MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms

Yuke Wang, Boyuan Feng, and Zheng Wang, *University of California Santa Barbara;* Tong Geng, *University of Rochester;* Kevin Barker and Ang Li, *Pacific Northwest National Laboratory;* Yufei Ding, *University of California Santa Barbara*

https://www.usenix.org/conference/osdi23/presentation/wang-yuke

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms

Yuke Wang, Boyuan Feng, Zheng Wang, †Tong Geng, *Kevin Barker, *Ang Li, and Yufei Ding
†*University of Rochester,* **Pacific Northwest National Laboratory*
University of California, Santa Barbara

Abstract

The increasing size of input graphs for graph neural networks (GNNs) highlights the demand for using multi-GPU platforms. However, existing multi-GPU GNN systems optimize the computation and communication individually based on the conventional practice of scaling dense DNNs. For irregularly sparse and fine-grained GNN workloads, such solutions miss the opportunity to jointly schedule/optimize the computation and communication operations for high-performance delivery.

To this end, we propose **MGG** , a novel system design to accelerate full-graph GNNs on multi-GPU platforms. The core of MGG is its novel dynamic software pipeline to facilitate fine-grained computation-communication overlapping within a GPU kernel. Specifically, MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping. MGG also incorporates an intelligent runtime design with analytical modeling and optimization heuristics to dynamically improve the execution performance. Extensive evaluation reveals that MGG outperforms state-of-the-art full-graph GNN systems across various settings: on average 4.41×, 4.81×, and 10.83× faster than DGL, MGG-UVM, and ROC, respectively.

1 Introduction

Over the recent years, graph-based deep learning has attracted lots of attention from the research and industry communities. Among various graph-learning methods, graph neural network (GNN) [21, 43, 49] gets highlighted most due to its success in many deep learning tasks (*e.g.*, node feature vector (embedding) generation for node classification [11, 13, 19] and link prediction [7, 22, 42]). GNNs consist of several layers, where layer $k+1$ computes the embedding for a node v based on the embeddings at the previous layer k ($k \geq 0$) by applying

$$a_v^{(k+1)} = \textbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)})$$
$$h_v^{(k+1)} = \textbf{Update}^{(k+1)}(a_v^{(k+1)})$$

where $h_v^{(k)}$ is the embedding of node v at layer k. The *Aggregate* function accumulates neighbors'($\mathbf{N}(v)$) embeddings of node v. The *Update* function consists of a fully-connected NN layer. The neighbor aggregation (*Aggregate*) is the key bottleneck that dominates the overall computation due to its high computation sparsity and irregularity [46, 50]. Compared with conventional graph analytics (e.g., random walk [14, 39]), GNN features higher accuracy [21, 49] and better generality [16, 55] on various applications.

GNN computation on large input graphs (millions/billions of nodes and edges) usually counts on powerful multi-GPU platforms (e.g., NVIDIA DGX [35]) for scaling up the performance. The multi-GPU system (that can potentially store all data required for the computation in the aggregate memory of all GPUs on a single machine) can benefit from aggregated memory capacity and bandwidth (HBM and NVLinks) with more GPUs. There is also a popular trend for state-of-the-art hyper-scale systems employing GPU-centric building blocks. For example, the recent NVIDIA DGX SuperPod [33] consists of 32×DGX-H100 servers (each with 8×H100). Unfortunately, the runtime performance of GNNs does not scale proportionally with the aggregated compute capability and memory capacity of the platform. This is mainly because the irregular and sparse local memory access of neighbor aggregation in the single-GPU settings now "scales" to more expensive inter-GPU communication (i.e., remote memory access). Such intensive inter-GPU communication becomes the new critical path of multi-GPU GNN execution and offsets the performance gains from multi-GPU computation parallelism.

Based on this observation, we highlight a more promising way of formalizing GNN computation on multi-GPU systems. Our key insight is that GNN execution can be more precisely abstracted as a fine-grained dynamic software pipeline to encourage communication and computation overlapping, which will largely hide the communication cost. The opportunities for building such fine-grained pipelines widely exist at different granularities in GNNs. For instance, on a single graph node, the remote neighbor access can be overlapped with the local neighbor computation. Among different graph nodes,

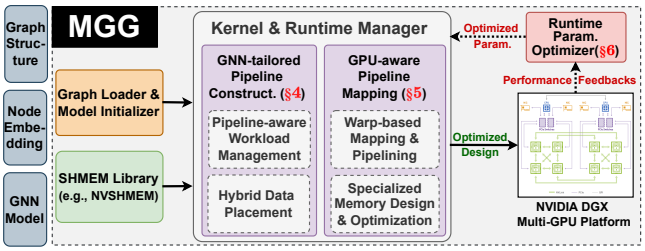


Figure 1: Overview of MGG.

the remote neighbor access for certain nodes would potentially be overlapped with the local neighbor computation of some other nodes. However, prior research could hardly exploit such benefits since they rely on hardware and software infrastructures tailored for coarse-grained [18, 28] and regular communication patterns [12, 26]. To capitalize on the fine-grained pipelining benefits, there are three major challenges.

The first challenge is *how to craft the pipeline structure*. A work-efficient pipeline for GNNs demands comprehensively considering multiple factors (e.g., the operations and the number/granularity of each pipeline stage) to best fit the GNN algorithm and multi-GPU computation/communication. The second challenge is *how to map the pipeline to the GPU processing units*. Given the GPU's architectural complexity (e.g., multi-granular processing units and multi-layer memory hierarchy), different mapping and primitive choices would bring performance and design flexibility tradeoffs. The third challenge is *how to find and adapt toward the "optimal" pipeline configuration swiftly*. Given the diversity of GNN inputs (*e.g.*, graph structures) and hardware (*e.g.*, different types/numbers of GPUs), pinpointing the best-off design configuration with high-performance delivery relies on combined insights from the properties of the software pipeline, GNN inputs, and GPU programming and execution paradigms.

To this end, we introduce a set of principles for multi-GPU GNN acceleration via a fine-grained dynamic software pipeline. *To construct fine-grained pipelines*, the original coarse-grained irregular GNN computation should be breakdown into fine-grained operations. The joint optimization of the GNN workload granularity and data layout should be carried out to facilitate operation overlapping. *To map pipelines to GPUs*, the proper GPU logical processing units (e.g., thread, warp, and block) should be selected for promoting GPU kernel efficiency and design flexibility. In addition, the right choice of communication primitives (e.g., NVSHMEM [36]) should be determined to provide fine-grained inter-GPU communication support. *To adapt pipelines dynamically*, customized kernel templates with tunning knobs should be devised. This will help to maintain pipelining effectiveness across a diverse range of GNN inputs and hardware platform settings.

We crystallize the above principles into MGG[1], a holistic system design and implementation for multi-GPU GNNs (Figure 1). Given the GNN models and inputs, MGG will automatically generate pipeline-centric GPU kernels for multi-GPU platforms and dynamically improve the kernel performance based on runtime feedback. The core of MGG is its **Kernel & Runtime Manager**, which constructs GNN-tailored pipelines and maps such pipelines to proper communication primitives and GPU logical processing units. It can also dynamically orchestrate GPU kernels based on new configurations. MGG also incorporates a **Runtime Parameter Optimizer**, which will monitor the performance (e.g., latency) from the actual execution and generate new configurations for the next iteration based on the analytical performance model and optimization heuristics. To the best of our knowledge, we are the first to explore the potential of GPU kernel operation pipelining for accelerating irregular GNN workloads. Moreover, MGG can be generalized to other applications (e.g., deep-learning recommendation model (DLRM) [31]) that are sharing similar irregular communication demands (§7.3).

Overall, we make the following contributions in this paper:

- We propose a GNN-tailored pipeline construction technique (§4) with pipeline-aware workload management and hybrid data placement, for efficient communication-computation pipelining in a GPU kernel.
- We introduce a GPU-aware pipeline mapping strategy (§5), encompassing warp-based mapping and pipelining, and specialized memory designs and optimizations to comprehensively promote kernel performance.
- We devise an intelligent runtime with lightweight analytical modeling and optimization heuristics to dynamically improve the performance of GNN training (§6).
- Comprehensive experiments demonstrate that MGG can outperform state-of-the-art multi-GPU GNN systems across various GNN benchmarks. Additionally, MGG can be generalized to other DL applications, like DLRM.

2 Related Work

Recent deep-learning applications expand their scope from handling structured dense inputs (e.g., images) to unstructured sparse inputs (e.g., graphs). Along with such algorithmic/application expansion is the exploration of new system designs and optimizations for more efficient deep learning. One of the most important topics is the ability to handle large-scale inputs, which are usually out of the computation and memory capacity of one GPU. For scaling regular deep-learning applications, like dense DNNs, various abstractions (e.g., data and model parallel) and high-performance communication libraries (e.g., NCCL [34]) have been developed. While the scaling approach for irregular GNN applications is still initial and suffers from unsatisfactory performance.

Compared to scaling dense DNNs, scaling sparse GNNs is significantly more challenging. The irregular fine-grained sparse GNNs workload cannot fit the regular coarse-grained

[1] https://github.com/YukeWang96/MGG-OSDI23-AE.git

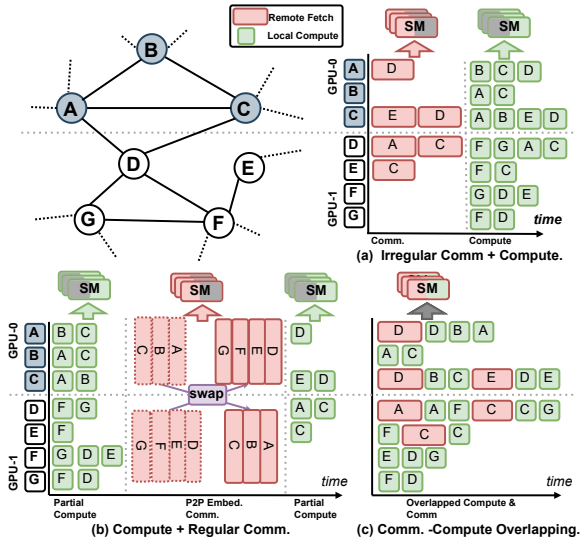


Figure 2: Different Multi-GPU GNN strategies for computation and communication. Note that red and green boxes indicate aggregation workload on remote and local neighbors. "SM" boxes with grey areas indicate potential idleness.

workload abstraction for dense DNNs. The cost of irregular communication in GNNs cannot be easily amortized by simply batching more requests as dense DNNs due to their randomness and sparseness. Scaling strategies largely vary among different GNN inputs while tiling/schedule strategies would be reused across different inputs of dense DNNs. Therefore, an array of dedicated designs have been introduced to scale the sparse GNNs, focusing on three major directions.

Operator Specialization for Sparse Communication: This is the mainstream solution that treats the communication as a standalone operator for irregularly sparse GNN communication (Figure 2(a)). DGL [45] is the state-of-the-art GNN framework and its most recent update incorporates PyTorch-Direct [28] (a GNN-tailored communication design based on zero-copy memory [41]) for large-scale GNN training across GPUs. Work from [6] introduces a communication planning algorithm for distributed GNNs by considering links, communication, contention, and load balancing. However, these efforts optimize the communication standalone and thus miss the opportunities to jointly optimize computation and communication operations/schedules which can potentially reduce the overall latency and improve GPU utilization.

Algorithm Modification for no Communication: The second typical type is to eliminate irregular communication by altering algorithms [25, 45, 51]. They harness various algorithmic adaption solutions, such as neighbor sampling and mini-batch to prefetch the remote neighbors to local devices, and then train the GNN model in a data-parallel fashion as the traditional dense DNN. However, existing research [8, 18] shows that such an algorithmic modification would compromise the accuracy of GNN models compared to the original GNNs. It would also destabilize the algorithmic performance (*e.g.*, the lower convergence speed and final accuracy) under different inputs and sampling configurations.

Schedule Transformation for Dense Communication: The third type is to transform irregular communication to regularized communication (e.g., AlltoAll, P2P), which has been optimized by existing communication kernels (Figure 2(b)). ROC [18] delegates communication to its underlying NVIDIA Legion runtime [5], which manages irregular remote neighbor access via a DMA engine. It batches fine-grained embeddings into large embedding tiles on CPUs to facilitate coarse-grained data movement between the host and GPUs. NeuGraph [26] tiles the large node embedding matrices by rows (as embedding chunks) and then forwards each chunk to GPUs sequentially via coarse-grained P2P communication. P3 [12] spots the potential of transforming irregular embedding communication to regular all-to-all communication for embedding column tiles. However, this type of effort would introduce many unnecessary data movements and non-trivial overhead to transform original algorithms and data inputs.

To sum up, existing designs explore solutions in a limited scope and have yet to extend their solution search to a broader context by exploring the synergy between the multi-GPU GNN workloads, GPU execution paradigms, and communication patterns. Therefore, these designs could hardly enjoy the full potential of multi-GPU platforms.

3 Motivation

Different from prior solutions, we propose a new view for multi-GPU GNN workload. We spot that by removing the explicit barrier between the computation and communication stage in multi-GPU GNNs, we can co-schedule the operations from both stages in a holistic way that can reduce the GPU resource idleness and promote performance (Figure 2(c)). For example, when GPUs initiate remote access requests and are waiting for the arrival of remote data, the idle cycles of GPUs can be fulfilled by other local computing workloads. Such insight enables us to abstract the multi-GPU GNN workload as a fine-grained dynamic software pipeline for communication and communication overlapping. Specifically, "Fine-grained" means that the operations at each pipeline stage are tiny (e.g., the aggregation of one neighbor's embeddings) versus DNN layers."Dynamic" means that the division of computation into pipeline stages would vary among different inputs in contrast to DNNs with a relatively fixed pipeline. Such a new design is motivated by our three major observations.

GNN Workload Speciality: The first observation reveals the specialty of GNN workloads, which feature two major types of partial dependency that facilitate pipelining [1]. The first type is the fine-grained neighbor aggregation dependency, where the neighbor embeddings of individual graph nodes are aggregated either sequentially or in parallel with proper synchronization. The second type is the dynamic execution

Listing 1: NVSHMEM APIs in CUDA C.

```
// Initialize an NVSHMEM context on CPUs.
nvshmem_init();
// Get the current GPU device ID on CPUs.
int gpu_id = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
// Set the GPU based on its device ID on CPUs.
cudaSetDevice(gpu_id);
// Define NVSHMEM memory visible for all GPUs on CPUs.
d_shared_mem = (void*) nvshmem_malloc (num_bytes);
// Define global memory visible only for the current GPU.
cudaMalloc((void**) &d_mem, num_bytes);
// Remote access API called by a thread/warp/block.
__device__ nvshmem_float_get_{warp/block}(void *dst, const
      void *src, size_t nelems, int src_gpu_id);
// Sync all GPUs within an NVSHMEM context on CPUs.
nvshmem_barrier_all();
// Release NVSHMEM objects on CPUs.
nvshmem_free(d_shared_mem);
// Terminate the current NVSHMEM context on CPUs.
nvshmem_finalize();
```

dependency on limited processing units, where different operations would compete for limited GPU resources (e.g., SMs) during the runtime. Such two types of dependencies expose new opportunities for us to amortize communication costs by overlapping neighbor aggregation from different nodes.

GPU Execution Characteristics: The second observation highlights the characteristics of the GPU execution paradigm. One key design principle of GPUs is their massive computation/communication parallelism to amortize the unit cost of individual computation/communication operations [40]. The underlying mechanism of GPU hardware design to facilitate this is to simultaneously schedule multiple logical processing units (e.g., threads/warps/blocks) to share the hardware processing units (i.e., GPU SMs). Such a design provides the essential ingredient for pipelining, which is that computation and communication operations can co-run on the same units at the same time to fulfill the idle GPU cycles and maximize the utilization of the GPU hardware processing units. Moreover, with the precise control of GPU kernel launching parameters (e.g., the size of the block and shared memory), the effectiveness of co-running heterogeneous operations can be adjusted so that we can flexibly accommodate different inputs while maintaining high-performance delivery.

Multi-GPU Programming Support: The third observation features the recent advancement of the GPU communication technique and its programming support. The one highlighted most is the NVSHMEM [36], which provides GPU intra-kernel APIs for fine-grained (several to tens of bytes) inter-GPU communication (Listing 1). NVSHMEM is the main communication backend for MGG. Other existing techniques such as Zero-copy memory can also serve as an alternative to NVSHMEM for fine-grained communication. The performance will be similar while NVSHMEM offers better programmability. Some other traditional strategies for inter-GPU communication, would either offer too coarse-grained communication solutions (e.g., unified virtual memory [38] uses KB-level communication granularity) or resort to the default communication strategies of existing multi-GPU-based runtime system (e.g., NVIDIA Legion [5]) without GNN-tailored communication optimization.

These observations and insights motivate MGG, a holistic multi-GPU GNN system with a novel view of GNN workloads as an operation pipeline. MGG automates the pipeline construction, detailed pipeline mapping, and dynamic input-driven pipeline adaption, to improve the GNN scaling.

4 GNN-tailored Pipeline Construction

Constructing a GNN-tailored pipeline are facing two major challenges: *1) How to effectively partition and schedule multi-GPU GNN workloads* so that pipeline efficiency can be maximized; *2) How to properly layout input* so that the hierarchy of GNN inputs and the memory/storage of multi-GPU systems can be carefully matched to facilitate pipeline execution. MGG addresses these challenges with *Pipeline-aware Workload Management* and *Hybrid GNN Data Placement*.

4.1 Pipeline-aware Workload Management

Managing irregularly sparse GNN workloads for pipelining is challenging and could hardly benefit from the prior practice and exploration of the DNN pipeline [29, 30].

Difference from DNN pipeline *First*, balancing the GNN workloads among GPUs has to jointly optimize the computation capacity and the computation/communication irregularity. While the DNN pipeline only needs to balance the computation/memory capacity, since its pipeline stages are well-structured and their inputs are regularly dense. Distributed DNNs require dense regular communication (e.g., Allreduce) that is naturally fit for existing GPU interconnects optimized for throughput and has been optimized by many libraries (e.g., NCCL). In contrast, distributed full-graph GNN (with the entire graph cached on GPUs) is much more challenging since it requires sparse irregular communication that is naturally at odds with the existing hardware interconnects, and fewer efforts have optimized its performance. *Second*, the GNN pipeline workload is more irregular and non-structural and can easily cause pipeline stalls/bubbles. For example, remote neighbor aggregation would have different stages (remote access + aggregation) compared with local neighbor aggregation (local access + aggregation), making it challenging to mix those two heterogeneous workloads. While in the DNN pipeline, all inputs should consistently pass through the same pipeline stages. *Third*, GNN pipeline stages are more fine-grained (e.g., fetching individual embeddings) compared with coarse-grained layers (e.g., GEMMs and Convolutions) in the DNN pipeline. Such small workload granularity enables different pipeline stages to overlap with each other on GPU processing units, like Streaming Multiprocessors (SMs). In

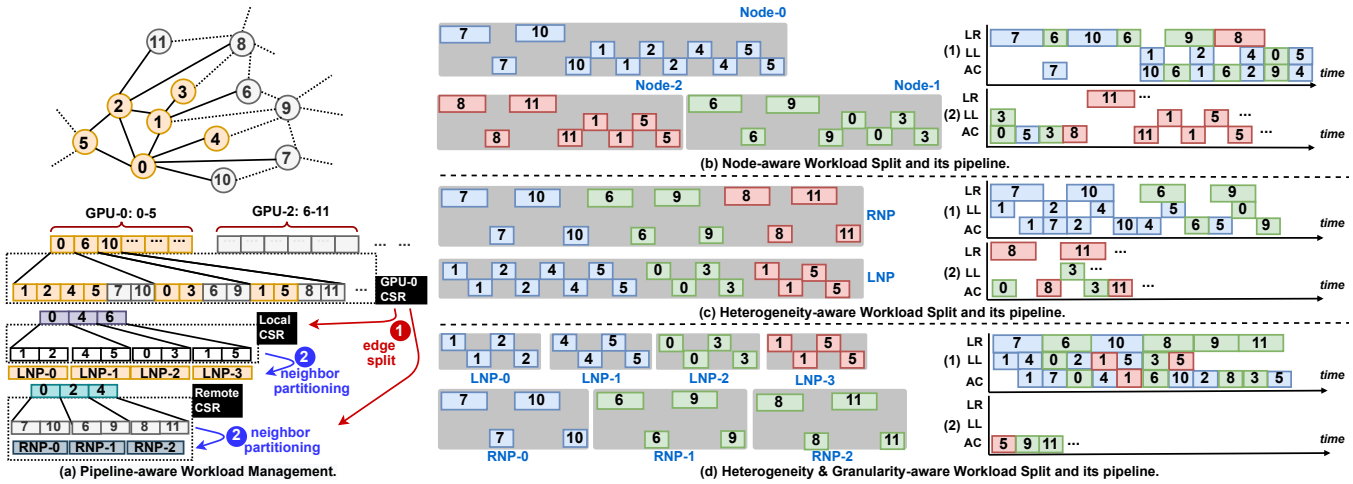


Figure 3: (a) Pipeline-aware workload management. "LNP"/"RNP" indicate local/remote workload partitions. (b)(c)(d) Different strategies of workload decomposition and pipelining. Each box indicates a certain (local/remote) aggregation workload and its length indicates its relative latency. "LR": loading remote neighbors, "LL": loading local neighbors, "AC": aggregation computation. Each grey rectangular shadow indicates a workload partition to be processed by one GPU processing unit. (1) and (2) indicate that the same pipeline is chunked into two parts along its time axis due to space limitations.

contrast, DNN pipelines can only overlap layer-wise computation and communication operations among different GPUs.

With the above insights, we propose a three-stage dynamic software pipeline design. The three stages include *loading remote neighbors* (**LR**), *loading local neighbors* (**LL**), and *aggregation computation* (**AC**). Aggregation of a certain neighbor will only take two stages. The remote neighbor aggregation will take the stage LR and AC while local neighbor aggregation will take the stage LL and AC. The stage-wise pipelining is achieved with two steps: 1) assigning aggregation workload to different GPU logical processing units (LPUs), like warps and blocks, and 2) scheduling different LPUs on the same GPU SM to overlap their execution. Three-phase pipeline can generalize to different GNN models, which essentially consist of the different numbers of basic remote and local operations. For example, GCN has a lower local-vs-remote operation ratio while GAT features a higher local-versus-remote operation ratio. Three-phase pipeline can also capture differences among inputs. For instance, a more sparse graph will have a higher remote-to-local operation ratio.

However, the direct construction and execution of such three-stage pipelines would be inefficient, because of its ignorance of GNN workload heterogeneity and irregularity on multi-GPU platforms. To address these challenges, MGG highlights a GNN-tailored pipeline construction strategy to build and optimize the software pipeline in three steps.

Step-1: Workload-aware inter-GPU pipeline workload balancing. This step aims to construct the "raw" pipeline and balance workloads among pipelines on different GPUs. Our insight is that GPUs with massive processing units (e.g., SMs) will serve many pipelines concurrently, and the key to maximizing GPU performance and utilization is to ensure that each pipeline will get a similar amount of workload, thereby avoiding execution critical path on certain "long" pipelines. We, therefore, develop a *range-constrained binary search algorithm* (Algorithm 1) based on prior graph partitioning exploration [3]. Our solution features a lower runtime cost to split the GNN input graph into chunks (one chunk per GPU) while balancing the number of edges within each chunk. Then the workload from the same chunk is grouped by nodes as *workload partitions* mixed local and remote neighbors (Figure 3(b)). From its potential execution pipeline, we can see many idle cycles (indicated by blank spaces in different pipeline stages) which would result in low pipeline efficiency and GPU resource occupancy. Note that in the software pipeline, workloads from different partitions can be overlapped as they will be processed by different LPUs. While the workloads from the same partition are sequentially processed by one LPU and their relative order should be maintained even after being mixed with other partitions.

Step-2: Heterogeneity-aware pipeline bubble reduction. The pipeline constructed from the previous step is still inefficient due to its scattered workloads among stages, namely pipeline bubbles. The optimization in this step is to minimize such pipeline bubbles for better pipeline efficiency. The key is to reduce the heterogeneity of workload partitions that hinders effective overlapping. To achieve this, we categorize the sparse multi-GPU GNN computation into two types. The first type has local neighbor access only, which has shorter execution latency. The second type has remote neighbor access, which features high latency overhead. We delicately handle different types of workloads via grouping (Figure 3(a)-❶), where two separate CSRs for *local* and *remote* subgraphs will be built. The aggregation will be conducted on local and

Algorithm 1: Range-constrained Binary Search.

```
input : Graph node pointer array (nPtr), edge list array
        (eList) , and the number of GPUs (numGPUs).
output: list of graph edge split points (numGPUs − 1).
outList = {};
lastPos = 0;
/* Compute approximated #edges per GPU.                    */
ePerGPU = (len(eList) + numGPUs − 1)/numGPUs;
for sId in [0, 1, ..., numGPUs − 1] do
    nid = binSearch(nPtr, ePerGPU, lastPos, numNodes);
    lastPos = nid;
    outList[sId] = nid;
end
return outList;
/* Search split points on nPtr.                            */
Function binSearch(nPtr, ePerGPU, lastPos,
  numNodes):
    i = lastPos;
    j = numNodes;
    target = min(nPtr[i] + ePerGPU, nPtr[numNodes]);
    while i < j do
        mid = (nPtr[i] + nPtr[j])/2;
        if mid > target then
            j = (i + j)/2;
        else
            i = (i + j)/2;
    end
    return i;
```

remote subgraphs separately and followed by a result synchronization at the end. Such a remote-local split is also backed by the fact that on platforms with all-to-all GPU interconnections (e.g, DGX-A100/H100), accessing different GPUs under the same data granularity has approximately equal communication cost [24]. Such heterogeneity awareness in workload partitioning (Figure 3(c)) enables a more densely overlapped workload between the stage LR and LL/AC.

Step-3: Granularity-aware intra-GPU pipeline enhancement. While the second optimization improves pipeline efficiency by reducing the workload heterogeneity, there is still plenty of room for further enhancement. The optimization in this step is to facilitate a more balanced workload distribution among pipeline stages. This key is to find the proper workload granularity for local and remote subgraphs so that those originally sequentially processed workload partitions can be overlapped. Our key observation is that nodes in the local/remote subgraphs would have a diverse number of neighbors. Such a specialty makes it challenging for massively parallel GPUs to harvest the real performance gains due to the imbalance workload and diverged execution flow. Therefore, we approximate such coarse-grained irregular workloads with fine-grained fixed-sized partitions so that the workload imbalance across nodes can be amortized. For example, with 2 neighbors per partition (Figure 3(a)-❷), we can get a more balanced workload among nodes in their local and remote neighbor aggregation. With such granularity awareness, the individual pipeline can be further condensed along its time axis with more overlapping of the LL and AC stage. (Figure 3(d)). Meanwhile, the irregular workload can be more evenly distributed to GPU SMs for higher GPU utilization. On the other side, partition granularity should also be balanced with synchronization overhead, since more fine-grained partitioning can bring more parallelism at the cost of more synchronization overhead. This is because workloads from different partitions for the same target node need to be reduced via synchronization, like inter-thread shuffling and atomics.

MGG design can also be generalized to multiple machines with a minor adaptation. For example, in Figure 3(d), when there are inter-node (over Inifite-Band) remote neighbors (longer latency due to lower inter-node communication speed), the size of remote neighbor partitioning (RNP) should be adjusted to a smaller size (e.g., from 2 to 1 remote neighbor) to facilitate better overlapping with local computation.

4.2 Hybrid GNN Data Placement

In collaboration with our multi-step pipeline construction, we introduce a *hybrid GNN data placement* strategy to exploit the benefits of different types of memory in SHMEM-enabled multi-GPU systems. The major impact of such hybrid placement on pipelining is two-fold. First, placing GNN data in different memory spaces will lead to different ratios of local and remote workloads, thus, affecting workload balance among pipelines. Second, different memory spaces will offer different access performances (e.g., latency), thereby, affecting the execution efficiency of the individual pipelines, such as the number of pipeline bubbles.

Our strategy focuses on two major aspects. Firstly, for workload balance among pipelines, we leverage NVSHMEM "shared" global memory to store the node embeddings (**NEs**) of the whole graph (Figure 4 *left*). Our major consideration here is that such shared global memory space can be accessed by all GPUs with the approximated equal access speed, which is vital to facilitate a more even distribution of remote workloads to GPUs in terms of their size and unit access costs. In addition, NEs are generally large in terms of size (due to high dimensionality), which are beyond the device memory limit of a single GPU. Therefore, NEs are ideal to be placed in shared global memory space with sufficient space (with aggregated memory of different GPUs), which also provides direct remote access support across GPUs. Specifically, we will partition the NEs of input graphs into n equal-sized partitions (where n is the number of GPUs) and place each of them in one GPU's shared global memory space.

Secondly, for the efficiency of individual pipelines, we allocate the "private" global memory space for storing partitioned graph structure (**GP**) data, which is only visible to kernels on the current GPU. Our key insight is that GP (e.g., edge lists), is all scalar values and usually small in size, and will only

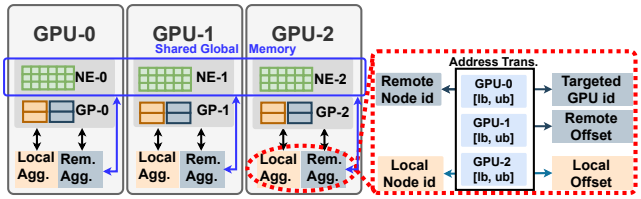


Figure 4: MGG Storage Layout and Communication Pattern. Note that "NE-i" is the node embedding partition stored on the i-th GPU. "GP-i" is the neighbor partition processed by the i-th GPU. "GPU-i [lb, ub]" is the node-id range [lowerbound, upperbound] of the node embeddings on the i-th GPU.

be accessed by the local GPU. Therefore, GP is ideal to be placed in individual GPUs' DRAM. Such a placement is also important to reduce unnecessary and inefficient remote access on those tiny scalars for fewer pipeline bubbles. In our design, GP data (e.g., edges) from private GPU global memory will be processed by a *address translation* unit for fetching correct NEs on local/remote GPU since the NE indices are rebased to zero on each GPU (Figure 4 *right*).

5 GPU-aware Pipeline Mapping

Efficient pipelining also demands effective mapping of well-constructed pipeline workload and their schedules to the low-level GPU logical processing units (*e.g.*, GPU threads/warps/blocks) to overlap computation and communication. To achieve this, we propose *Warp-based Mapping & Pipelining* and *Specialized Memory Design & Optimization* to jointly optimize the pipeline execution efficiency, GPU utilization, and end-to-end design flexibility.

5.1 Warp-based Mapping & Pipelining

An effective pipeline mapping demands comprehensive consideration of two major aspects. 1) *Which type of GPU logical processing units (e.g., warps, blocks) should be used for pipeline workload partitions?* We choose GPU *warp* as the basic working unit to handle the workload of each partition. This is because threads in a warp can collaboratively work on different dimensions of a node embedding simultaneously. Whereas using a single or several threads (less than the size of a warp, 32 threads) would hardly explore the computation parallelism and would cause warp-level divergence. Besides, NVSHMEM remote access initiated by a warp of threads would merge the requests into one remote memory transaction to amortize the overhead. 2) *Which pattern of mapping should be used for benefiting pipeline execution efficiency?* The most straightforward way is to continuously map the neighbor partitions from the local and remote workload list to GPU warps with continuous IDs (Figure 5). However, this strategy would easily suffer from workload imbalance among GPU SMs. This is because warps with continuous IDs are

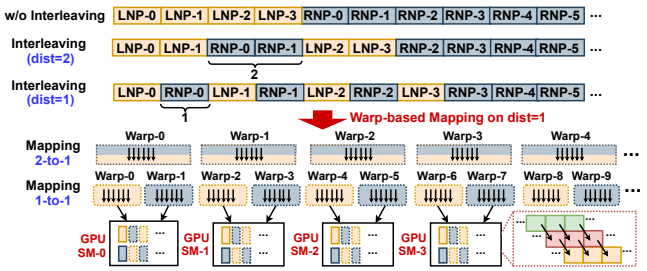


Figure 5: Warp-based Mapping and Pipelining. Note that "LNP" refers to the local neighbor partitions; "RNP" refers to the remote neighbor partitions. Workload and Warps are matched based on colors. Tiny boxes in GPU SM indicate decomposed workload operations for overlapped execution.

more likely to be placed into the same thread block, which is assigned to one SM for processing. Therefore, SMs assigned with warps for handling remote neighbor partitions would lead to much longer latency than SMs assigned with warps for processing local neighbor partitions. Such a workload imbalance would lead to poor GPU utilization and runtime execution performance.

To this end, we introduce our novel *workload interleaving* strategy to balance the workload among SMs on GPUs. Each warp of threads running on GPU would handle one or more pairs of local/remote workload partitions. To more precisely calibrate the warp-to-SM mapping for different pipeline stages to achieve efficient pipelining, we introduce a new metric – *interleaving distance*. We give examples with the interleaving distance equals 1 and 2 for illustration (Figure 5). By mixing different types (both local and remote) of workload together, better GPU utilization can be achieved since when one warp is blocked for high-cost remote access, other warps that are working on local computation can still be served by the SMs warp scheduler for filling up these idle GPU cycles. Moreover, such a design would improve design flexibility. For instance, given an input graph with a selected neighbor partition size, we can adjust the size of interleaving distance and the workload per warp so that waiting cycles of the remote access can be hidden by the computation cycles of the neighbor aggregation. Thus, each warp can be fully utilized while the design can achieve sufficient parallelism.

MGG currently processes the neighbors of adjacent nodes (based on node-ids) to the same thread block where the same block will be scheduled on the same SM. If there are common remote neighbors for those adjacent nodes, their remote requests will be merged. Improving such locality requires reordering the graph nodes to maximize their common neighbors. Such an exploration is orthogonal to our current contribution. In future GPUs, there is a trend to explore the locality among independent processing units. For instance, in Hopper, several thread blocks can be grouped together as thread-block groups. We can explore the tradeoff between the locality benefits and group synchronization overhead.

5.2 Specialized Memory Design & Optim.

Efficient software pipelining also demands careful management of high-bandwidth shared memory for promoting data access efficiency and asynchronized primitives for exploiting intra-warp operation pipelining.

GPU SM Shared Memory Layout: Based on our MGG's warp-based workload design, we propose a *block-level shared memory orchestration* to maximize the performance gains. We have several key insights for such a dedicated memory layout design within each thread block. *First*, our neighbor-partition-based workload will generate the intermediate results that can be cached at the high-speed shared memory for reducing the frequent low-speed global memory access. *Second*, NVSHMEM-based remote data access demands a local scratch-pad memory (e.g., registers, shared and global memory) to hold the remote data for local operations.

For the *local* neighbor aggregation, we reserve a shared memory space with D (D is the embedding dimension) floating-point numbers for embeddings of the target node in each neighbor partition so that threads from a warp can cache the intermediate results of partial reduction in shared memory. For the *remote* neighbor aggregation, the shared memory space is doubled $2 \times wpb \times D$ (wpb is the warps per block). The reason is that we need the first half $wpb \times D$ for caching the partial aggregation results of each warp and the remaining for the remotely accessed neighbor embeddings. For each MGG kernel design, we will first identify the warp-level information, like warp IDs. Then within each thread block, we define the customized shared memory layout by splitting the contiguous shared memory address into three different parts for neighbor ids, partial aggregation results, and the remotely-fetched node embeddings. We use the dynamic shared memory for design flexibility since those parameters (*e.g.*, wpb and D) can only be determined at runtime. During execution, we will first calculate the total shared memory size per block and then pass it as a kernel launching parameter.

Pipelined Memory Operation: §5.1 have discussed assigning local (LNP) and remote (RNP) neighbor aggregation workloads to warps so that different warps can overlap their computation and communication to fully saturate the active cycles of the GPU SM scheduler. However, only exploiting the inter-warp communication-computation overlap is not enough to maximize the utilization of GPU resources. We further explore the overlapping of the computation and communication at the intra-warp level by carefully scheduling the memory operations. Figure 6(a) shows the case with two LNPs and two RNPs by using the synchronized remote access, we can just sequentially process the two LNPs and the two RNPs. The long-latency remote access can happen only after the completion of its preceding LNP. This could lead to a longer GPU stall for memory operations and low GPU SM utilization. Our profiling also shows that without overlapping, the remote access usually dominates the overall execution

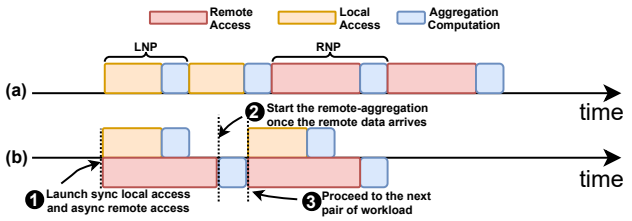


Figure 6: Illustration of (a) w/o and (b) w/ asynchronized primitives for overlapping computation and communication of an individual warp. Note that the length of each rectangular box indicates the estimated latency cost of each operation.

(around 60% of overall latency) compared to the time for local data access plus the time for aggregation computation (around 40% of overall latency). Such observation justifies our design to mainly hide the latency from remote access.

To amortize the cost of remote access for each warp, we introduce *asynchronized remote memory operations* (Figure 6(b)). This improved design consists of two major steps. First, we can simultaneously launch the local memory access while initializing the remote memory access for fetching the node embedding (❶), therefore, the time for remote access can be amortized by the processing of LNP. Second, once the remote access is completed, the current warp will start aggregation on the remotely-fetched node embedding data (❷). The next step will start the new iteration of the previous two steps, which will process a new pair of LNP and RNP.

6 Intelligent Runtime Design

In this section, we will discuss our intelligent runtime design with performance/resource analytical modeling and heuristic-based cross-iteration optimization strategy.

Performance-Resource Analytical Modeling: The performance/resource model of MGG has two variables: *workload per warp* (*WPW*) and *shared memory usage per block* (*SMEM*), which can be measured by

$$\begin{aligned} WPW &= 2 \cdot ps \cdot D \cdot dist, \\ SMEM &= ps \cdot wpb \cdot IntS + 2 \cdot wpb \cdot D \cdot FloatS \end{aligned} \tag{1}$$

where ps, wpb, and D are the sizes of neighbor partition, warp per block, and node embedding dimension, respectively; $dist$ is the interleaved distance of local/remote workloads (§5.1); $IntS$ and $FloatS$ are both 4 bytes on GPUs. To determine the value of the ps, wpb, and $dist$ of a given input graph, we will first compute the total number of warps by using

$$numWarps = \frac{\max\{local, remote\}}{dist} \tag{2}$$

where $local$ and $remote$ are the number of local and remote partitions, respectively. Then we compute the total number of

blocks and the estimated block per SMs by using

$$numBlocks = \frac{numWarps}{wpb}, \quad blocksPerSM = \frac{numBlocks}{numSMs} \tag{3}$$

Later, based on our micro-benchmarking results on diverse datasets, we define our parameter search space and constraints: 1) $ps \in [1 \ldots 32]$ to balance the computation parallelism and synchronization overhead; 2) $dist \in [1 \ldots 16]$ to effectively overlap the computation and remote memory access; 3) $wpb \in [1 \ldots 16]$ to maintain SM warp scheduling flexibility for better occupancy and throughput; 4) $numSMs \leq c_1$, $SMEM \leq c_2$, where c_1 and c_2 are hardware constraints [48], *e.g.*, NVIDIA A100 has 108 SMs and 164KB shared memory per SM.

Heuristic-based Cross Iteration Optimization To optimize the design of MGG, the parameter *ps*, *dist*, and *wpb* are initialized as the value 1 at the beginning. Then we optimize one parameter in each of the following iterations. *First*, we increase the *ps* to maximize the warp utilization. When further increasing the *ps* would also increase the latency, we would stop the search on *ps* and switch to *dist*. *Second*, we apply a similar strategy to locate the value of *dist* that can maximize the overlap of local computation and remote access. *Third*, we increase *wbp* to maximize the utilization of the entire SM. If any increase of *wpb* would increase the latency, we know that there may be too large thread blocks or too heavy workloads on individual warps that lower SM warp scheduling efficiency or computation parallelism. We would "retreat" (*i.e.*, decrease) *ps* to its second-highest value if necessary and restart the increase of *wpb*. This optimization algorithm will stop when any decrease of *ps* and increase of *wpb* would lead to higher latency than the top-3 lowest latency. The latency of each iteration during the optimization will be recorded by a configuration lookup table. Finally, the configuration with the lowest latency will be applied.

This particular optimization order of parameters (*ps*, *dist*, and *wpb*) is based on two major aspects: (i) *Spatially speaking*, the granularity is from coarse-grained algorithm-level partitioning through *ps*, to medium-grained pipeline construction through *dist* (according to the partition plan), to fine-grained pipeline-to-warp fine-tuning through *wpb* (according to the pipeline design). (ii) *Temporally speaking*, the three optimizations are applied at loading-time (*ps* to decide layout), kernel initialization (*dist* to decide pipeline), and runtime (*wpb* to decide pipeline mapping), respectively.

The above parameter adaption for dynamic pipelining is vital for design/optimization generality. This is because the characteristics of graphs (#nodes/edges and embedding sizes) would lead to different efficiency of kernel pipelines. Our later experimental studies (as shown in Figure 11) demonstrate its benefits with up to 70% of performance improvements.

Table 1: Datasets for Evaluation.

Dataset	#Vertex	#Edge	#Dim	#Class
reddit(**RDD**) [45]	232,965	114,615,892	602	41
enwiki-2013(**ENWIKI**) [23]	4,203,323	202,623,226	300	12
it-2004 (**IT04**) [10]	41,291,594	1,150,725,437	256	64
ogbn-paper100M(**PAPER**) [12]	111,059,956	1,615,685,872	128	64
ogbn-products(**PROD**) [17]	2,449,029	61,859,140	100	47
ogbn-proteins(**PROT**) [17]	132,534	39,561,252	8	112
com-orkut(**ORKT**) [23]	3,072,441	117,185,083	128	32

7 Evaluation

Benchmarks & Datasets Despite the diversity of GNN models, the fundamental computation and communication paradigm (vector-based scatter-gather operation) in multi-GPU GNNs remains the same. We evaluate two distinctive and representative GNN models on *node classification* tasks:

The first type of GNN model uses a *non-discriminated* neighbor aggregation strategy, where all neighbors contribute equally when doing the aggregation. We choose **Graph Convolutional Network (GCN)** [21], which is the most popular GNN model and is also the key backbone network for many other GNNs, such as GraphSAGE [16] and Differentiable Pooling [52]. We use *2 layers with 16 hidden dimensions* for GCN, which is also the setting from the original paper [21]. The computation of a 2-layer GCN can be expressed as

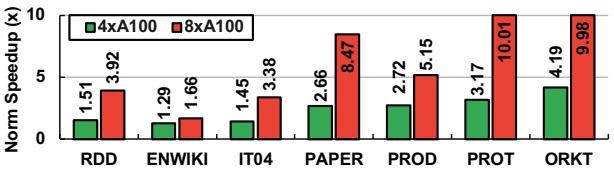
$$Z = Softmax(\hat{A}\ ReLU(\hat{A}XW^1)W^2). \tag{4}$$

where $\hat{A}$ is the adjacent matrix of the input graph with self-loop edges, and X is the input node embedding matrix, where $X \in R^{N \times D}$; N is the number of nodes in a graph; D is the size of node embedding dimensions. W^1 and W^2 are trainable weight matrices in layer-1 and layer-2, respectively.

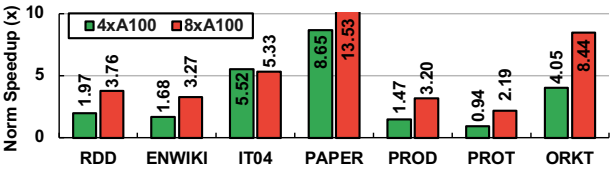
The second type uses a *discriminated* neighbor aggregation strategy, where neighbors would contribute differently depending on their calculated edge-specific features. We choose **Graph Isomorphism Network (GIN)** [49], which aims to distinguish the graph structure that cannot be identified by GCN. Each layer of GIN can be expressed as

$$h_v^{l+1} = MLP^l((1+\varepsilon^l)\dot{h}^l + \sum_{u \in N_{(v)}} h_u^l). \tag{5}$$

where l is the layer ID and $l \in \{0,1\}$, *MLP* is a fully-connected neural network, h_v is the node embedding for node v, and $N_{(v)}$ stands for the neighbors of node v. GIN mainly differs from GCN in its aggregation function, which introduces a weight parameter as the ratio of contribution from its neighbors and the node itself. In addition, GIN is the reference architecture for many other advanced GNNs with more edge properties, such as Graph Attention Network [43]. For GIN evaluation, we use *5 layers with 64 hidden dimensions*, which is also the setting used in the original paper [49]. Graphs (Table 1) used in our evaluation are large in their number of nodes and edges that demand multi-GPU capability for effective GNN computation. *#Class* is the output dimension



(a) GCN Model.



(b) GIN Model.

Figure 7: Performance comparison with DGL. Note that full-graph PAPER on DGL requires A100-80GB.

(*#labels*) for the node classification task. *#Dim* is the embedding dimension of the input graph.

Baselines In this evaluation, we compared MGG with several existing systems that support large full-graph GNN (i.e., caching the entire graph on GPUs) on multi-GPU platforms. **1) Deep Graph Library (DGL)** [45] is the state-of-the-art framework for large-scale GNNs across GPUs. It leverages PyTorch-Direct [27] as the communication backend for GPU-initiated zero-copy memory access [41] to fetch neighbors embedding from the CPU host. **2) MGG-UVM** [20] is a GNN design by adapting MGG to leverage unified virtual memory (UVM). UVM has been highlighted in handling irregular graph computations (such as PageRank) on large graphs [20]. However, [20] is not open-sourced, we thus generalize the pipeline kernel designs and optimizations (§4 and §5) of MGG to build such a UVM baseline and incorporate optimizations from [20]. Note that UVM and zero-copy memory are different communication backends [1]. Thus, MGG-UVM does not implement zero-copy data transfer. We remark UVM is the key communication protocol before the new hardware support for fine-grained direct GPU-GPU communication (e.g., NVSHMEM). UVM is more coarse-grained and will require the engagement of CPUs (e.g., host memory management) for communication. The reason to use MGG-UVM is to show that if there is no advanced hardware support (e.g., NVSHMEM) for fine-grained direct GPU-GPU communication, the benefits of our elaborated pipeline can be offset by UVM communication overhead. **3) ROC** [18] is a popular distributed multi-GPU system for full-graph computation. ROC highlights its learning-based partitioning and leverages NVIDIA Legion [5] runtime for communication and task scheduling.

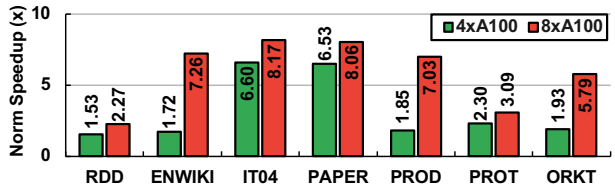
Other multi-GPU GNN designs, like NeuGraph [26] and *P*3 [12], are not publicly available. Initially, we plan to evaluate MGG on AMD ROC_SHMEM [2]. However, as indicated in its document, the existing ROC_SHMEM is an experimental prototype and is not officially ready to be applied in practice due to very strict software limitations (*e.g.*, only supports ROCm v4.3) and hardware (*e.g.*, only supports AMD GFX9 GPUs), which are quite challenging to find and deploy and not supported by any existing GNNs frameworks [6, 18, 28] for comparison. We believe that once ROC_SHMEM becomes ready and generally applicable, MGG can be easily migrated to AMD multi-GPU platforms.

There is no existing design that can leverage GPU-to-GPU communication only for distributed full-graph GNN computation. We try our best to measure the best-possible baseline performance. DGL and ROC have longer latency in the earlier iteration due to cache warmup for node embedding on GPU memory. We thus perform warm up iterations until their per-iteration latency becomes stable, and then measure their performance with minimized CPU-GPU data movements.

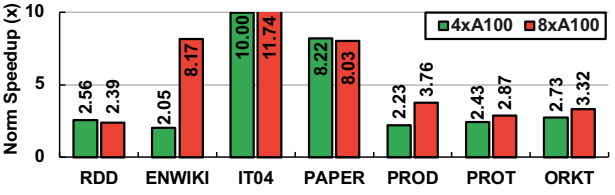
Platforms & Tools The implementation of MGG consists of ∼9K LoC. We compile and link MGG with CUDA (v11.2), OpenMPI (v4.1.1), NVSHMEM (v2.0.3), and cuDNN (v8.2) library. Our major platform is an NVIDIA DGX-A100 with dual AMD Rome 7742 processors (each with 64 cores, 2.25 GHz), 1TB host memory, and 8×A100 GPUs (40 GB) connected via NVSwitch, which offers 600 GB/s GPU-to-GPU bi-directional bandwidth. For the modeling study, we also leverage DGX-1 with 4×V100 GPUs connected via NVLinks. We use NVIDIA NSight Compute to get the kernel-level profiling metrics. Speedup is averaged over 100 runs.

7.1 End-to-End Performance

Compared with DGL In this section, we will compare with the state-of-the-art DGL framework, which leverages PyTorch-Direct for cross-GPU communication. We evaluate different datasets and platform settings (with 4 and 8 A100 GPUs). As shown in Figure 7, MGG outperforms DGL with averaged 4.25× and 4.57× speedups on GCN and GIN models, respectively. We also notice a trend that MGG demonstrates a more pronounced speedup with more GPUs. With the increasing number of GPUs, DGL suffers from heavy memory access contention, since multiple GPUs are initiating massive requests to access the neighbor embeddings on the CPU host memory. Another observation is that on GIN ($D = 64$) with higher hidden dimensionality for smaller datasets (e.g., PROD and PROT), the performance gap between DGL and MGG is smaller compared to GCN ($D = 16$) since as indicated in [28], zero-copy memory would be beneficial from more coarse-grained data movement (with larger embedding vector) that can saturate the PCIe cache line (128 Bytes). While such an advantage of DGL diminishes for those larger datasets (e.g., IT04 and PAPER) on GIN due to significantly increased sparsity and irregularity. In addition, compared with MGG, DGL assumes the one-size-fits-all communication strategy would work well for all input datasets. Therefore, it ignores the importance of the inputs and hardware properties, which would bring non-trivial (more than 30%) benefits (§7.2).



(a) GCN Model.



(b) GIN Model.

Figure 8: Performance comparison with MGG-UVM.

Table 2: Additional performance comparison of MGG and DGL on GraphSAGE and GAT.

Model	RDD	ENWIKI	IT04	PAPER	PROD	PROT	ORKT
SAGE	4.97×	1.76×	1.99×	3.53×	7.05×	3.39×	3.53×
GAT	2.65×	1.62×	2.06×	3.04×	2.06×	3.39×	3.04×

MGG can also be extended to cover other GNN models. The following results show the speedups of MGG over DGL on GraphSAGE with layerwise node neighbor sampling and GAT with dot-product edge attention. Table 2 shows that the performance results of GAT and SAGE also agree with our prior observations on the GCN and GIN, demonstrating the generality and effectiveness of our proposed design and optimizations to handle more complex dataflow (e.g., edge attention and softmax) in multi-GPU GNN computation.

Despite that MGG (NVSHMEM) and DGL (with CPU-GPU zero-copy memory [41]) both rely on GPU-initiated communication and overlap communication with computation, their underlying mechanism is different, and MGG shows more performance advantages. MGG can leverage inter-GPU communication while DGL can only rely on CPU-GPU communication with limited bandwidth. This makes the communication costs pronounced in DGL and offsets the performance gains from massive thread-level parallelism. This experiment also shows that MGG can serve as a drop-in replacement for the existing communication backend of DGL to improve large-scale full-graph GNN computation.

Compared with MGG-UVM In this experiment, we compare MGG with its UVM-based counterpart, MGG-UVM, which uses UVM in place of NVSHMEM for remote communication. Figure 8 shows that MGG achieves 4.58× speedup and 5.04× speedup on average compared to MGG-UVM on GCN and GIN, respectively. The MGG-UVM leverages the page-faulting-based remote data access that is more coarse-grained (around 4 KB) in comparison with a single node embedding size (less than 0.4KB), which leads to higher overhead and lower effective bandwidth usage per embedding transfer. Such an overhead would exacerbate with more GPUs and also make MGG-UVM challenging for GPU SM schedulers to effectively dispatch instructions for the next available warps. This is mainly because most of the warps wait for the long-cycle page-faulting and migration.

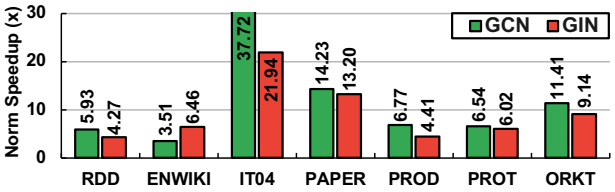


Figure 9: Performance comparison with ROC with 8×A100.

We notice that with the increase of the dimension size (i.e., data movement granularity), the speedup over MGG-UVM becomes higher. We later found out that the increase of data-movement granularity actually increases the overall page-fault counts. This is because embedding vectors are generally stored continuously for memory efficiency instead of aligning with the size of memory pages. Therefore, increasing the size of individual embedding also increases the likelihood of triggering multiple pagefaults per embedding transfer.

Comparing among datasets, for graphs (e.g., PAPER) with more nodes/edges and lower average node degree, MGG would demonstrate more speedups since these graphs exhibit more irregular and sparse access that can not well fit into regular fix-sized pages. This also indicates the importance of amortizing communication overhead. Thanks to pipeline-centric workload management, we can effectively amortize such costs with careful operation scheduling.

We further measure two performance-critical GPU kernel metrics that are the key indicators of our pipeline efficiency (§4.1): *Achieved Occupancy* (the ratio of the average active warps per active cycle to the maximum number of warps supported in an SM) and *SM utilization* (the utilization of all available SMs on a single GPU). MGG improves SM utilization (by 21.15% on average) and occupancy (by 39.20% on average) compared to MGG-UVM. This indicates that MGG can effectively 1) distribute irregular workloads to SMs to balance workloads among pipelines and improve the overall GPU utilization, and 2) overlap the remote access and local aggregation computation from different warps to reduce pipeline bubbles and maximize SM occupancy.

Compared with ROC In this experiment, we compare MGG with ROC [18] on their officially released GCN model implementation. We originally plan to evaluate both 4 and 8 GPU settings. However, ROC reports many out-of-memory (OOM) errors for those large graphs on GCN/GIN model and medium graphs on the GIN model due to its aggressive caching of those intermediate tensors on GPUs. Therefore, we keep our comparison to 8 GPUs. Performance-critical ROC runtime configurations (e.g., #CPU cores, GPU/host memory size) are optimized to fully utilize the DGX-A100.

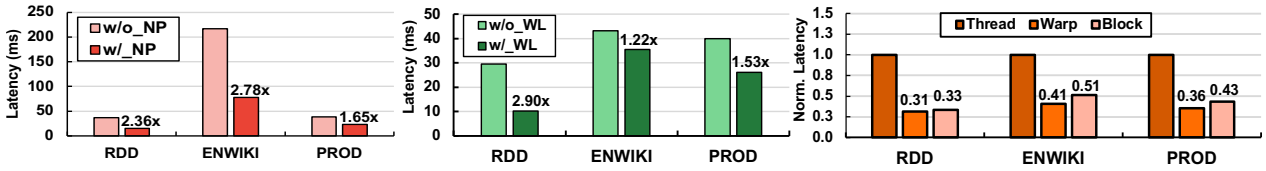


Figure 10: Optimization Analysis: (a) Neighbor Partitioning; (b) Workload Interleaving; (c) Choice of Communication Primitives.

Figure 9 shows that MGG achieves averaged 12.30× and 9.35× speedups over ROC on GCN and GIN, respectively. MGG demonstrates a more pronounced speedup over ROC on the larger graph (e.g., IT04 and PAPER), which has more irregular neighbor embedding access. The Legion runtime of ROC relies on the DMA engine for bulky data (batched embeddings) transfer between host and GPU memory, leading to higher throughput but inferior latency performance. Besides, ROC relies on a separate communication-computation design, where computation happens after the full completion of communication. Such a design eliminates the opportunity to fill idle GPU cycles with computation during communication. In addition, the learning-based partitioning (to reduce communication) of ROC shows benefits on relatively smaller datasets (e.g., RDD and PROT) but hard to find optimal partition plans for large graphs due to the input structure complexity.

7.2 Optimization Analysis

Neighbor Partitioning (NP) We compare MGG with a baseline design without applying the neighbor partitioning technique (*i.e.*, each aggregation workload consists of all local/remote neighbors) on 4×A100. We apply the workload interleaving for both implementations and fix the warp-per-block size to 2 to eliminate the impact from other performance-related factors. Figure 10(a) shows higher latency (averaged 2.26×) for designs without applying neighbor partitioning, since the workload imbalance becomes more severe across different warps without neighbor partitioning, especially for those graphs with many remote access demands, leading to limited computing parallelism and GPU underutilization.

Workload Interleaving (WL) We compare MGG with a baseline design without workload interleaving (*i.e.*, remote neighbor aggregation and local neighbor aggregation are mapped separately to the GPU warps. We fix the neighbor partition size to 16 and the warp-per-block size to 2. Figure 10(b) shows that MGG consistently outperforms the non-interleaved baseline with an average of 1.89× speedup. Without interleaving the local/remote workload, the workload distribution would be highly skewed, where the heavy and intensive remote aggregation would be gathered on certain warps close to each other while the lightweight local aggregation would be gathered on some other warps close to each other. This leads to inefficient warp scheduling and higher latency.

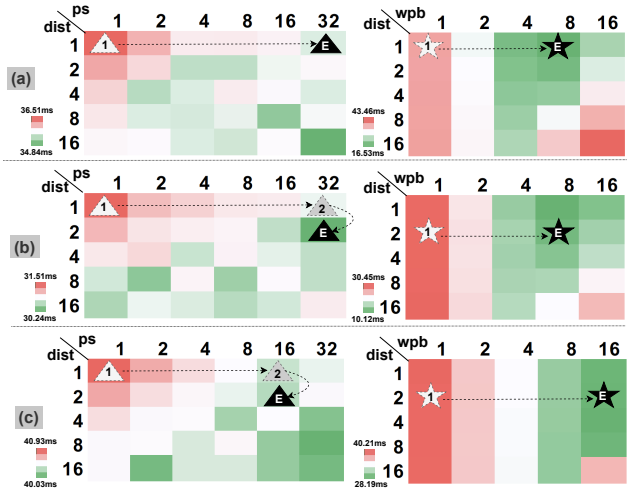


Figure 11: Parameter selection for three different settings. (a), (b), and (c) are for setting I, II, and III, respectively. Note that the left-side figures show the runtime latency for different combinations of *ps* and *dist*, while the right-side figures show the latency for different combinations of *wpb* and *dist*. The solid black triangle with "E" is the searched "optimal" combination for *ps* and *dist*, while the black solid star with "E" is the searched "optimal" *wpb* given *dist* and *ps*.

Communication Primitives We adopt MGG with different NVSHMEM primitives at the *thread*, *warp*, and *block* levels. We fix the number of GPUs to 2, the hidden dimension to 16, the neighbor partition size to 2, and the distance of workload interleaving to 2. Figure 10(c) shows that warp-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_warp_get`) for remote accessing can bring the lowest latency. For thread-level NVSHMEM primitives (*e.g.*, `nvshmem_float_get`), it would not coalesce the remote memory access to reduce unnecessary transactions. For the block-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_block_get`), the higher overhead comes from collaborating a block of threads for remote access, since thread blocks (usually consisting of multiple warps) is larger than a single warp, thus, leading to higher synchronization and scheduling cost. This study also shows that our choice of warp-level primitives strikes a good balance between memory access efficiency and scheduling flexibility.

Modeling and Optimization We further analyze the effectiveness of our lightweight analytical model for design space

Table 3: Accuracy-Latency of GNNs w/ and w/o sampling.

Dataset	Accuracy w/ sampling	Accuracy w/o sampling	Latency (w/o *vs.* w/ sampling)
RDD	0.937	0.957	1.07×
PROT	0.776	0.825	1.25×

search. Specifically, three key parameters are studied, the size of neighbor partitioning (*ps*), the interleaving distance (*dist*), and the warps per block (*wpb*). We consider three different settings on a 2-layer GCN model: I: RDD on 4×A100 as the basic setting. II: RDD on 8×A100 to demonstrate the adaptability toward the different numbers of GPUs. III: RDD on 4×V100 [37] to demonstrate the adaptability toward the different types of GPUs. We decompose searching results into two parts corresponding to the output of the ***second*** and ***third*** steps of the optimization discussed in §6.

Figure 11 shows that our performance modeling and parameter selection strategy can pinpoint the low-latency design for the above three settings. The overall searching process only requires about 10 iterations to reach the final "optimal" settings. Note that here we show latency results for all possible settings for comparison. While in practice, we only need to traverse a small part of the whole design space (as indicated by the boxes touched by the dot lines). By comparing the final optimal runtime configuration setting and the initial configuration, we can see that modeling and cross-iteration optimization can decrease the execution time by up to 68%. In the end-to-end GNN training (usually more than 100 iterations), such a latency saving would also be significant.

7.3 Additional Study

Accuracy-latency Tradeoff This study will analyze the accuracy-latency tradeoff between GNNs with sampling and full-graph (w/o sampling) on 8×A100. Table 3 shows an evident node classification accuracy increase (2% to 5%) of GNN w/o sampling over GNN w/ sampling. The accuracy of sampling-based GNN would be affected by many factors (*e.g.*, sampling rate at each GNN layer and graph structure). It is thus highly tricky to choose the "optimal" value for those factors. Here we follow the conventional way for GNN sampling [45]. The accuracy difference agrees with previous GNN algorithmic work [16]. In many real-world applications (e.g, e-commerce), such an accuracy advantage of full-graph GNNs are be more preferred by users. Because even 1% accuracy would make significant profit gains when deploying services at scale while the latency penalty is relatively minor.

Generality to other applications The design of MGG can be generalized to other similar applications. We demonstrate the typical and popular deep-learning recommendation model (DLRM) [31,47,54] that has been widely used in the industry. In multi-GPU DLRM, the large embedding tables are partitioned by rows and stored in different GPUs. The DLRM inputs (embedding access queries) will request embeddings from tables on different GPUs and then apply operations (e.g., elementwise addition or dot product) on those fetched embeddings. Such embedding lookup is highly sparse and irregular and dominates (> 80% latency [15,54]) the overall DLRM computation. We improve the mainstream DLRM system [31] with the design and optimizations of MGG to accelerate embedding lookup and element-wise addition and compare with the original system (which relies on NCCL) [31] under 4-GPU settings on the popular Criteo Kaggle [9] dataset. Table 4 shows that DLRM with MGG effectively reduces the lookup time (2.64×). The fine-grained remote access of MGG can reduce redundant inter-GPU traffic by using NCCL and offset the cost by massively parallel GPU-initiated communication.

Table 4: DLRM [31] with MGG in Embedding Lookup.

Implementation	DLRM [31]	DLRM (MGG)
Time (ms)	315.27	119.66

8 Discussion

Deep Learning Pipelines: Despite the popularity of the pipeline concept in the conventional dense DL, the generalization of such a technique in sparse GNN computation is yet to be explored in-depth. PiPAD [44] overlaps the communication (CPU-to-GPU) and processing (on GPUs) between adjacent graph partitions. Adopting this strategy, we will get designs as Figure 3(c), which would still suffer from pipeline bubbles due to workload imbalance. vPipe [56] dynamically assigns a DNN layer to certain pipeline stages during the runtime. It improves pipeline efficiency and GPU utilization for DNN models. However, adopting this approach in our fine-grained kernel pipeline would incur high overhead due to frequent workload reassignment and context switching. In addition, the pipeline bubbles in dense DNN are predictable, input-agnostic, and can be reduced offline. However, the pipeline bubble for GNN can only be figured out at runtime due to input dependency. It, therefore, demands careful online workload balance and a pipeline schedule/mapping.

Graph Partitioning Strategies: Besides our current ID-based graph partitioning, our designs/optimizations could also be extended to support other graph partitioning strategies from prior graph processing and GNN work. There are several major categories. *1) Locality-driven partitioning* (e.g., Gemini [57] and Rabbit order [4]) minimizes the communication/synchronization cost in distributed graph processing/GNN computing. Such partition strategies are orthogonal to our current design optimization. Despite it will reduce the total size of communication, the communication pattern remains the same with irregular, sparse, and fine-grained data movements. Our MGG design can be modified to accommodate such reduced-communication cases through dynamic kernel re-configuration (e.g., fine-tuning the interleaving distance and warp-to-block mapping) to maximize

communication and computation efficiency. *2) Workload-driven partitioning* (e.g., NeuGraph [26] and CUBE [53]) balances the irregular graph/GNN workload among different devices. This type of strategy typically maintains multiple replicas of nodes and node properties on different devices and synchronizes partial results in replicas after local computation on each device. Our current design be adapted to handle such cases by inserting device synchronization primitives (NVSHMEM collective communication primitives, such as `nvshmem_float_sum_reduce`) for maintaining data consistency among different replicas. *3) Learning-based partitioning* (e.g., ROC [18]) dynamically learns an "optimal" partitioning strategy that can maximize the computation performance. Our current design/optimization can also support this partitioning strategy by incorporating the overhead of NVSHMEM remote memory access in the runtime prediction model when optimizing partitioning strategies online.

9 Conclusion

This paper presents MGG, a novel multi-GPU system design, and implementation to exploit the potential of leveraging GPU intra-kernel software pipeline for accelerating GNNs. MGG consists of GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping, and an intelligent runtime design to dynamically improve the GNN runtime performance. Experiments show the advantages of MGG over state-of-the-art solutions and its generality towards other DL applications.

10 Acknowledgment

We would like to appreciate the great help and support from OSDI shepherd and anonymous reviewers. This work was supported in part by NSF-2124039 and CloudBank [32]. We also appreciate the generous help and support from Amazon Faculty Research Award 2021 for Professor Yufei Ding and NVIDIA Graduate Fellowship 2022-2023 for Yuke Wang.

References

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, 2013.

[2] AMD. Rocm openshmem. https://github.com/ROCm-Developer-Tools/ROC_SHMEM.

[3] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.

[4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.

[7] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*, 2005.

[8] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations (ICLR)*, 2018.

[9] Criteo. Criteo display ad challenge. https://kaggle.com/c/criteodisplay-ad-challenge.

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 2011.

[11] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[13] Jaume Gibert, Ernest Valveny, and Horst Bunke. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition*, 2012.

[14] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.

[15] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia,

Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems (NeurIPS)*, 33, 2020.

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of the 3rd MLSys Conference, 2020*.

[19] Riesen Kaspar and Bunke Horst. *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[20] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[21] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

[22] Jérôme Kunegis and Andreas Lommatzsch. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data, 2014.

[24] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2019.

[25] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference (ATC), 2019*.

[27] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv preprint arXiv:2101.07956*, 2021.

[28] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *Proc. VLDB Endow.*, 2021.

[29] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[30] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning (ICML)*, 2021.

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[32] Michael Norman, Vince Kellen, Shava Smallen, et al. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing*. 2021.

[33] Nvidia. Dgx superpod. https://nvidia.com/en-us/data-center/dgx-superpod/.

[34] Nvidia. Nvidia collective communication library (nccl). https://developer.nvidia.com/nccl.

[35] Nvidia. Nvidia dgx a100. https://nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[36] Nvidia. Nvshmem communication library. https://developer.nvidia.com/nvshmem.

[37] Nvidia. Tesla v100. https://nvidia.com/en-us/data-center/v100/.

[38] NVIDIA. Unified memory for cuda beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[39] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

[40] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *The 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, 2008.

[41] Tim Schroeder. Peer-to-peer & unified virtual addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[42] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.

[43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[44] Chunyang Wang, Desen Sun, and Yuebin Bai. Pipad: Pipelined and parallel dynamic gnn training on gpus. *28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023.

[45] 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. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[46] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[47] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. El-rec: efficient large-scale recommendation model training via tensor-train embedding table. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[48] wikipedia. Nvidia gpu micro-architecture. https://en.wikipedia.org/wiki/CUDA.

[49] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[50] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[51] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[52] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *The 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[53] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[54] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 2019.

[55] Yufeng Zhang, Xueli Yu, Zeyu Cui, Shu Wu, Zhongzhen Wen, and Liang Wang. Every document owns its structure: Inductive text classification via graph neural networks. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[56] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, et al. vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2021.

[57] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

A Artifact Appendix

MGG is a holistic runtime for exploiting intra-GPU-kernel communication-computation pipelining to accelerate multi-GPU GNNs. MGG consists of two parts. The first part is the host-side CPU program. It is responsible for dataset loading, runtime configuration generation, and invoking the GPU-side program. The second part is the device-side GPU program, called kernels. It is responsible for the major computation and communication of the GNN model on sparse neighbor-aggregation across GPUs and dense node-update phase within each GPU. MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping.

- Code repository: **Github**[2] and **Zenodo**[3].
- **Hardware, OS & Compiler**:
  - NVIDIA DGX-A100 with dual AMD Rome 7742 processors (each with 64 cores, 2.25 GHz), 1TB host memory, and 8×A100 GPUs (40 GB) connected via NVSwitch (600 GB/s).
  - Operating systems: Ubuntu 20.04+.
  - Compilers: NVCC (v11.2), GCC (v7.5.0),
  - Libraries: CUDA (v11.2), OpenMPI (v4.1.1), NVSHMEM (v2.0.3), cuDNN (v8.2).
  - Datasets: SNAP [23] and OGB [17].

Environment Setup

Step-1: Download libraries and datasets.

– 1.1. Download libraries.

```
wget storage.googleapis.com/mgg_data/local.tar.gz
tar -zxvf local.tar.gz
tar -zxvf local/nvshmem_src_2.0.3-0/build_cu112.tar.gz
```

– 1.2. Download datasets and Setup Baselines.

```
wget storage.googleapis.com/mgg_data/dataset.tar.gz
tar -zxvf dataset.tar.gz
cd dgl_pydirect_internal/
wget storage.googleapis.com/mgg_data/graphdata.tar.gz
&& tar -zxvf graphdata.tar.gz
&& rm graphdata.tar.gz
cd ..
gsutil cp -r gs://mgg_data/roc-new/ .
```

– 1.3. Launch Docker for MGG.

```
cd docker
./launch.sh
```

– 1.4. Compile MGG implementations.

```
mkdir build && cd build && cmake .. && cd ..
./0_mgg_build.sh
```

[2] https://github.com/YukeWang96/MGG-OSDI23-AE.git
[3] https://doi.org/10.5281/zenodo.7853945

Step-2. Run Initial Tests.

Please try below Section-3.4 and Section-3.5.

Step-3: Experiments.

– 3.1. Compare with UVM (Fig.8a and Fig.8b).

```
./0_run_MGG_UVM_4GPU_GCN.sh
./0_run_MGG_UVM_4GPU_GIN.sh
./0_run_MGG_UVM_8GPU_GCN.sh
./0_run_MGG_UVM_8GPU_GIN.sh
```

Results can be found at `Fig_8_UVM_MGG_4GPU_GCN.csv`, `Fig_8_UVM_MGG_4GPU_GIN.csv`, `Fig_8_UVM_MGG_8GPU_GCN.csv`, `Fig_8_UVM_MGG_8GPU_GIN.csv`

– 3.2. Compare with DGL (Fig.7a and Fig.7b).

```
cd dgl_pydirect_internal/
./launch_docker.sh
cd gcn/
./0_run_gcn.sh
cd ../gin/
./0_run_gin.sh
```

Results of DGL can be found at `1_dgl_gin.csv` and `1_dgl_gcn.csv`. MGG reference is in `MGG_GCN_8GPU.csv` and `MGG_8GPU_GIN.csv`.

– 3.3. Compare with ROC on 8xA100 (Fig.9).

```
cd roc-new/docker
./launch.sh
```

Results can be found at `Fig_9_ROC_MGG_8GPU_GCN.csv`, `Fig_9_ROC_MGG_8GPU_GIN.csv`.

– 3.4. Compare NP with w/o NP (Fig.10a).

```
python 2_MGG_NP.py
```

Note that the results can be found at `MGG_NP_study.csv`.

– 3.5. Compare WL with w/o WL (Fig.10b).

```
python 3_MGG_WL.py
```

Note that the results can be found at `MGG_WL_study.csv`.

– 3.6. Compare API (Fig.10c).

```
python 4_MGG_API.py
```

Note that the results can be found at `MGG_API_study.csv`.

– 3.7. Design Space Search (Fig.11a).

```
python 5_MGG_DSE_4GPU.py
python 5_MGG_DSE_8GPU.py
```

Results can be found at `Reddit_4xA100_dist_ps.csv`, `Reddit_4xA100_dist_wpb.csv`, `Reddit_8xA100_dist_ps.csv`, `Reddit_8xA100_dist_wpb.csv`.