. First, the preprocessing uses a heuristic design and is executed in parallel. Second, as full-graph GNN training follows the same execution pattern in different layers, the preprocessing only needs to be performed once.

7.4 Performance Breakdown

We provide a performance breakdown to analyze the time consumption of different components, including the host-GPU communication (H2D), inter-GPU communication (D2D), GPU-based computation (GPU), and CPU-based gradient accumulation (CPU). Figure 9 shows the experimental results. The GPU computation time varies among different GNNs due to their varying computation complexities. In GCN with simple arithmetic edge computation, GPU computation accounts for 10%-14% of the overall runtime. In contrast, in GAT with parameterized edge computation, the GPU computation time is 4.5 times longer than that of GCN and accounts for 54%, 28%, and 35% of the total runtime. The communication (H2D+D2D) time varies among different GNNs. GCN benefits from recomputation-caching-hybrid training, reducing its communication time by 21%-29% compared to GAT training. Overall, the communication time accounts for 58%-61% and 36%-50% of the overall runtime on GCN and GAT, respectively. As HongTu utilizes CPUs to accumulate the neighbor gradients, the CPU computation time is proportional to the volume of transferred neighbors, which accounts for 8% to 30% of the overall runtime.

7.5 Sensitivity Study

Performance with varying layers. Since the computation pattern is exactly the same, communication deduplication is equally effective for all GNN layers. Therefore, increasing the number of layers will not decrease the optimization effect. As shown in Figure 9. HongTu achieves 1.4×-1.5×, 2.5×-2.7×, 3.2×-3.4×, 1.3×-1.3×, 2.3×-2.4×, and 2.6×-2.8× speedups over the vanilla approach under different layer configurations. The optimization effect is stable.

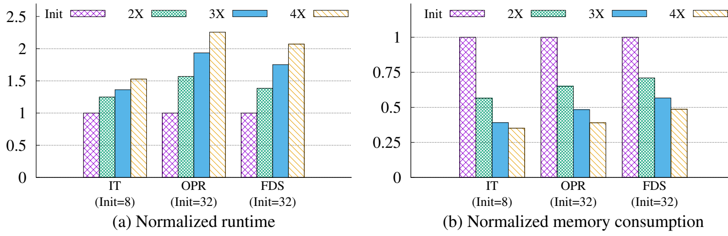


Fig. 10. Runtime and memory analysis of HongTu with different chunks. We run GCN on each graph from an initial chunk size and increase the chunk size to 2×, 3×, and 4×.

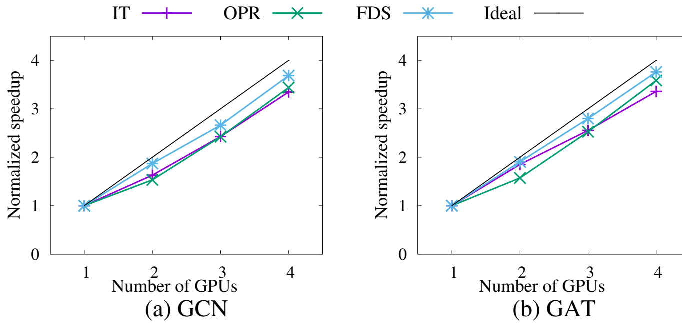


Fig. 11. Scaling perf. when varying GPU number from 1 to 4.

Performance with varying chunks. The chunk size in HongTu is a configurable parameter that controls the memory consumption of training data. However, increasing the chunk size also leads to increased duplicated neighbors, which subsequently increases the volume of host-GPU communication. To evaluate the impact of chunk size, we run GCN on three large graphs, starting from the initial chunk size and increasing it by a factor of 2, 3, and 4. The experimental results in Figure 10 show that as the chunk size increases by 4×, the memory consumption decreases by 51%-65%, and the runtime increases by 1.5× to 2.2×. The increase in runtime is either linear or sublinear, and is proportional to the decrease in memory consumption. Additionally, the performance of HongTu can be improved by using GPUs with larger memory capacity, although our approach can be adapted to GPUs of different grades.

7.6 Scalability of HongTu

We evaluate the scalability of HongTu by varying the number of GPUs used in training. Figure 11 shows the normalized speedups of GCN and GAT training on it-2004, ogbn-paper, and friendster. The execution time of HongTu is reduced when using more GPUs. Specifically, when the number of GPUs increases from 1 to 4, HongTu achieves 3.3-3.7× speedups for GCN training and 3.4-3.8× speedups for GAT training. The speedups from 1 to 2 sockets is lower than that from 2 to 4 sockets because we do not have enough CPU memory to enable the NUMA-aware vertex data allocation. When using two or fewer GPUs, we must use the memory from all sockets, resulting in remote memory access overhead.

8 RELATED WORK

As the size of DNN models increases, the traditional data parallel (DP) training method, which replicates model parameters across all training processes, faces scalability issues due to the increasing memory consumption [1, 36, 54]. To address this, various parallel training methods have been proposed, including model parallelism [6, 54], pipeline parallelism [17, 36, 37], partitioned data

parallelism [44] (which partitions model states among data-parallel workers to eliminate memory redundancy), and 3D parallelism [33] (which combines model, pipeline, and data parallelism to leverage the aggregated GPU memory of a cluster). However, the scalability of these frameworks remains constrained by the available GPU resources. To address the limitation of GPU memory, DeepSpeed [33] incorporates CPU-based data offloading techniques [44–46]. It achieves this by partitioning the model state into smaller slices and storing them in CPU DRAM or NVM. During training, the required slices are loaded into the GPUs sequentially as they are accessed. While DeepSpeed and our design share similarities in offloading memory-intensive data to CPU memory, they differ in their optimization objectives. DeepSpeed primarily focuses on DNNs with large model parameters. In DNNs, model parameters consist of dense matrices that can be partitioned into disjoint slices without interdependencies. These slices can be efficiently communicated between the CPU and GPUs due to their regular data access patterns. In contrast, HongTu is tailored for GNN training, where the memory overhead primarily arises from vertex data, and the model data typically have small sizes. Due to the inherent complexity of graph structures, vertex data are randomly distributed and duplicated across partitions, resulting in irregular and increased host-GPU data communication. DeepSpeed's approach does not adequately address these challenges. However, HongTu effectively resolves this problem through its deduplicated communication framework.

9 CONCLUSION

We present HongTu, a scalable and efficient system for training full-graph GNNs on limited GPU memory. Our system leverages two key components to achieve its performance, including a memory-efficient GNN training framework that combines the partition-based GNN training and recomputation-cache-hybrid intermediate data management, a deduplicated communication framework that converts the redundant host communication for duplicated neighbors to inter-GPU and intra-GPU data access. Our experiments demonstrate that HongTu can efficiently train on billion-scale graphs using just 4 GPUs by fully utilizing CPU, GPU, and interconnects.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002), the Ministry of Education AcRF Tier 2 grant (No. MOE-000242-00/MOE-000242-01), a grant from NUS Advanced Research and Technology Innovation Centre (ARTIC), and Google South & Southeast Asia Research Award 2022.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283.

[2] AMD. 2022. AMD Infinity Architecture. https://www.amd.com/en/technologies/infinity-architecture.

[3] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[4] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 130–144.

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* 1232–1240.

[7] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 3837–3845.

[8] DGL 2020. Deep Graph Library:towards efficient and scalable deep learning on graphs. https://www.dgl.ai/.

[9] Euler 2019. Euler. https://github.com/alibaba/euler/wiki/System-Introduction.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[11] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1603–1618.

[12] Congcong Ge, Xiaoze Liu, Lu Chen, Baihua Zheng, and Yunjun Gao. 2022. LargeEA: Aligning Entities for Large-scale Knowledge Graphs. *PVLDB* 15, 2 (2022), 237–245.

[13] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 916–931.

[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 1024–1034. http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs

[15] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034.

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. https://doi.org/10.48550/ARXIV.2005.00687

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.* 103–112.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020.*

[19] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. 1995. Chapter 4 The traveling salesman problem. In *Network Models*. Handbooks in Operations Research and Management Science, Vol. 7. Elsevier, 225–330. https://doi.org/10.1016/S0927-0507(05)80121-5

[20] George Karypis and Vipin Kumar. 1996. Parallel Multilevel Graph Partitioning. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*. 314–319.

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.*

[22] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[23] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9, 14 (2016), 1647–1658.

[24] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[25] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.*

[26] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 401–415.

[28] Qi Liu, Maximilian Nickel, and Douwe Kiela. 2019. Hyperbolic Graph Neural Networks. *CoRR* abs/1910.12892 (2019).

[29] Xiaoze Liu, Junyang Wu, Tianyi Li, Lu Chen, and Yunjun Gao. 2023. Unsupervised Entity Alignment for Temporal Knowledge Graphs. In *WWW*. 2528–2538.

[30] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. 443–458.

[31] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, Martha Palmer, Rebecca Hwa, and Sebastian Riedel (Eds.). Association for Computational Linguistics, 1506–1515.

[32] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: scalable distributed training for large-scale graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 76.

[33] Microsoft. 2020. Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scalemodel-training-for-everyone.

[34] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 114–127.

[35] Seungwon Min, Kun Wu, Sitao Huang, Mert Hidayetoglu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei W. Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (2021), 2087–2100.

[36] 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, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 1–15.

[37] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7937–7947.

[38] NVIDIA. 2022. NVIDIA A100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/a100/.

[39] NVIDIA. 2022. NVIDIA Collective Communication Library. https://developer.nvidia.com/nccl.

[40] NVIDIA. 2022. NVLink and NVSwitch. https://www.nvidia.com/en-sg/data-center/nvlink/.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, 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: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 8024–8035.

[42] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.

[43] PyTorch 2020. Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org/.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 20.

[45] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59.

[46] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564.

[47] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 12:1–12:16.

[48] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 28:1–28:12.

[49] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. 2244–2252.

[50] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 495–514.

[51] Alok Tripathy, Katherine A. Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. 70.

[52] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[53] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[54] Minjie Wang, Chien-Chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 26:1–26:17.

[55] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[56] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1301–1315.

[57] Shiwen Wu, Wentao Zhang, Fei Sun, and Bin Cui. 2020. Graph Neural Networks in Recommender Systems: A Survey. *CoRR* abs/2011.02260 (2020).

[58] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=ryGs6iA5Km

[60] D. Yang, J. Liu, J. Qi, and J. Lai. 2022. WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 767–780.

[61] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 417–434.

[62] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *CoRR* abs/2010.05337 (2020).

[63] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 573–588.

[64] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *PVLDB* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.

3352127

[65] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 301–316.

Received April 2023; revised July 2023; accepted August 2023