NeutronOrch: Rethinking Sample-based GNN Training under CPU-GPU Heterogeneous Environments

Xin Ai*
Northeastern Univ., China
aixin0@
stumail.neu.edu.cn

Qiange Wang*
National University of Singapore, Singapore
wang.qg@nus.edu.sg

Chunyu Cao
Northeastern Univ., China
chunyucao@
stumail.neu.edu.cn

Yanfeng Zhang
Northeastern Univ., China
zhangyf@
mail.neu.edu.cn

Chaoyi Chen
Northeastern Univ., China
chenchaoy@
stumail.neu.edu.cn

Hao Yuan
Northeastern Univ., China
yuanhao
@stumail.neu.edu.cn

Yu Gu
Northeastern Univ., China
guyu@
mail.neu.edu.cn

Ge Yu
Northeastern Univ., China
yuge@
mail.neu.edu.cn

ABSTRACT

Graph Neural Networks (GNNs) have shown exceptional performance across a wide range of applications. Current frameworks leverage CPU-GPU heterogeneous environments for GNN model training, incorporating mini-batch and sampling techniques to mitigate GPU memory constraints. In such settings, sample-based GNN training can be divided into three phases: sampling, gathering, and training. Existing GNN systems deploy various task orchestration methods to execute each phase on either the CPU or GPU. However, through comprehensive experimentation and analysis, we observe that these task orchestration approaches do not optimally exploit the available heterogeneous resources, hindered by either inefficient CPU processing or GPU resource bottlenecks.

In this paper, we propose NeutronOrch, a system for sample-based GNN training that ensures balanced utilization of the CPU and GPU. NeutronOrch decouples the training process by layer and pushes down the training task of the bottom layer to the CPU. This significantly reduces the computational load and memory footprint of GPU training. To avoid inefficient CPU processing, NeutronOrch only offloads the training of frequently accessed vertices to the CPU and lets GPU reuse their embeddings with bounded staleness. Furthermore, NeutronOrch provides a fine-grained pipeline design for the layer-based task orchestrating method. The experimental results show that compared with the state-of-the-art GNN systems, NeutronOrch can achieve up to 11.51× performance speedup.

PVLDB Reference Format:
Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, Ge Yu. NeutronOrch: Rethinking Sample-based GNN Training under CPU-GPU Heterogeneous Environments. PVLDB, 17(8): 1995 - 2008, 2024.
doi:10.14778/3659437.3659453

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AiX-im/Sample-based-GNN.

*Equal contribution.

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659453

1 INTRODUCTION

Graph Neural Networks (GNNs) [48] are a novel class of Deep Neural Networks (DNNs) designed to process graph-structured data. They have demonstrated remarkable effectiveness across various graph applications [2, 7, 8, 21, 38, 46, 54]. Recently, GPUs have been extensively used to accelerate GNN training owing to their high memory bandwidth and massive parallelism [15, 20].

Considering the growing sizes of graphs in real-world applications, full-graph GNN training that loads and trains the entire graph on the GPU is impractical due to the limited GPU memory capacity [17, 24, 33, 42, 44]. As a result, sample-based approaches that train on sampled subgraphs have emerged as a promising solution for training large graphs with limited GPU resources [11, 53, 56]. In this approach, the input data, including vertex features and graph structure, are stored in the host memory while the training process is offloaded to GPUs [43, 57]. The sample-based GNN training divides the training vertices into multiple individual computation units (i.e., batch). To train on each batch, the training process first samples the K-hop subgraphs for the training vertices, following a given neighbor sampling rule, then gathers the required vertex features based on the sampled subgraph and loads the subgraph and the involved vertex features onto the GPU. Finally, GPU performs training process and updates the corresponding model parameters.

This training mechanism, known as the **sample-gather-train** paradigm has gained widespread adoption in various GNN systems [3, 5, 9, 14, 23, 28–30, 36, 39, 40, 42, 43, 47, 50, 52, 55]. These systems decouple the three steps and deploy them on different computation devices to achieve high performance. Early systems [5, 23, 36, 42, 43] employ graph sampling and feature gathering on CPUs. These systems store the graph data in the CPU memory, perform CPU-based graph sampling, and collect the required features on the CPU side before transferring them to GPUs. This approach allows the GPUs to be dedicated solely to training tasks. On the other hand, some other systems [3, 19, 28, 39, 43, 47, 50, 52, 55] leverage GPU-accelerated graph sampling. They store the graph topology in either GPU or pinned CPU memory and perform the sampling on GPUs, relieving CPUs from the heavy random memory access. Moreover, recent systems [3, 23, 28, 39, 47, 50, 52, 55] also employ GPU-based feature gathering to optimize overall performance. They store the vertex feature completely or partially (with a caching mechanism) on GPUs, converting the heavy host-GPU data communication and CPU memory access to efficient GPU global memory access. The

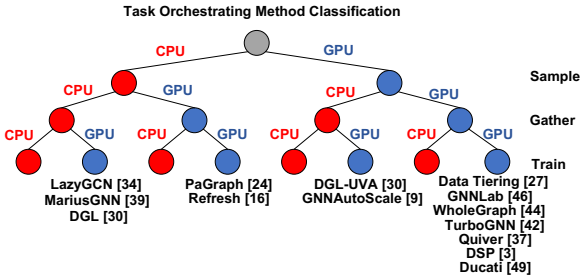

Figure 1: Classification tree of existing GNN systems and their task orchestrating methods.

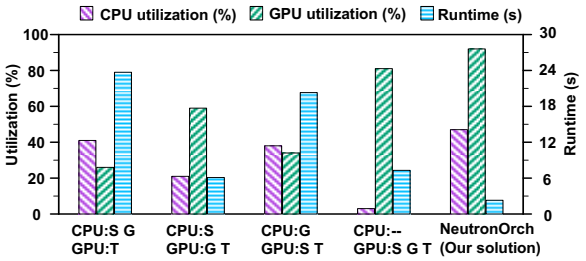

Figure 2: Comparison of resource utilization and per-epoch execution time for different task orchestrating methods. The S, G, and T represent the sample, gather, and train steps of sample-based GNN training.

GPU-based graph sampling and GPU-based feature caching can be used jointly to optimize performance.

Figure 1 provides an overview of existing GNN systems and their task orchestrating methods using a 3-layer binary tree. Each path from the root to the leaf represents a specific task orchestrating method, where the CPU is selected when traversing the left child, and the GPU is selected when traversing the right child. Since performing training on the CPU is not an efficient option when GPU is equipped, the existing task orchestrating methods on CPU-GPU heterogeneous framework contain only four categories.

Having conducted intensive experiments and analysis, we find that existing task orchestrating methods that decouple the computation based on the step do not fully utilize heterogeneous resources due to inefficient CPU processing or GPU resource contention. Assigning one or two steps to the CPU may cause inefficient CPU processing to become a bottleneck and lead to a long GPU waiting time to receive the input training data. In contrast, assigning two or more steps to the GPU may result in GPU resource contention although GPU parallel processing can significantly enhance the performance of each single step. This is because the training data, cached features, and graph topology need to be simultaneously stored within the constrained GPU memory while the computation kernels of sampling and training are completed for GPU cores.

To illustrate this, we implement the four task orchestrating methods in DGL [43] and compare their performance on a 3-layer GCN model with the Reddit dataset [21]. We evaluate the GPU utilization, CPU utilization, and per-epoch runtime as shown in Figure 2. The CPU and GPU utilization (the formal definitions are given in Section 2.3) implies the efficiency of task orchestration which affects the training performance on heterogeneous platforms. Generally, high GPU utilization is a prerequisite for achieving high performance, i.e., shorter runtime, but strategically offloading computation to the CPU can further enhance overall performance. Take the runtime results in Figure 2 as an example, despite the lower GPU utilization of the second method (CPU:S GPU:G T) compared to the fourth method (CPU:- GPU:S G T), the second method boasts higher CPU utilization, resulting in higher overall resource utilization. Consequently, the second method achieves a shorter runtime compared to the fourth one. Based on these observations, we design a new task orchestrating method that comprehensively utilizes heterogeneous resources and achieves optimal runtime performance.

In this work, we rethink the roles of CPU and GPU in sample-based GNN training and propose a hotness-aware layer-based task orchestrating method to optimize performance. Unlike existing methods that allocate each step to a single device, we decouple the training task by layers and employ the computation of each sub-task (sample-gather-train) to a specific device. This is based on our observation that the multiple hops of neighbor sampling cause most of the training resources (especially the memory) consumption to be in computations from the bottom layer (i.e., the outermost layer of the sampled subgraph). Offloading computation of the bottom layer to the CPU can significantly reduce the computation and memory requirement of the GPU. Furthermore, the volume of CPU-GPU communication can be significantly reduced by transferring computed embeddings instead of raw features. Considering offloading the computation of a complete GNN layer to the CPU may make the CPU processing a new bottleneck if not carefully optimized, we propose a hotness-aware embedding reusing method that reduces CPU computation by periodically computing the historical embedding (HE) of high-hotness vertices (i.e., frequently accessed vertices) and reusing them across batches. To manage the staleness caused by reusing HE from previous batches, we propose a super-batch pipelined training, which combines bounded staleness processing [6, 12, 30, 33] with CPU-GPU pipelining techniques. Specifically, we combine k adjacent batches into a super-batch and control the staleness of HE among super-batches, overlapping GPU and CPU computation tasks while strictly and efficiently implementing bounded staleness. Previous HE-based methods [6, 9, 14, 33] uniformly reuse HE for all vertices, neglecting the influence of vertex hotness on resource utilization or failing to ensure bounded staleness among batches. Such approaches are not applicable to our method, which emphasizes balancing GPU-CPU heterogeneous resource utilization while ensuring bounded accuracy. By integrating the above techniques, we propose NeutronOrch, a sample-based GNN system that effectively utilizes heterogeneous resources.

We make the following contributions.

- We provide a comprehensive analysis of resource utilization issues associated with the task orchestrating methods for sample-based GNN systems on GPU-CPU heterogeneous platforms.
- We propose a hotness-aware layer-based task orchestrating method that effectively leverages the computation and memory resources of the GPU-CPU heterogeneous system.
- We propose a super-batch pipelined task scheduling method that seamlessly overlaps different tasks on heterogeneous resources and efficiently achieves strict bounded staleness.

Table 1: Notations

Notation	Description
$\mathbf{a}_v^l$	aggregation results vector of vertex v at the l^{th} layer
$\mathbf{h}_v^l$	embedding vector of vertex v at the l^{th} layer
$\mathbf{A_i}$	Adjacency matrix of the i^{th} iteration
$\hat{\mathbf{A}}_\mathbf{i}$	Adjacency matrix after symmetric normalization of the i^{th} iteration
$\mathbf{W}_i^l$	Weight matrix at the l^{th} layer of the i^{th} iteration
$\mathbf{H}_i^l$	Embedding matrix at the l^{th} layer of the i^{th} iteration
$\mathbf{Z}_i^l$	Input matrix to activation function at the l^{th} layer of the i^{th} iteration
$\nabla\mathcal{L}$	Gradient of the loss function
$\nabla_{\mathbf{Z}^l}\mathcal{L} = \delta^l$	Gradient matrix of the loss $\mathcal{L}$ with respect to $\mathbf{Z}^l$ at the l^{th} layer
$\nabla_{\mathbf{W}^l}\mathcal{L}$	Gradient matrix of the loss $\mathcal{L}$ with respect to $\mathbf{W}^l$ at the l^{th} layer
$g(W_i)$	Gradient of the loss function with respect to W at the i^{th} iteration
$\mathcal{L}$	Loss of the GNN
I	Number of iterations in an epoch or number of batches
L	Number of layers of GNN model
η	Learning rate
ϵ	Staleness bound

We evaluate the performance of NeutronOrch using three popular GNN models, GCN [21], GraphSAGE [11], and GAT [41]. Experimental results demonstrate that, compared with five state-of-the-art sample-based GNN systems, DGL [43], Pagraph [23], GNNlab [52], DSP [3], and GNNAutoScale [9], NeutronOrch achieves speedups of up to 11.51×, 9.72×, 2.43×, 1.63×, and 9.18×, respectively.

2 BACKGROUND

2.1 Graph Neural Networks

Graph-structured data serves as the input for Graph Neural Networks (GNNs), where each vertex or edge is associated with a high-dimensional feature vector. A typical GNN model comprises multiple layers that compute a low-dimensional embedding for each vertex. Each layer contains an aggregation phase and an update phase [58]. For example, in a GNN with L layers, during the aggregation phase of layer l, each vertex v combines its neighbors' embedding vectors at the $l-1$ layer with its own embedding vector to generate the aggregation result a_v^l using an aggregation function:

$$a_v^l = AGGREGATE(h_u^{l-1} | \forall u \in N_{in}(v) \cup \{v\}) \quad (1)$$

where $N_{in}(v)$ represents the incoming neighbors of vertex v, h_v^l represents the node embedding vector of vertex v at l-th layer, and h_v^0 is the input feature of vertex v. The aggregate functions can be `sum`, `average`, `max/min`, etc. Next, during the update phase, each vertex computes its output embedding vector h_v^l by applying an update function to the aggregation result a_v^l:

$$h_v^l = UPDATE(a_v^l, W^l) \quad (2)$$

After L layers, each vertex's feature vector becomes a low-dimensional embedding of its neighbors up to L hops away.

2.2 Sample-based Mini-batch GNN Training

Sample-based mini-batch GNN training splits the training vertices into multiple mini-batches. It first splits training vertices into mini-batches according to the batch size, then samples the multi-hop subgraph of each batch. The sampled subgraph is generated by a reverse traversal from the training vertices. There is a *fanout* parameter to specify the number of sampled neighbors per layer or vertex for controlling the size of the sampled subgraph. According to the sampled subgraph, we launch the gathering step to collect the features of sampled vertices. The batch of training vertices, the gathered features, and the sampled subgraph are then fed into the GNN model for training. Specifically, there is a forward computation process of vertex embeddings, loss computation with predicted labels and ground-truth labels, and a backward computation process of gradients. The layer-wise parameters are updated after each batch processing, so the parameters are updated multiple times instead of once in full-graph training.

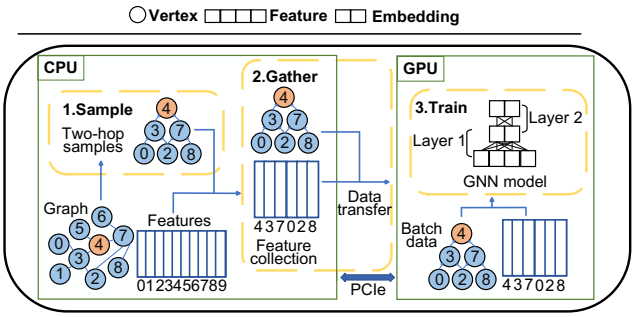

Figure 3: An example of sample-based GNN training for a two-layer GNN, where vertex 4 is the vertex with a ground-truth label for training.

Figure 3 shows an illustrative example of sample-based mini-batch GNN training model mapping to CPU-GPU heterogeneous execution environment. This is a 2-layer GNN model on a graph of 9 vertices with a training set containing only a single vertex 4. The input data, including graph data and high-dimensional features, are typically stored in host memory, while GPUs perform the GNN training on mini-batched and sampled subgraphs. In each training batch, this approach follows a **sample-gather-train** processing flow. Firstly, the graph sampling algorithm uniformly selects two neighboring vertices for each vertex. Secondly, the required vertex features are gathered based on the sampled subgraph and the training vertices. Then, they are loaded from the CPU to the GPU through a PCIe interconnect. Finally, the GPU performs forward and backward computations, computing and updating the model parameters correspondingly.

2.3 Resource utilization

We adopt the existing definitions of CPU and GPU utilization (U_{CPU} and U_{GPU}) following the existing works [18, 25, 26, 31]. The CPU utilization U_{CPU} is defined by the ratio of busy cycles to total cycles across all physical cores [18], i.e.,$U_{CPU} = \frac{\sum_{i=1}^{|cores|} T_{busy}^{(i)}}{T_{total} \times |cores|} \times 100\%$, where $T_{busy}^{(i)}$ denotes the number of busy cycles of core i, and T_{total} represents the total cycles of each core. The GPU utilization U_{GPU} is defined as the percentage of the elapsed time during a past sample period in which one or more kernels were executing [31], i.e., $U_{GPU} = \frac{T_{busy}}{T_{total}} \times 100\%$. Here, T_{busy} represents the time elapsed on actual processing, and T_{total} denotes the total elapsed time. In the implementation, U_{CPU} and U_{GPU} are measured by reading the `/proc/stat` file and executing the `nvidia-smi` command once every second, respectively.

Table 2: The runtime breakdown (in seconds) of sample and gather steps on DGL with different datasets. The FC and FT represent the feature collection and feature transfer of the gather step. GNN: A 3-layer GCN [21].

Dataset	Sample	Gather (FC)	Gather (FT)	Total
Reddit	2.7/11%	9.1/38%	6.0/25%	23.7
Lj-large	128.8/14%	384.4/41%	252.5/27%	935.3
Orkut	78.8/10%	384.3/48%	249.1/31%	813.3
Wikipedia	209.4/12%	651.8/40%	570.9/33%	1669.1
Products	9.9/37%	7.2/27%	4.1/15%	26.8
Papers100M	11.5/32%	8.6/24%	6.4/18%	36.84

3 EXISTING SYSTEMS AND THEIR LIMITATIONS

A key challenge in achieving high-performance sample-based GNN training is orchestrating tasks on the GPU-CPU heterogeneous environments. Existing sample-based GNN systems [23, 28, 29, 36, 42, 43, 47, 50, 52] typically separate the training process based on the operations, i.e., **sample**, **gather**, and **train**, and assign each operation to the GPU or the CPU. However, they often exhibit suboptimal performance due to inefficient CPU processing or GPU resource contention. To illustrate this, we conduct a series of experiments to analyze the problems of existing task orchestration methods.

3.1 Existing Task Orchestrating Methods

In Figure 1, we classify task orchestrating methods and list representative systems. Among eight possible methods, CPU-based training is not utilized if a GPU is available on the platform. Therefore, we only consider placing sampling and gathering on different devices in the analysis. Further information regarding the used graphs, test platforms, and system configurations can be found in Section 5.

Case 1: Placing sampling and gathering on CPUs suffers from inefficient CPU processing We show an example of this case in Figure 4 (a). In this case, inefficient sampling and feature gathering on the CPU block the training pipeline, making data preparation the main bottleneck for overall performance.

Neighbor sampling traverses the graph to obtain a multi-hop sampled subgraph for each training vertex. It incurs massive computation and irregular memory access, which makes the limited computational resources of the CPU hard to accelerate [19]. On the other hand, deploying the gathering step on the CPU, including storing vertex features in the host memory and performing feature collection and transfer, also proves to be inefficient [23]. This inefficiency arises from two main factors. Firstly, the slow external PCIe interconnect makes the host-GPU feature transfer much slower than GPU memory access [29]. Secondly, when loading the features of neighbors, collecting and organizing the fragmented vertex feature into contiguous memory before communication (to leverage the CUDA memory copy engine, `cudaMemcpy`) also consumes substantial CPU resources due to massive random memory accesses. We run this task orchestrating method with a 3-layer GCN on all six real-world graphs and analyze the time distribution for sampling and gathering steps. As shown in Table 2, sampling and gathering steps occupy 19.3% and 61.2% of the total runtime, respectively. Specifically, the most significant overhead is the feature collection in the gathering step, which accounts for 36.3% of the total runtime.

Table 3: The runtime breakdown (in seconds) of a training epoch on DGL with CPU-based or GPU-based sampling. S, G, and T represent sample, gather, and train steps. GNN: A 3-Layer GCN [21]. Batch size: 10000.

Configuration	S	G	T	Total	+pipeline
CPU-based sampling	2.28	2.84	2.76	7.88	3.42 (-56.6%)
GPU-based sampling	0.78	2.69	2.75	6.22	3.54 (-43.1%)

On the other hand, the long sampling and gathering time also causes low GPU utilization. The training step in the GPU needs to wait for the slow CPU tasks. We can observe from Figure 2 that merely around 25% of the GPU computational resources are utilized when the CPU executes both sampling and gathering steps. Even if only separate sampling or gathering is performed on the CPU, the overall GPU utilization is still less than 65%.

Case 2: Placing sampling on the GPU suffers from GPU resource contention We show an example of this case in Figure 4 (b). In this case, the slow sampling can be accelerated through the massively parallel processing capability of GPUs. However, the sampling and training operations will compete for the GPU resources, leading to suboptimal overall performance.

Sample-based GNN training typically employs a pipeline design that overlaps the different steps to achieve high performance in heterogeneous systems [23, 42, 43, 52]. Figure 5 (a) depicts an ideal pipeline implementation where the operations of three batches (numbered 1, 2, and 3) are fully overlapped. However, this optimization requires different steps to be executed on independent resources. When placing sampling on the GPU, it competes for computation resources with the training step, leading to inefficient pipelining, as shown in Figure 5 (b). To illustrate this, we evaluate the performance of DGL with pipelined optimization and non-pipelined under both CPU-based and GPU-based sampling. We run a 3-layer GCN on the Reddit graph and report the results in Table 3. The GPU-based sampling with pipeline optimization reduces the runtime of its non-pipelined version by 43.1%, which is a lower improvement than the effect of pipeline optimization on CPU-based sampling. Furthermore, with pipeline optimization, GPU-based sampling shows inferior overall performance compared to CPU-based sampling.

Case 3: Placing gathering on the GPU suffers from GPU memory contention GPU-based gathering method leverages GPU memory to cache the vertex features, converting the slow host-GPU data communication into fast GPU memory access as much as possible [3, 23, 28, 39, 47, 50, 52, 55]. However, the performance of GPU data caching is affected by the available GPU memory. We show an example of this case in Figure 4 (c). In actual training, a substantial amount of global memory must be allocated for storing the training data and intermediate results, leaving only a small portion of memory available for feature caching. If using a large cache buffer, the memory allocated for training has to be reduced, resulting in the training with a small batch size. Unfortunately, this trade-off may lead to inferior performance since the computational power of GPUs cannot be fully utilized [10, 16]. To illustrate the impact of batch sizes and the ratio of cached vertices (denoted by cache ratio) on performance, we conduct experiments on DGL using the 3-layer GCN model on the Wikipedia. The results in Figure

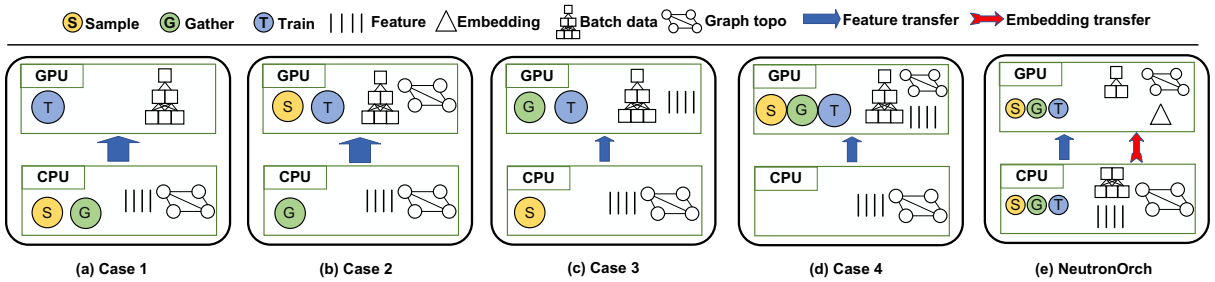

Figure 4: Illustration of the four task orchestrating methods and NeutronOrch. (a) Case 1 [36, 42]: Placing sampling and gathering on CPUs. (b) Case 2 [43]: Placing sampling on the GPUs. (c) Case 3 [14, 23]: Placing gathering on the GPUs. (d) Case 4 [3, 28, 40, 47, 50, 52, 55]: Placing sampling and gathering on the GPUs. (e) Hotness-aware layer-based task orchestrating method of NeutronOrch. The width of the arrow is positively proportional to the transfer data volume. The volumes of the circles S, G, and T are proportional to the task amount.

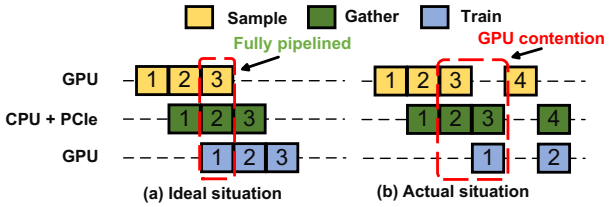

Figure 5: Two examples of multi-stream pipeline design in the ideal and actual situations.

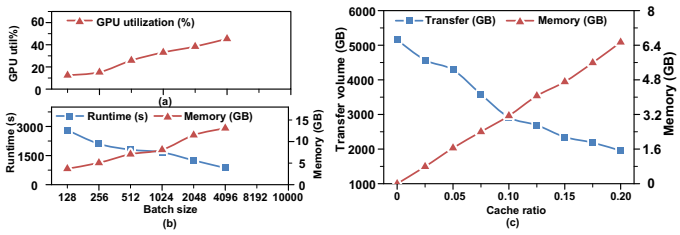

Figure 6: (a) GPU utilization with different batch sizes. (b) Per-epoch runtime and memory usage with different batch sizes. (c) Transfer volume and memory usage with different ratios of cached vertices (cache ratio).

6 (a) and (b) demonstrate that training with a larger batch size is beneficial for GNNs in terms of achieving better GPU utilization. Although this consumes more GPU memory, it results in faster execution. The results in Figure 6 (c) reveal that a larger cache ratio results in a linear transfer reduction of features in the gathering step. However, when increasing the batch size from 128 to 4096, the ratio of cached vertices is decreased from 0.37 to 0.05 due to insufficient GPU memory, which results in numerous cache misses. Moreover, as the graph topology data and feature dimensions increase, the benefits brought by caching vertices will further diminish.

Case 4: Placing sampling and gathering on the GPU suffers from GPU memory and resource contention We show an example of this case in Figure 4 (d). When all steps of sample-based GNN training are executed on the GPU, it results in GPU contention and CPU idle. The reasons for this situation have been discussed in cases 2 and 3. Firstly, the computation kernels of sampling and training compete for the limited GPU cores, leading to a slowdown in both. Secondly, the batch data for training and cache data for gathering contend with the limited GPU memory. When further making the GPU responsible for the sampling step, the GPU needs to additionally hold the graph topology data, which can make the GPU memory contention worse.

3.2 Summary

We conduct experimental analysis on different task orchestrating methods in GPU-CPU heterogeneous environments. Our observations reveal that step-based task orchestrating leads to an imbalanced allocation of computational and memory resources. Assigning two or more steps to the GPU may result in memory or GPU resource contention. On the other hand, assigning one or two steps to the CPU may cause inefficient CPU processing to become a bottleneck. A well-designed CPU-GPU heterogeneous system should ensure adequate and balanced CPU and GPU utilization to achieve optimal performance. However, the step-based task orchestrating methods fail to achieve this. This motivates us to design a resource-balanced task orchestrating method.

4 NEUTRONORCH

We propose NeutronOrch, a sample-based GNN training system that effectively improves CPU and GPU resource utilization through two critical techniques.

Hotness-aware layer-based task Orchestrating. Unlike step-based task orchestrating methods, NeutronOrch decouples the training process by layers rather than steps and employs the sample-gather-train computation of each sub-task to a single device, eliminating the constraint of computing each step entirely on the CPU or GPU. To prevent CPU computation from becoming a bottleneck, NeutronOrch allows the CPU to compute embeddings only for the hot vertices that are frequently accessed. Moreover, NeutronOrch extends stale synchronous processing [12] to guarantee bounded staleness of embedding reuse, thereby guaranteeing final accuracy.

Super-batch pipelined training. Concurrently executing sub-tasks deployed on the GPU and CPU is essential to achieve high performance. However, it is challenging for layer-based task orchestrating because of the cross-layer data dependencies between the CPU and GPU training process. To solve this, we propose a super-batch pipelined training. In each super-batch, the CPU and GPU execute sub-tasks concurrently and independently of each other so that the training process is fully pipelined. In addition, this pipeline

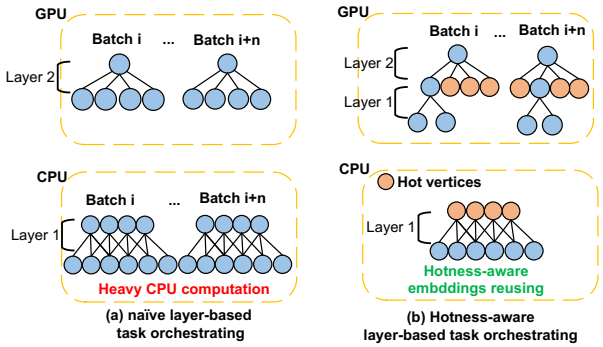

Figure 7: An illustrative of CPU and GPU workload in naive layer-based task orchestrating and hotness-aware layer-based task orchestrating.

design ensures that the historical embeddings from the previous super-batch are only accessible within the current super-batch, naturally and efficiently enabling strict bounded staleness.

4.1 Hotness-Aware Layer-based Task Orchestrating

In this section, we give a detailed discussion on the hotness-aware layer-based task orchestrating. It maximizes CPU-GPU resource utilization based on two principles. Firstly, partitioning the tasks in a fine-grained manner to balance the workloads on the CPUs and GPUs. Secondly, combining CPU and GPU resources to minimize CPU-GPU communication overhead.

4.1.1 Layer-based Task Orchestrating. Unlike existing methods that allocate each step to a single device, we decouple the training task by layers and employ the computation of each sub-task (sample-gather-train) to a specific device. This is based on our observation that in the k-hop sampled subgraph of GNNs, the number of vertices grows exponentially across layers. The bottom layer, which represents the outermost layer of the sampled subgraph, constitutes over 50% of the training workload, so the computation and transfer overhead for the bottom layer is the most expensive no matter using a large or a small number of layers. Therefore, offloading the computation of the bottom layer to the CPU can significantly reduce the computation and memory requirement of the GPU.

As shown in Figure 4 (e), the CPU is responsible for the bottom layer computation of GNN training, while the GPU is responsible for the computation of the other layers. The CPU exhibits an inherent advantage when executing the bottom layer. Firstly, as the storage of features, the CPU can directly perform the GNN computation of the bottom layer without executing the time-consuming feature collection stage. Secondly, the CPU-GPU communication overhead will decrease as the transfer objects are changed from the neighbors' features to the computed embeddings. On the other hand, this method makes more GPU memory available for training data, as it maintains the batch data in both CPU and GPUs, resulting in reduced GPU memory usage.

By employing layer-based task orchestrating, NeutronOrch significantly reduces the computational workload and memory footprint of GPU training.

4.1.2 Hotness-aware Embedding Reusing. As shown in Figure 7 (a), executing a complete bottom layer in the CPU may cause the CPU processing a new bottleneck because the computation volume of the entire layer is high. Previous studies on caching policies have demonstrated that a subset of vertices is frequently accessed during sample-based GNN training [23, 28, 52], which is commonly referred to as the "hot vertices". By caching the features of these vertices in the GPU, existing works can avoid repeatedly loading them from the host memory, thereby reducing CPU-GPU communications. NeutronOrch follows similar ideas to design a hotness-aware computation offloading method, which computes only the embeddings for hot vertices in the CPU and allows the GPU to reuse these embeddings between batches within bounded staleness [6, 12, 33], our task orchestrating method can substantially decrease the CPU workload and CPU-GPU communication.

The main idea of reusing embeddings within bounded staleness is to maintain the historical embedding $\tilde{h}_i^{(l)}$ for exact embedding $h_i^{(l)}$ as an affordable approximation with a given bound [6, 12, 33]. Bounded staleness expects $\tilde{h}_i^{(l)}$ and $h_i^{(l)}$ to be similar if the model weights do not change too fast during the training. In NeutronOrch, we use the number of model parameter updates, i.e., the number of batches, as the bound. Specifically, if the bound is N, a recently computed embedding can be reused within the subsequent N batches. Using historical embeddings avoids the need for aggregating neighbor features and the associated backward pass. This not only reduces the amount of raw features to load but also minimizes the CPU workload.

Figure 7 (b) illustrates an example of the hotness-aware layer-based task orchestrating method. During training, for hot vertices, the CPU samples them in one hop and computes their embeddings, which will be fetched and reused by the GPU training process to reduce computation and communication. For the cold vertices in the bottom layer, the GPU pulls the features of their vertices and computes them locally. To guarantee model accuracy, it is essential to control the version of reused embeddings within a specified range of batches during training, which is known as a concept of bounded staleness [12]. To achieve this, we propose a super-batch pipelined training method that achieves bounded staleness among batches by packing each consecutive N batches into a super-batch and controlling the embedding reusing among super-batches during computation. In this way, each embedding of the hot vertex is updated at least once within N batches. A detailed discussion is provided in Section 4.2. Additionally, we offer theoretical analysis in Section 4.3 to demonstrate the correctness of our design. By adopting this approach, NeutronOrch effectively offloads computations to the CPU and prevents CPU computation from becoming a performance bottleneck.

4.1.3 Hybrid Hot Vertices Processing for high-performance GPU servers. Orchestrating tasks across CPUs and GPUs balances CPU and GPU resource utilization and reduces CPU-GPU data communication by offloading the computation of hot vertices to the CPU. However, in cases where the computing server is equipped with multiple GPUs but has limited CPU computing power, such as in a single-CPU-multi-GPU environment, the contribution of using CPU computation decreases because the improvement may

not surpass the benefit brought about by the increased GPU resources. To address this problem, we propose a hybrid processing policy for hot vertices that further balances CPU and GPU utilization by assigning hot vertices to both CPU computation and GPU feature caching. When GPU resources significantly outnumber CPU resources, NeutronOrch adaptively assigns hot vertices to the GPU, utilizing feature caching to avoid computing them on the CPUs. Specifically, during execution, NeutronOrch monitors the time elapsed on GPU idleness caused by CPU computation and the remaining available GPU memory. It adjusts the allocation of hot vertices to GPU and CPU through a worklist, ensuring that GPU memory does not overly subscribe while minimizing GPU idle time. Note that NeutronOrch stops assigning hot vertices to GPU when GPU memory is exhausted or the GPU idle time reaches zero, as appropriately utilizing CPU computation can effectively reduce data transfers. Compared to computing embedding of all hot vertices on the CPU, our hybrid processing optimization can further improve heterogeneous resource utilization on servers where GPU resources are significantly powerful than CPU resources.

4.1.4 Discussion of NeutronOrch and other HE-based GNN training frameworks. Recent studies [6, 9, 14, 33] have shown the benefits of using historical embedding (HE) for GNN training acceleration. VR-GCN [6] uses HE in GNN training to reduce the number of vertices to sample. GNNAutoScale [9] reuses HE for all vertices to reduce training memory consumption on the GPU. Refresh [14] caches HE in GPU memory to reduce the CPU-GPU feature communication and GPU training time. SANCUS [33] reuses HE for boundary vertices to reduce communication in distributed full-graph training. However, these frameworks focus on scenarios where training is exclusively deployed on GPUs [6, 9, 14, 33], and either target different training paradigms [33], uniformly reuse HE for all vertices [6, 9, 14] which does not consider the impact of vertex hotness to the resource utilization, or do not strictly guarantee bounded staleness [6, 9] among batches, which is not applicable to NeutronOrch that focuses on balancing GPU-CPU heterogeneous resource utilization and guaranteeing bounded accuracy. To achieve this goal, NeutronOrch adopts several critical designs. 1) NeutronOrch selectively computes HEs for the frequently accessed vertices and maximizes CPU-GPU resource utilization by adjusting the proportion of hot vertices assigned to CPU computation and GPU feature cache (Section 4.1.3). 2) We conduct a theoretical analysis on the convergence of NeutronOrch's HE to demonstrate its correctness (Section 4.3). Furthermore, NeutronOrch provides a super-batch pipelining method, combining HE version control with pipelined task scheduling to achieve efficient and strict bounded staleness (Section 4.2).

4.2 Super-batch Pipelined Training

In this section, we discuss the data dependencies arising from the integration of CPU and GPU computations and propose a super-batch pipelined training method to manage them efficiently.

4.2.1 Data Dependencies. Seamlessly overlapping tasks across diverse computing resources through pipelining is essential to achieve high performance on heterogeneous systems [3, 29, 47]. However, implementing high-performance task pipelining in NeutronOrch is a non-trivial task because the layer-based task orchestrating creates cross-layer data dependencies between the CPU and GPU training process. Figure 8 (a) illustrates data dependencies in NeutronOrch when setting the stale bound to one epoch [6, 9]. The CPU updates the embeddings for hot vertices when it reaches the stale bound, and the training on the GPU must wait for the CPU to complete the embedding computation for hot vertices. Consequently, the CPU and GPU need to fall back to sequential computation, resulting in inefficient resource utilization.

4.2.2 Pipeline Design. To address the above issues, we propose a super-batch pipelined training method. Firstly, the embeddings update for hot vertices in the CPU is partitioned into multiple sub-computing tasks, each of which computes the embeddings for hot vertices for subsequent multiple batches, which provides more opportunities for pipelined designs. Secondly, we limit the stale bound to several batches instead of one epoch because more tightly controlled bounded staleness can help ensure model accuracy.

Before training starts, NeutronOrch combines n batches into a single super-batch. Then, we select a hot vertices queue for each super-batch. As shown in Figure 8 (b), this is an illustrative example of a super-batch pipelined training when $n = 4$. During the pipelined training, the CPU computes embeddings for hot vertices for the next super-batch. In each super-batch, the GPU computes n batches of training for a given batch size, sharing the historical embeddings provided by the CPU from the last super-batch. Compared to updating the embedding for all hot vertices, updating the embedding for only the next super-batch has less computation overhead and reduces the version gap between historical embeddings and exact embeddings. Our pipeline design consists of four stages, each serving a specific purpose. We now elaborate on these individual stages.

Stage 1: Sampling. The sampling tasks are divided between the CPU and GPU. Firstly, the CPU samples the hot vertices and generates a one-hop subgraph for bottom-layer training. Next, the GPU initiates n-hop sampling, and upon reaching the bottom layer, it skips the vertices previously sampled by the CPU. Meanwhile, the GPU completes n rounds of sampling before n training rounds to mitigate potential resource contention between the sampling and training kernels.

Stage 2: Embedding computation for hot vertices. The CPU is responsible for updating the embeddings of the hot vertices intended for the subsequent super-batch within each super-batch. The CPU conducts training based on the hotness order of the vertices and utilizes the most recent version of parameters.

Stage 3: Feature gathering. During the gathering step, the embeddings of hot vertices and the features of cold vertices are transferred. After being transferred, the embeddings of the hot vertices will be cached in the GPU for use in other batches within a super-batch.

Stage 4: Training on GPU. The GPU training starts after n rounds of GPU sampling. During the forward pass, the GPU pulls the historical embeddings of hot vertices in the bottom layer from either the CPU or the GPU cache. For other layers, the GPU performs aggregation and updates alternately as normal. In the backward pass, the GPU updates the parameters of each layer and shares the parameters of the bottom layer with the CPU.

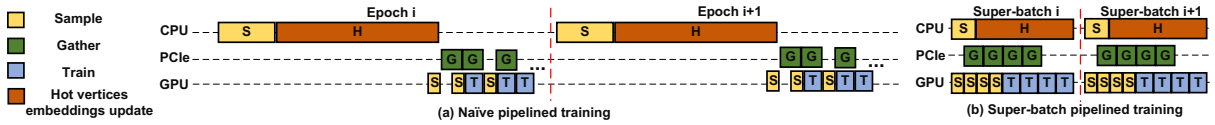

Figure 8: An illustrative example of super-batch pipelined training.

Bounded Staleness. Model parameters are updated once within each batch, so we record the model parameter version by batch number. The CPU trains hot vertices in order of hotness and utilizes the latest model parameters. As shown in Figure 8 (b), during the first super-batch, the embeddings for hot vertices in the CPU may have versions ranging from 0 to $n-1$, where n is the number of batches contained within a super-batch. In the second super-batch, when the GPU pulls embeddings for hot vertices from the CPU, it may receive historical embeddings with model parameters that are older than the current version. The CPU must complete the embeddings update for hot vertices for the next super-batch within the current super-batch. This constraint ensures that the version gap remains within two super-batches, effectively preventing excessive staleness. Figure 8 (b) illustrates that the most significant version gap may occur in the last batch of the second super-batch. During this batch, the model parameter version is $(i+2)\cdot n-1$, and it may utilize historical embeddings from the previous super-batch with a model parameter version of $i\cdot n$. For the other batches, the version gap between historical embeddings and exact embeddings is smaller than the upper bound of $2n-1$.

4.3 Convergence Analysis

In this section, we provide a theoretical analysis of convergence guarantees for NeutronOrch. Bounded staleness has been widely used by machine learning systems [6, 12, 30, 33, 52], and we present the theoretical analysis referring to the SANCUS [33] and VR-GCN [6]. To ensure bounded staleness, NeutronOrch designs super-batch pipelined training to limit the version (batch number) bound to $2n$, where n is the number of batches in a super-batch. In this process, we monitor the model weight variation between adjacent super-batches, and the maximum model weight variation ϵ that can be tolerated be defined as $\epsilon = max\triangle\left\|W\right\|\times 2n$, where $max\triangle\left\|W\right\|$ denotes the maximum value variation of W in a model weight update. With this staleness bound, we deduce the convergence guarantee as follows.

- **Proposition 1** provides the necessary and fundamental inequality operations required for the theoretical analysis;
- **Lemma 1** states that by imposing bounded staleness on the weights, the approximations of the embeddings and intermediate matrix results are close to the exact results;
- **Lemma 2** further demonstrates that the approximations of gradients in the training process closely match the exact gradients;
- **Theorem 1** concludes that the weight changes during training occur at a sufficiently slow rate, ensuring that the gradients are asymptotically unbiased and guaranteeing convergence;

Proposition 1. *Let* $\left\|A\right\|_\infty = max_{ij}\left|A_{ij}\right|$, *then we have*:

$$\left\|AB\right\|_\infty \le col(A)\left\|A\right\|_\infty\left\|B\right\|_\infty$$
$$\left\|A\odot B\right\|_\infty \le \left\|A\right\|_\infty\left\|B\right\|_\infty$$
$$\left\|A+B\right\|_\infty \le \left\|A\right\|_\infty+\left\|B\right\|_\infty$$

where, $col(A)$ *denotes the number of columns of the matrix* A, $\odot$ *denotes the element wise product.* This proposition has been proved by VR-GCN [6], and we omit the proof. We further denote C as the maximum number of columns that exist in our analysis.

Lemma 1, lemma 2, and Theorem 1 have also been proved by SANCUS [33] and VR-GCN [6], and we omit the proof and give the necessary and sufficient conditions for their establishment.

lemma 1. *Assume all the activations are* ρ*-Lipschitz, the* $\left\|W_i\right\|_\infty$ and $\left\|\hat{A}_i\right\|_\infty$ *are bounded by some constant* B, *and the historical weights* $\tilde{W}_i$ *are close to the exact weights* W_i *with the staleness bound* ϵ *where* $\left\|\tilde{W}_i - W_i\right\| \le \epsilon, \forall i$. *Then the approximation error of the stale embedding* $\tilde{H}$ *and stale activation* $\tilde{Z}$ *is bounded by some constant* K *that depends on* ρ, C, B: $\left\|H_i^l - \tilde{H}_i^l\right\|_\infty < \epsilon K, \forall i > I, l = 1, ..., L-1$; $\left\|Z_i^l - \tilde{Z}_i^l\right\|_\infty < \epsilon K, \forall i > I, l = 1, ..., L$.

lemma 2. *Assume that activation function* $\sigma(\cdot)$ *and the gradient* $\nabla\mathcal{L}$ *are* $\rho-$*Lipschitz*, the $\left\|\nabla\mathcal{L}\right\|_\infty$, $\left\|\hat{A}_i\right\|_\infty$, $\left\|W_i\right\|_\infty$, *and* $\left\|\sigma'(Z_i)\right\|_\infty$ *are bounded by some constant* B, *and the historical weights* $\tilde{W}_i$ *are close to the exact weights* W_i *with the staleness bound* ϵ *where* $\left\|\tilde{W}_i - W_i\right\| \le \epsilon, \forall i$. *Then the approximation error of the gradient* $\tilde{g}(W_i)$ *is bounded by some contents:* $\left\|\mathbb{E}\tilde{g}(W_i) - \nabla\mathcal{L}(W_i)\right\|_\infty \le \epsilon K$ *and* $\forall i > I$, *where* K *depends on* ρ, C, B.

Theorem 1. *Given the local minimizer* $W^\star$. *Assume that (1) the activation* $\sigma(\cdot)$ *is* ρ*-Lipschitz, (2) the gradient of the loss function* $\nabla\mathcal{L}(W_i)$ *is* ρ*-Lipschitz and bounded, (3) The gradient matrices* $\left\|\tilde{g}(W)\right\|_\infty$, $\left\|g(W)\right\|_\infty$ and $\left\|\nabla\mathcal{L}(W)\right\|_\infty$ *are bounded by some constant* $G > 0$. *(4) The loss* $\mathcal{L}(W)$ *is* ρ*-smooth, i.e.,*

$$\left|\mathcal{L}(W_2) - \mathcal{L}(W_1) - \langle\nabla\mathcal{L}(W_1), W_2 - W_1\rangle\right| \le \frac{\rho}{2}\left\|W_2 - W_1\right\|_F^2, \forall W_1, W_2$$

, *where* $\langle A, B\rangle = tr(A^TB)$ *is the inner product of matrix* A *and matrix* B. *Then, there exists* $K > 0$, *s.t.,* $\forall N > I$, *if we run SGD for* $R \le N$ *iterations, where* R *is chosen uniformly from* $[I+1, ..., N]$ *and the learning rate* $\eta = min\{\frac{1}{\rho}, \frac{1}{\sqrt{N}}\}$, , we have:

$$\mathbb{E}_R\left\|\nabla\mathcal{L}(W_R)\right\|_F^2 \le 2\frac{\mathcal{L}(W_1) - \mathcal{L}(W^\star) + \frac{\rho K}{2}}{\sqrt{N}}$$

when $N\to\infty$, $\mathbb{E}_R\left\|\nabla\mathcal{L}(W_R)\right\|_F^2 \to 0$, the above concludes that the convergence is guaranteed.

5 EVALUATION

This section evaluates the performance of NeutronOrch using three representative GNN models and six real-world graph datasets.

5.1 Experimental Setup

Environments. The experiments are conducted on an Aliyun server equipped with an Intel Xeon Platinum 8163 CPU (96 cores and 736 GB main memory) and eight NVIDIA V100 (16GB) GPUs. The eight GPUs are connected to the CPU via four PCIe-3.0 switches and equipped with NVLink interconnects similar to NVIDIA DGX-1 [32]. The GPU is enabled with CUDA 11.4 runtime and 418.67 drivers. The serversruns Ubuntu 18.04 with Linux kernel 4.13.0.

Table 4: Dataset description.

Dataset	|V|	|E|	ftr. dim	#𝕃	hid. dim
Reddit [11]	232.96K	114.61M	602	41	256
Lj-large [1]	10.69M	224.61M	400	60	256
Orkut [51]	3.1M	117M	600	20	160
Wikipedia [22]	13.6M	437.2M	600	16	128
Products (PR) [13]	2.4M	61.9M	100	47	64
Papers100M (PA) [13]	111M	1.6B	128	172	64

GNNs and model configurations. NeutronOrch can support common message-passing based GNNs (e.g., GCN [21], GraphSAGE [11], GAT [41], and GIN [49]) without requiring additional tuning on the parameters in the system because we do not change training semantic. We evaluate NeutronOrch using three representative GNN models: GCN [21], GraphSAGE [11], and GAT [41]. The training batch size is set to 1024, the model depth is set to 3.

Sampling algorithm. NeutronOrch provides a set of efficient sampling algorithm implementations (e.g., k-hop neighbor and layer-wise) following the DGL [44] framework to leverage the mature graph sampling optimizations. In the experiments, we use k-hop neighbor sampling for our evaluated GNNs following the configuration of DGL. The sampling fan-out is set to [25, 10, 5] in the first three layer and fixed to 5 in the following layers, i.e., [25, 10, 5, 5 $\cdots$]. In this work, we do not provide an in-depth discussion of sampling algorithm performance as the sampling optimization is orthogonal to the layer-based task orchestrating. Nevertheless, NeutronOrch maintains the flexibility to incorporate advanced sampling implementations.

Datasets. For evaluation, we utilize six real-world graph datasets, as listed in Table 4. These include Reddit [11] and Orkut [51], which are social networks, the Wikipedia (Wiki) network [22] comprising wikilinks from the English Wikipedia, the LiveJournal communication network (Lj-large) [1]. The Products (PR) [13] dataset is based on Amazon's co-purchasing network. The Papers100M (PA) [13] is a citation graph, where each vertex represents a paper and the edge represents the citation relation. The "ftr. dim" column represents the dimension of vertex features, the "#L" column represents the number of vertex classes, and the "hid. dim" column represents the embedding dimension of the hidden layer output. For graphs without ground-truth properties (Lj-large, Orkut, and Wikipedia), we use randomly generated features, labels, training (65%), test (10%), and validation (25%) set division.

Baselines. We compare NeutronOrch with DGL [43], DGL-UVA [29] , PaGraph [23], GNNLab [52], GNNAutoScale (GAS) [9], and DSP [3]. All comparison systems use GPUs to conduct training. DGL and DGL-UVA store both the graph structure and features in CPU memory. The difference is that DGL conducts sampling using the CPU, while DGL-UVA conducts sampling using the GPU by utilizing the UVA technique. PaGraph and GNNLab utilize GPU-based feature caching to reduce CPU-GPU communication. PaGraph conducts the sampling using the CPU, and GNNLab stores the graph structure in GPU memory and conducts the sampling using the GPU. GAS conducts feature gathering in the CPU and utilizes historical embedding to accelerate training. DSP is a multi-GPU GNN training system that uses multi-GPU cooperative sampling. It also caches the graph topology and popular vertex features in GPU memory to accelerate the gathering step.

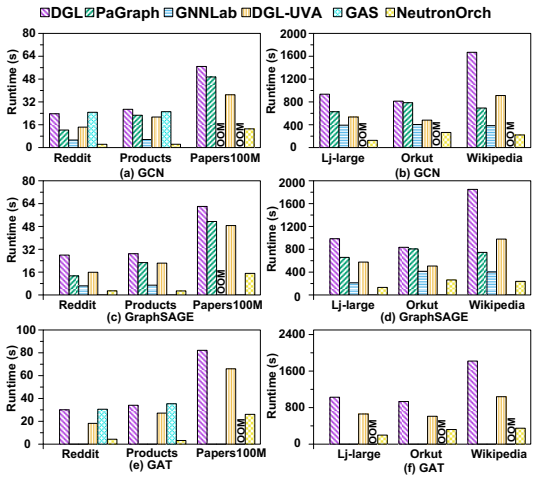

Figure 9: Overall training performance comparison ("OOM" denotes out of memory).

5.2 Single GPU Performance

Figure 9 shows the average training time of one epoch for all GNN systems. Note that GNNLab [52] and PaGraph [23] do not provide GAT support, and GAS [9] does not provide GraphSAGE support.

(1) Comparison with DGL [43]: NeutronOrch achieves a speed-up ranging from 2.91× to 11.51× over DGL. DGL exhibits inferior performance compared to the other systems due to inefficient CPU-based sampling and gathering steps. NeutronOrch effectively avoids inefficient CPU processing by selectively computing historical embeddings for hot vertices on the CPU.

(2) Comparison with PaGraph [23]: NeutronOrch achieves a speed-up ranging from 2.68× to 9.72× over PaGraph. The performance of PaGraph is limited by slow CPU sampling and GPU memory contention. NeutronOrch provides more flexible task orchestration for accelerating training while avoiding GPU contention.

(3) Comparison with GNNLab [52]: NeutronOrch achieves a speed-up ranging from 1.52× to 2.43× over GNNLab. NeutronOrch performs better when training with larger models or datasets. When the training memory requirement increases, GNNab's performance degrades due to a decreased cache hit rate. Moreover, when handling deeper GNN models, GNNLab encounters out-of-memory (OOM) issues due to GPU memory exhaustion.

(4) Comparison with DGL-UVA [43]: NeutronOrch achieves a speed-up ranging from 1.81× to 9.18× over DGL-UVA. DGL-UVA supports accessing graph topology and features via the zero-copy transfer engine [29]. Compared to saving the graph topology in the GPU for sampling, DGL-UVA saves GPU memory while introducing access latency between CPU and GPU. On the other hand, DGL-UVA has a larger communication compared to NeutronOrch because it transfers all features needed for training in every iteration.

(5) Comparison with GAS [9]: NeutronOrch achieves a speed-up ranging from 7.08× to 11.05× over GAS. GAS computes historical embeddings for all vertices and transfers them back to the CPU memory. Although GPU memory is saved, additional overhead is incurred due to frequent CPU-GPU communication. GAS also faces CPU memory limitations on graphs with many vertices, as it needs to store embeddings for all vertices across every layer. In contrast, NeutronOrch selectively computes historical embeddings for frequently accessed vertices on the CPU, achieving efficient historical embedding reuse while reducing memory overhead.

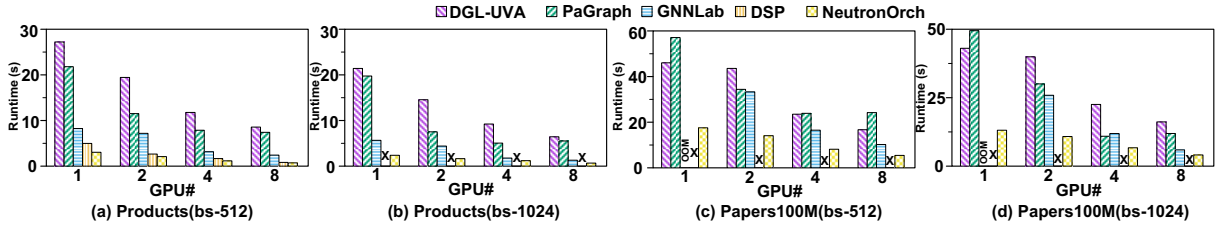

Figure 10: Per-epoch runtime of NeutronOrch and other systems under multi-GPU environment. (bs denotes batch size, "x" denotes illegal memory access, and "OOM" denotes out of memory).

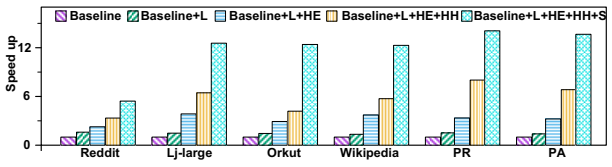

Figure 11: Performance gain analysis.

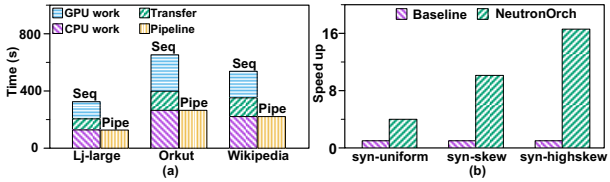

Figure 12: (a) Performance breakdown of NeutronOrch , where 'Seq' for the sequential execution of different phases and 'Pipe' for the concurrent execution of different phases. (b) Performance improvement analysis under synthetic datasets with different degree distributions.

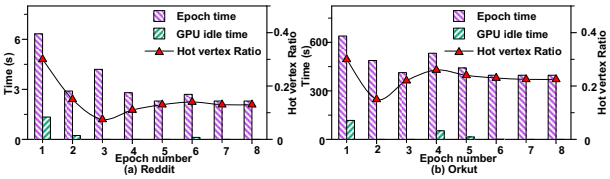

Figure 13: Dynamic adjustment of hot vertex ratio.

5.3 Multi-GPU Performance

We conduct a comparative analysis of NeutronOrch, PaGraph [23], DGL-UVA [29], GNNLab [52], and DSP [3] to evaluate the scalability by varying the number of GPUs used in training. Figure 10 shows the results of training GraphSAGE against two real-world datasets with different numbers of GPUs. NeutronOrch consistently demonstrates superior performance over the baselines with different batch sizes and different numbers of GPUs. Compared to DGL-UVA [29] and PaGraph [23], NeutronOrch achieves on average 6.33× and 5.20× speedups. The performance of DGL-UVA and PaGraph is limited by extensive CPU-GPU communication and inefficient CPU sampling. In contrast, NeutronOrch minimizes CPU-GPU communication by reusing the historical embeddings and adaptively adjusting the workload between CPU and GPUs. Compared to GNNLab [52] and DSP [3], NeutronOrch achieves on average 2.28× and 1.36× speedups. GNNLab and DSP deploy all steps (sample-gather-train) on the GPU and leave the CPU idle. When the number of GPUs decreases, GPU memory contention makes the benefit of their caching method decrease. In addition, when handling large-scale graphs (Papers100M), both DSP and GNNLab report memory errors due to memory exhaustion. NeutronOrch effectively trains large-scale GNNs by offloading computations to the CPU, reducing both CPU-GPU communication and GPU memory overhead.

5.4 Performance Analysis of NeutronOrch

Performance gain. We analyze the performance gain of layer-based task orchestrating (L), hotness-aware embedding reusing (HE), hybrid hot vertices processing (HH), and super-batch pipelined training (S) on the GCN model with six datasets. We start from a baseline with NeutronOrch 's codebase for a fair comparison and gradually integrate four optimizations. The baseline employs GPU-based graph sampling, GPU-based training, and CPU-based gathering. Figure 11 shows the normalized speedups. The layer-based task orchestrating aggregates and updates all vertex features on the CPU and transfers the vertex embedding to the GPU for computation, significantly reducing CPU-GPU communication. It can achieve an average speedup of 1.52× compared to the baseline. The hotness-aware embedding reusing optimizes the layer-based task orchestrating of NeutronOrch, significantly reducing CPU computation. It provides an average speedup of 1.96× over the baseline+L. The hybrid hot vertices processing further reduces CPU-GPU communication by effectively balancing CPU and GPU resource utilization. It provides an average speedup of 1.53× over the baseline+L+HE. Finally, the super-batch pipeline design provides an average speedup of 1.92× over the baseline+L+HE+HH. This design enables the overlap of computations between the CPU and GPU, reducing overall execution time substantially.

Performance breakdown. We further provide a performance breakdown to analyze the time consumption of different phases, including the bottom-layer training task on the CPU (CPU work), data transfer between the CPU and GPU (Transfer), and the training tasks of other layers on the GPU (GPU work). We first disable the super-batch pipelining optimization to show the results of sequential execution of the three phases and then enable the pipelining to show the results of optimized version. Since different GNN models exhibit similar patterns on these graphs, we only show the results of the GCN on the three large datasets. As shown in Figure 12 (a), the time elapsed on both CPU and GPU is roughly the same, which exceeds the time elapsed on transfer. By overlapping the three phases through super-batch pipelining, the total runtime can be significantly reduced (ranging from 2.13× to 2.32×).

5.5 Analysis of Hotness-Aware Layer-Based Task Orchestrating

Dynamic adjustment of hot vertex ratio. NeutronOrch overlaps CPU computation and GPU computation through super-batch

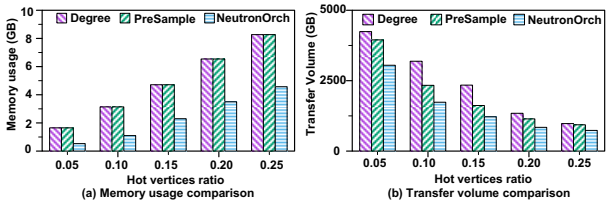

Figure 14: The memory consumption and transfer volume comparison.

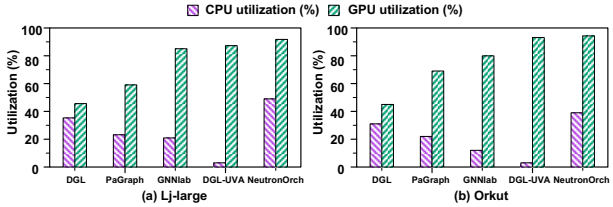

Figure 15: GPU utilization and CPU utilization comparison.

pipelining. The hot vertex ratio influences the time elapsed on CPU computation and GPU idleness because it determines the computation volume assigned to GPUs and CPUs. To optimize performance, NeutronOrch adopts an adaptive hot vertex ratio adjustment approach to balance the time elapsed on CPU and GPU computation and minimize device idleness. During execution, NeutronOrch monitors the time elapsed on GPU idleness caused by CPU computation and adjusts the hot vertex ratio to ensure GPU idle time reaches zero. We achieve this goal through a binary search method.

We evaluate the dynamic adjustment process of the hot vertex ratio across two datasets on the GCN model. The hot vertex ratio is initialized to 0.3 (30%), and the binary search is employed to adjust the hot vertex ratio of current epochs according to the time elapsed on GPU idleness caused by CPU computation. The hot vertex ratio adjustment terminates when the binary search interval reaches less than 0.01. Figure 13 shows the hotness tuning and runtime among the first 8 epochs. We can observe that the adaptive adjustment method can find the optimal hot vertices ratio at the early stages of training, given that GNN training often requires hundreds to thousands of epochs. Therefore, the overhead dynamic adjustment of hot vertex allocation can be amortized.

Data transfer reduction. We implement the degree-based cache policy (Degree) [23] and the pre-sample-based cache policy (PreSample) [52] in NeutronOrch and compare them with our method (NeutronOrch). We conduct experiments using the GCN model on the Wikipedia dataset. NeutronOrch reduces the data transfer volumes through a combination of CPU computation offloading and historical embedding reusing. Specifically, NeutronOrch offloads the computation of the bottom layer to the CPU and transfers the computed embeddings to the GPU. Since embeddings generally have smaller dimensions than features, the data transfer volume can be significantly reduced. Furthermore, the embeddings of frequently accessed vertices are reused as historical embeddings multiple times after being transferred to the GPU, further reducing the volume of data transfers. Figure 14 (a) illustrates that the NeutronOrch leads to an average reduction of 55.1% in GPU memory consumption for caching compared to static cache policies because caching historical embeddings of hot vertices is more memory-efficient. On the other hand, under different hot vertices ratios, the average transfer volume of NeutronOrch is 63.2% and 75.8% of the static caching strategies Degree and PreSample, respectively.

5.6 Analysis of Resource Utilization

We evaluate the resource utilization during the training of GCN on Lj-large and Orkut. Figure 15 presents a comparison of GPU and CPU utilization. NeutronOrch exhibits superior utilization of CPU and GPU resources by scheduling training tasks across CPUs and GPUs, averaging 44.5% and 92.9%, respectively. DGL and PaGraph have good CPU utilization and poor GPU utilization. This is because performing a complete sampling or gathering step on the CPU improves CPU utilization but causes the GPU to wait. DGL-UVA and GNNLab have poor CPU utilization and good GPU utilization. They have good performance with sufficient GPU resources, but exploiting CPU resources can further improve performance.

Table 5: Per-epoch runtime of different systems with different model depths (GCN).

Systems	Products			Wiki		
	3-layer	4-layer	5-layer	3-layer	4-layer	5-layer
DGL	28.1	55.4	114.8	1669.1	OOM	OOM
PaGraph	20.1	45.0	78.0	693.2	OOM	OOM
DGL-UVA	21.8	41.6	71.3	911.7	1782.3	OOM
GNNLab	5.66	10.7	22.9	384.2	OOM	OOM
GAS	26.2	30.9	35.2	OOM	OOM	OOM
NeutronOrch	**2.33**	**4.58**	**10.2**	**221.1**	**483.4**	**1852.3**

Table 6: Per-epoch runtime of different systems with different batch sizes. (3-layer GCN)

Systems	Products				Wiki			
	Batch size				Batch size			
	256	1024	4096	10000	256	1024	4096	10000
DGL	35.1	28.1	11.9	5.24	2104.1	1669.1	861.9	OOM
PaGraph	31.6	20.1	15.5	11.9	1054.6	693.2	OOM	OOM
DGL-UVA	30.1	21.8	8.87	4.99	1624.5	911.7	592.4	301.6
GNNLab	12.3	5.65	2.78	1.65	774.8	384.2	OOM	OOM
GAS	56.4	26.2	18.1	15.1	OOM	OOM	OOM	OOM
NeutronOrch	**4.19**	**2.33**	**1.35**	**0.71**	**409.2**	**221.1**	**129.9**	**84.6**

5.7 Sensitivity Study

Performance with varying model depths. As the model depth increases, the effectiveness of NeutronOrch will not decrease, although NeutronOrch only offloads the lowest-level calculations to the CPU. In sample-based GNN training, the number of vertices exhibits exponential growth across layers [5, 11]. As a result, more than half of the entire training workload comes from the bottom layer. For example, on Wiki dataset with a 3-layer model, the bottom layer has 1.94M vertices, occupying 65% of the total workload, while the other layers only have 1.04M vertices in total. For the 4-layer model and 5-layer model, the bottom layer occupies 61% and 59% of the total workload, respectively. We run a GCN model on all systems and report the per-epoch runtime in Table 5, with different model depths. For the 3-layer model, NeutronOrch achieves on average 6.43× speedup over the baselines. For the 4-layer model and 5-layer model, the speedups are 5.84× and 6.31×, respectively.

Performance with varying batch sizes. To study the effectiveness of NeutronOrch under different batch sizes, we run a 3-layer GCN model on all systems and report the per-epoch runtime in Table 6, with the batch size ranging from 256 to 10000. For a batch size of 256, NeutronOrch achieves on average 5.64× speedups over different systems. For batch sizes of 1024, 4096, and 10000, the speedups are 6.43×, 9.85×, and 7.25×, respectively. The setting of

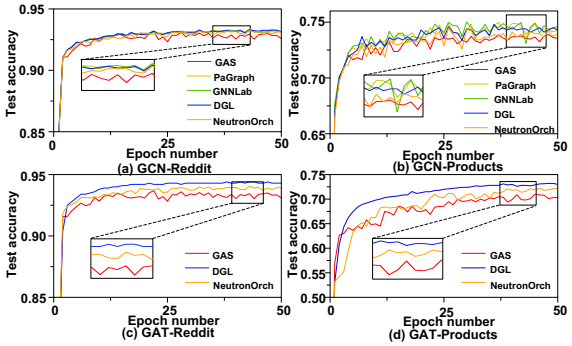

Figure 16: Epoch-to-accuracy

batch size will not affect the effectiveness. Furthermore, on large-scale graphs, NeutronOrch's CPU offloading efficiently mitigates the increasing memory overhead when using large batch sizes.

Performance with varying degrees of skew. The skewed degree distribution is expected to have the most significant impact on performance because it affects the ratio of hot vertices. We employ synthetic datasets with varying skewed degree distributions to validate this point. We employ the R-MAT [4] to generate datasets with the same graph scale but different degree distributions and randomly generate features and labels. Specifically, we configure the synthetic datasets to have 5M vertices and 200M edges, with feature dimensions of 600 and label dimensions of 40. The chosen model is a 3-layer GCN with a hidden layer dimension of 256. We create three datasets with distinct degree distributions following the guidelines of [4]. syn-uniform approximates a uniform distribution. syn-skew follows a skewed degree distribution; and synd-highskew follows a more pronounced skewed degree distribution. As shown in Figure 12 (b), we compare the performance improvement of NeutronOrch over NeutronOrch configured using GPU-based graph sampling, GPU-based training, and CPU-based gathering. On syn-uniform, NeutronOrch achieves a speedup of 4.05×. On syn-skew and syn-highskew, NeutronOrch achieves speedups of 10.13× and 16.59×, respectively. We can observe that increasing the skewness of the graph distribution leads to enhanced performance improvements.

5.8 Training Convergence

We plot the epoch-to-accuracy curve on different systems for both GCN and GAT algorithms. Benefiting from the strict version control based on super-batch pipelining, NeutronOrch accelerates GNN training while maintaing high accuracy. As shown in Figure 16, compared to other GNN systems that use historical embeddings but do not strictly control versions (e.g., GAS[9]), NeutronOrch achieves higher accuracy due to the ability to achieve smaller cumulative errors across batches. In comparison to systems that do not use historical embeddings[43], NeutronOrch achieves comparable convergence with an accuracy loss of no more than 1%.

6 FUTURE WORK

Currently, NeutronOrch operates under the assumption that all data can fit within the CPU memory. To further , we plan the following two directions:

Extension to distributed training. Distributed NeutronOrch would be interesting future work to enhance scalability. We can achieve this by integrating NeutronOrch into existing distributed GNN systems, e.g., NeutronStar [45] and DGL [43]. Specifically, NeutronOrch can be deployed on each node to accelerate single-node training by fully utilizing heterogeneous resources. The communication and graph partition module of existing systems can be used to implement efficient data parallelism.

Extension to disk-based training. Implementing SSD-based NeutronOrch would be another cost-effective solution to enhance scalability. Such an approach involves storing the graph data in persistence storage and accessing the data on demand at runtime. While supporting large graphs, such an approach introduces significant SSD I/O overhead and necessitates effective transfer management methods to reduce SSD accesses and overlap I/O and computation.

7 RELATED WORK

Task orchestrating in DNN training. Zero-Infinity DeepSpeed (ZID) [27] incorporates CPU offload techniques [34, 35, 37] to optimize the DNN training under heterogeneous environments. Although ZID and our design share similarities in scheduling training tasks across CPU and GPUs to improve scalability, their optimization objectives are different. ZID primarily focuses on reducing the memory consumption of model parameters by offloading them to the CPU. In DNNs, model parameters which consist of dense matrices that can be disjointly partitioned. These slices can be efficiently communicated between the CPU and GPUs because of their dense and regular data access patterns. In contrast, NeutronOrch primarily focuses on the efficiency of sample-based GNN training, minimizing communication and computation by offloading historical embedding computation to the CPU. Sample-based GNN training involves multiple data preparation stages (sample and gather). It exhibits irregular vertex access patterns due to the inherent complexity of graph structures, which pose new challenges in optimizing the data I/O between CPU and GPUs. ZID didn't address these challenges. However, NeutronOrch effectively resolves these problems by its task orchestrating method, transferring and reusing historical embedding for frequently accessed vertices.

8 CONCLUSION

We present NeutronOrch, a scalable and efficient GNN training system that fully utilizes CPU and GPUs. NeutronOrch leverages two key components to achieve its performance, including a hotness-aware layer-based task orchestrating method that combines CPU computation offloading with historical embeddings reuse to optimize computation and communication and a super-batch pipeline training method that utilizes CPU-GPU pipelining to achieve efficient and staleness-bounded version control. Our experiments demonstrate that NeutronOrch efficiently accelerates mini-batch GNN training with an accuracy loss of no more than 1%.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2018YFB1003400), the National Natural Science Foundation of China (U2241212, 62072082, 62202088, 62072083, and 62372097), and the Fundamental Research Funds for the Central Universities (N2216015 and N2216012). Yanfeng Zhang is the corresponding author.

REFERENCES

[1] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'06, Philadelphia, PA, USA*. 44–54.

[2] Jasmijn Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. 2017. Graph Convolutional Encoders for Syntax-aware Neural Machine Translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP'17, Copenhagen, Denmark*. Association for Computational Linguistics, 1957–1967.

[3] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP'23, Montreal, QC, Canada*. ACM, 392–404.

[4] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn (Eds.). SIAM, 442–446.

[5] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR'18, Vancouver, BC, Canada*. OpenReview.net.

[6] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning, ICML'18, Stockholmsmässan, Stockholm, Sweden (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 941–949.

[7] Ahmed El-Kishky, Michael M. Bronstein, Ying Xiao, and Aria Haghighi. 2022. Graph-based Representation Learning for Web-scale Recommender Systems. In *The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD'22, Washington, DC, USA*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 4784–4785.

[8] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference, WWW'19, San Francisco, CA, USA*. ACM, 417–426.

[9] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Jure Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 3294–3304.

[10] 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 *The 49th Annual International Symposium on Computer Architecture, ISCA'22, New York, USA*. ACM, 916–931.

[11] 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, NeurIPS'17 Long Beach, CA, USA*. 1024–1034.

[12] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. NeurIPS'12, Lake Tahoe, Nevada, United States*. 1223–1231.

[13] 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. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS'20, December 6-12*.

[14] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. 2023. ReFresh: Reducing Memory Access from Exploiting Stable Historical Embeddings for Graph Neural Network Training. *CoRR* abs/2301.07482 (2023).

[15] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, Virtual Event, Republic of Korea*. ACM, 119–132.

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, PPoPP '21, Republic of Korea*. ACM, 119–132.

[17] Ihab F. Ilyas, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, and Mohamed A. Soliman. 2022. Saga: A Platform for Continuous Construction and Serving of Knowledge at Scale. In *International Conference on Management of Data, SIGMOD'22, Philadelphia, PA, USA*. ACM, 2259–2272.

[18] Intel. 2022. Analyzing CPU Utilization. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.

[19] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Sixteenth European Conference on Computer Systems, EuroSys '21, Online Event, United Kingdom*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 311–326.

[20] 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'20, Austin, TX, USA*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org.

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR'17, Toulon, France, Conference Track Proceedings*. OpenReview.net.

[22] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW'13, Rio de Janeiro, Brazil*. International World Wide Web Conferences Steering Committee / ACM, 1343–1350.

[23] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *ACM Symposium on Cloud Computing, SoCC'20, Virtual Event, USA*. ACM, 401–415.

[24] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys'17, Belgrade, Serbia*. ACM, 527–543.

[25] Linux man pages. 2023. htop(1) — Linux manual page. https://man7.org/linux/man-pages/man1/htop.1.html.

[26] Linux man pages. 2023. top(1) — Linux manual page. https://man7.org/linux/man-pages/man1/top.1.html.

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

[28] Seungwon Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-Mei Hwu. 2022. Graph Neural Network Training and Data Tiering. In *The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD'22, Washington, DC, USA*. ACM, 3555–3565.

[29] 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.

[30] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI'21*. USENIX Association, 533–549.

[31] NVIDIA. 2018. gpu-monitoring-tools. https://github.com/NVIDIA/gpu-monitoring-tools.

[32] NVIDIA. 2022. DGX Systems. https://www.nvidia.com/en-sg/data-center/dgx-systems/dgx-1.

[33] 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.

[34] 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'20, Virtual Event / Atlanta, Georgia, USA*. IEEE/ACM, 20.

[35] 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'21, St. Louis, Missouri, USA*. ACM, 59.

[36] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Anand Sivasubramaniam, and Mahmut T. Kandemir. 2020. GCN meets GPU: Decoupling "When to Sample" from "How to Sample". In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS'20, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[37] 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, ATC'21*. USENIX Association, 551–564.

[38] Chenchen Sun, Yan Ning, Derong Shen, and Tiezheng Nie. 2023. Graph Neural Network-Based Short-Term Load Forecasting with Temporal Convolution. *Data Science and Engineering* (2023), 1–20.

[39] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July*

10-12, 2023, Julia Lawall and Dan Williams (Eds.). USENIX Association, 165–179.

[40] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter R. Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *CoRR* abs/2305.10863 (2023).

[41] 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'18, Vancouver, BC, Canada, Conference Track Proceedings*. OpenReview.net.

[42] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2022. Marius++: Large-Scale Training of Graph Neural Networks on a Single Machine. *CoRR* abs/2202.02365 (2022).

[43] 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).

[44] 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). *CoRR* abs/2311.14898 (2023).

[45] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *International Conference on Management of Data, Philadelphia, SIGMOD'22, PA, USA*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1301–1315.

[46] Wei Wu, Bin Li, Chuan Luo, and Wolfgang Nejdl. 2021. Hashing-Accelerated Graph Neural Networks for Link Prediction. In *The Web Conference 2021, WWW'21, Virtual Event / Ljubljana, Slovenia*. ACM / IW3C2, 2910–2920.

[47] Wenchao Wu, Xuanhua Shi, Ligang He, and Hai Jin. 2023. TurboGNN: Improving the End-to-End Performance for Sampling-Based GNN Training on GPUs. *IEEE Trans. Comput.* (2023).

[48] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Networks Learn. Syst.* 32, 1 (2021), 4–24.

[49] 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

[50] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. 2022. WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'22, Dallas, TX, USA*. IEEE, 1–14.

[51] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213.

[52] 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 *Seventeenth European Conference on Computer Systems, EuroSys '22, Rennes, France*. ACM, 417–434.

[53] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD'18, London, UK*. ACM, 974–983.

[54] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS'18, Montréal, Canada*. 5171–5181.

[55] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proc. ACM Manag. Data* 1, 2 (2023), 166:1–166:24.

[56] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2022. Deep Learning on Graphs: A Survey. *IEEE Trans. Knowl. Data Eng.* 34, 1 (2022), 249–270.

[57] 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. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3'20, Atlanta, GA, USA*. IEEE, 36–44.

[58] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.