NeutronAscend: Optimizing GNN Training with Ascend AI Processors

XIN AI, Northeastern University, Shenyang, China
BING ZHANG, Northeastern University, Shenyang, China
QIANGE WANG, National University of Singapore, Singapore
YANFENG ZHANG, Northeastern University, Shenyang, China
HAO YUAN, Northeastern University, Shenyang, China
SHUFENG GONG, Northeastern University, Shenyang, China
GE YU, Northeastern University, Shenyang, China

Graph Neural Networks (GNNs) have achieved remarkable success in graph data analysis. However, its application requires extensive computing resources. Recently, a new generation of Neural Processing Units (NPUs), such as the Ascend AI processor, has emerged as an essential component of computing power in data centers and technology companies due to its outstanding performance and low threshold for acquisition. Deploying GNN models on these accelerators for performance optimization is a natural choice; however, it remains a challenging task. The Ascend AI processor is typically architected on multiple AI Cores that are physically decoupled and designed for dense matrix computation. When processing graph data with inherent sparsity and power-law distribution, the Ascend AI processors suffer from the inter-core workload imbalance and inefficient intra-core resource utilization.

In this paper, we present NeutronAscend, an efficient GNN training framework tailored for the Ascend AI processor. NeutronAscend employs two critical designs for both inter-core and intra-core performance optimization. At the inter-core level, we employ GNN tensor-data hybrid parallelism that ensures load balance by partitioning vertex features evenly across AI Cores and partitioning graph data across small dimensions to maximize computing resource utilization. At the intra-core level, computation-aware task scheduling aligns the GNN process with computational units, leveraging locality-aware graph compression to reduce irregular memory access and redundant computation, while utilizing inter-unit pipelining to further enhance overall performance. Experimental results on the Ascend 910B processor demonstrate that NeutronAscend achieves an average 4.71× speedup compared to the current publicly available baselines on Ascend NPUs. While our work is tailored to Ascend, its design principles are broadly applicable to other NPUs that adopt physically decoupled, matrix-centric compute architectures.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Parallel computing methodologies**; • **Networks**;

New Paper, Not an Extension of a Conference Paper

Authors' Contact Information: Xin Ai, aixin0@stumail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Bing Zhang, zhangbing@stumail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Qiange Wang, wang.qg@nus.edu.sg, National University of Singapore, Singapore, Singapore, Singapore; Yanfeng Zhang, zhangyf@mail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Hao Yuan, yuanhao@stumail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Shufeng Gong, gongsf@mail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China; Ge Yu, yuge@mail.neu.edu.cn, Northeastern University, Shenyang, Liaoning, China.

ACM XXXX-XXXX/2018/4-ART
https://doi.org/XXXXXXX.XXXXXXX

Additional Key Words and Phrases: GNN training, Ascend AI Processor, Machine Learning System, Task Scheduling

ACM Reference Format:
Xin Ai, Bing Zhang, Qiange Wang, Yanfeng Zhang, Hao Yuan, Shufeng Gong, and Ge Yu. 2018. NeutronAscend: Optimizing GNN Training with Ascend AI Processors. 1, 1 (April 2018), 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Graph Neural Networks (GNNs) have rapidly emerged as a prominent method for analyzing graph-structured data, demonstrating exceptional capabilities across various application fields [5, 35, 42, 43]. In contrast to traditional deep neural networks (DNNs), GNNs leverage the relationships and structural information inherent in graph data, making them particularly effective for capturing complex dependencies. From a computational perspective, GNNs incorporate an interleaved execution phase that combines graph operations (sparse-dense matrix multiplication) in the aggregation phase with neural network (NN) operations (dense-dense matrix multiplication) in the update phase. By iteratively executing these two steps, the GNN model effectively captures structural information from multihop neighbors.

Training of GNN models typically involves hundreds of iterations over large-scale datasets, which imposes substantial demands on computational resources. Recently, a new generation of Neural Processing Units (NPUs), such as Huawei's Ascend AI processor [14], Google's TPU [12], Cambricon's MLU [18], and Intel's Gaudi processor [21], has emerged as a critical part of the computing infrastructure in data centers and AI platforms. These NPUs offer specialized hardware support for deep learning workloads and are typically optimized for matrix operations, high memory bandwidth, and energy-efficient performance. Among them, the Ascend NPU supports a variety of machine learning frameworks through its Compute Architecture for Neural Network (CANN) software stack, offering new possibilities for accelerating large-scale training tasks.

Efficient GNN training in complex hardware environments relies on the support of robust frameworks. The framework provides programming interfaces to users and manages the computation, storage, and network resources of the hardware environment. Serving as a bridge between data, models, and hardware, training frameworks play a crucial role in fully leveraging hardware capabilities. We evaluate the usage of hardware resources by different GNN training frameworks on Ascend AI processors and GPUs. Specifically, we compare NeutronStar [33], a popular GPU-based GNN training framework, with MindSpore Graph Learning (MindSporeGL), an Ascend-based GNN training framework developed on Mindspore, on various datasets in terms of per-epoch runtime and resource utilization. The details of the used graphs, test platform, and system configurations are given in Section 7. As shown in Figure 1, compared to DGL, MindSporeGL, as a naive GNN training framework, fails to fully exploit the hardware characteristics of Ascend AI processors to accelerate GNN training. MindSporeGL lags behind in training performance by an average factor of 3.71×, with a resource utilization gap of 4.2×. Notably, the computational power (FLOPS) of the Ascend 910B is comparable to that of the NVIDIA A5000, indicating that the performance disparity is due to the training framework rather than the hardware itself. Therefore, an efficient framework is crucial for any hardware environment and is a key focus of this work. This not only promotes the development of the NPUs ecosystem but also enhances the generalizability of GNN training frameworks across different hardware environments.

The vast majority of the computing power of the Ascend AI processor is provided by the AI Core [14]. The Ascend AI processor consists of multiple AI cores, each constructed on the DaVinci architecture [15] and features various computational units, including the Cube unit, Vector unit, and Scalar unit. Our experiments and analyses indicate that ensuring a balanced workload between

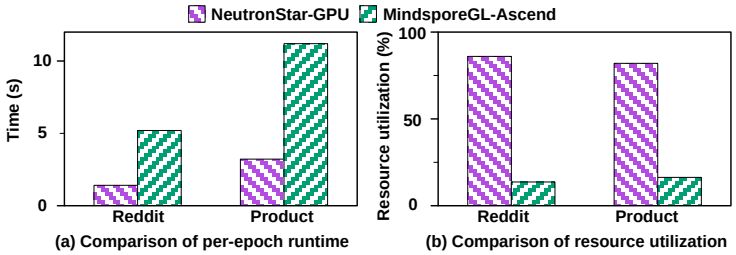

Fig. 1. Comparison of GNN Training Frameworks on Ascend AI Processors and GPUs. (**Model**: A 2-layer GCN; **Testbed**: Ascend 910B and Nvidia A5000; **Dataset**: Reddit and Product)

different AI cores and fully utilizing the various computational units within each AI core are crucial for the efficient use of the Ascend AI processor (see details in Section 3). Unlike traditional DNN training, GNN training involves graph aggregation operations, which pose unique challenges for accelerating GNN training on the Ascend AI processor. Firstly, due to the power-law distribution of graph data, achieving load balancing during workload partitioning across AI cores is challenging. Secondly, AI Cores that are originally optimized for dense computations struggle to efficiently handle the extensive sparse computations arising from the sparse structures of graph data. While our design and analysis are based on the Ascend AI processor, we observe that two fundamental challenges—inter-core workload imbalance and intra-core underutilization—are common across modern NPU architectures that share similar characteristics, such as physically decoupled compute units and heterogeneous memory hierarchies. These challenges are particularly pronounced when executing irregular workloads like GNNs, which exhibit sparse computation and non-uniform memory access patterns. As such, the techniques proposed in this work are broadly applicable to a wide range of NPU platforms, including TPUs, MLUs, and Gaudi.

In this paper, we efficiently accelerate GNN training on the Ascend AI Processors with two critical designs that simultaneously addresses inter-core workload imbalance and low intra-core resource utilization. At the inter-core level, we employ GNN tensor-data hybrid parallelism to manage task concurrency across AI Cores, combining GNN tensor parallelism for load balancing and GNN data parallelism to improve computational efficiency. Tensor parallelism partitions vertex features evenly across AI Cores, ensuring load balance, but may sacrifice efficiency with small feature dimensions. To address this, data parallelism is introduced for graph partitioning, reducing the frequency of feature slicing. AI Cores are grouped for tensor parallelism to maintain load balance and ensure sufficiently large feature slice sizes, while data parallelism is applied within these groups to optimize computational efficiency. At the intra-core level, we employ computation-aware task scheduling to allocate appropriate computational tasks to different units. Recognizing that Cube units are the primary computational resources of the DaVinci architecture, we assign graph aggregation operations and matrix multiplications in NN operations to the Cube units, while delegating vector operations in NN operations to the Vector units. Additionally, we partition and compress the original sparse graph to reduce redundant computations and irregular memory accesses incurred by the graph aggregation operations in Cube units. Finally, we employ inter-unit pipelining to overlap the computational overhead across different computation units and the transmission overhead of the MTE (Memory-Transfer Engine) unit, thereby further enhancing overall performance.

By integrating the above techniques, we propose NeutronAscend, an efficient GNN training system on the Ascend AI processor. In this paper, we make the following contributions.

- We perform a detailed analysis of the efficient use of the Ascend AI processor and identify the potential of the Ascend AI processor to accelerate GNN training.

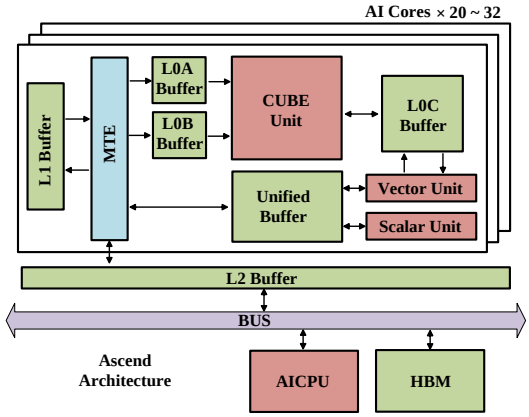

Fig. 2. The architecture of Ascend AI processer. Arrows represent data streams while boxes represent computing unit.

- We propose a GNN hybrid parallelism to manage task concurrency across AI Cores, combining GNN tensor parallelism for load balancing and GNN data parallelism to improve computational efficiency.
- We propose a computation-aware task scheduling that efficiently matches different computational stages of GNN training to the various computation units within the AI Core, maximizing resource utilization.
- We develop NeutronAscend, a GNN training system on the Asecnd AI processor that integrates two critical designs for both inter-core and intra-core performance optimization.

We evaluate NeutronAscend on an Ascend 910B processor. The experimental results show that NeutronAscend achieves average 4.71× speedups over the Mindspore Graph Learning framework on five real-world graphs. The rest of this work is organized as follows. Section 2 describes the background of GNN training and the Ascend AI processor. Section 3 describes the motivation. Section 4 gives an overview of NeutronAscend. Section 5 describes the GNN hybrid parallelism. Section 6 describes the computation-aware task scheduling. Section 7 presents the results. Section 8 presents a discussion of related works. Section 9 concludes the paper.

2 Background

2.1 Ascend AI Processor

Huawei's Ascend AI processor is designed with a specialized architecture for AI workloads, offering different SoC configurations that are suitable for a range of use cases [14]. The Ascend 910, dedicated to training, incorporates HBM2 along with unique computational units: AI Core and AI CPU. The AI Core, based on the DaVinci architecture [15] and illustrated in Figure 2, supplies most of the computational power of the chip. It includes three computation units: the Cube Unit that executes single-instruction multiplications of two 16×16 dense matrices in float16; the 2048 bit Vector Unit capable of handling 128 float16 elements in SIMD mode; and the Scalar Unit focused on scalar operations. The AI CPU, an ARM-based processor, handles essential operations that do not require high performance. Since the AI Core contributes the main computational capability of the Ascend processor, our primary focus is on enhancing performance there.

In terms of data transmission, the arrows in Figure 2 represent the direction of data streams. The AI Core reads or writes data from the HBM (global memory) via the BUS unit. The MTE (memory transfer engine) is responsible for managing the read and write operations of data within the AI

Core across different buffers, as well as performing various format transformations, such as padding and transposition. Taking fully connected layers and activation functions as an example, under optimal scheduling, data first enters the MTE unit and L2 buffers via the BUS from the HBM. It then passes through the L1, L0A and L0B buffers to the Cube unit. The results are transferred through the L0C buffer to the Vector unit for activation function computation. Finally, the processed data is written back to the HBM via the unified buffer. It is worth noting that any data within the AI Core that needs to be transmitted externally must pass through the unified buffer before being written back to the HBM. In summary, the AI Core involves extensive data transmission across multiple data buffers, a process that requires manual control by the programmer. We can optimize overall performance by using task scheduling to overlap computation with data transmission.

2.2 Graph Neural Networks

Graph-structured data are fed into graph neural networks (GNNs), where each node is characterized by a high-dimensional feature vector. Typically, GNNs derive low-dimensional embeddings for nodes across multiple layers, facilitating downstream tasks such as node classification and link prediction. Each layer comprises an aggregation phase followed by an update phase [48].

In a GNN with L layers, during the aggregation phase of layer l, each node v gathers embeddings from its predecessors at the $(l-1)$-th layer, along with its own, to generate a_v^l using a AGG function:

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

where $N_{in}(v)$ denotes the set of neighbors incoming from node v, h_v^l indicates the embedding vector of node v in the l-th layer, and h_v^0 is the initial feature of node v. Subsequently, during the update phase, each node calculates its resulting embedding vector h_v^l by applying a function $UPDATE$ to the aggregation result a_v^l:

$$h_v^l = UPDATE(W_v^l, a_v^l) \tag{2}$$

Upon completion of all L layers, the feature vector of each node is transformed into a low-dimensional embedding, reflecting the topological information of its neighbors across the L hops.

From the perspective of operation types, the GNN training process can primarily be formalized into three categories of operations: purse-density matrix multiplication (GEMM), sparse-matrix dense-matrix multiplication (SpMM), and vector operations. Specifically, the aggregation phase can be formalized as a SpMM process between the graph adjacency matrix (sparse matrix) and the vertex feature matrix (dense matrix). The update function in the update phase can be formalized as a pure dense GEMM between the vertex feature matrix (dense matrix) and the model parameter matrix (dense matrix). The update phase also includes some vector operations, such as the ReLU, Softmax, and loss functions.

2.3 GNN Parallelization Methods

GNN parallelism methods play a critical role in accelerating distributed GNN training. Similarly, task partitioning and parallel scheduling are also required among the AI Cores within Ascend AI processors. GNN data parallelism and GNN tensor parallelism are two common parallelization strategies that are widely used in many GNN training systems [1, 4, 7, 11, 19, 20, 29, 32, 33, 47, 49]. Here, we briefly introduce how they can be adopted and implemented in GNN training, offering guidance for task concurrency across AI Cores.

GNN Data Parallelism. Existing GNN systems [11, 19, 29, 32, 33, 47, 49] leverage data parallelism by partitioning the input graph and distributing it to multiple workers to train the same GNN

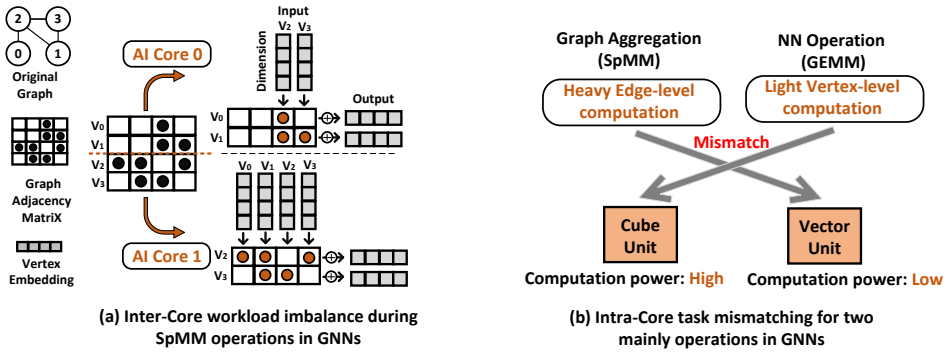

Fig. 3. An example of two challenges of GNN accelerating training on the Ascend AI processor. Note that in (a), "→" indicates loading data; "⊕" indicates neighbor embedding accumulation.

model collaboratively. Each worker holds a subgraph along with the corresponding vertex features. Workers can independently perform NN operations on their local vertices. However, during the graph aggregation, each worker must communicate with others to obtain remote neighbors.

GNN Tensor Parallelism. Existing GNN systems [1, 4, 7] leverage tensor parallelism by vertically slicing the vertex feature and distributing it along with the complete graph topology across workers. Workers can independently perform graph aggregation operations on their respective feature slices. However, during the NN operations, each worker must communicate with others to obtain the complete vertex features, as NN operations involve non-linear transformations that cannot be computed partially.

3 Motivation

In this section, we will discuss how to optimize GNN training by taking advantage of the hardware characteristics of the Ascend AI processor. We conduct experimental analysis using the MindSpore Graph Learning framework. The details of the used graphs, test platform, and system configurations are given in Section 7.

The GNN training involves random vertex data access and neural network computation, requiring high memory bandwidth and massive parallel computation. Considering the increasing sizes of real-world graphs, research efforts have been made toward extending GNN training to GPU platforms, leading to the design and widespread adoption of various GPU-accelerated GNN training frameworks [1, 17, 23, 29, 32, 33]. However, these GPU-specific optimizations cannot be directly applied to the Ascend AI processor due to its distinct hardware characteristics, which differ from those of GPUs. We focus our analysis on the AI cores of the Ascend processor, examining its hardware characteristics and the associated challenges in accelerating GNN training.

3.1 Challenge 1: Inter-Core Workload Imbalance

The manual workload distribution across AI Cores in the Ascend AI processor, coupled with the irregularity of graph data, leads to significant workload imbalance and underutilization of AI Cores, causing up to 70% idle time and substantial resource wastage during GNN training. In the Ascend AI processor, programmers must manually manage workload distribution among the AI Cores. Specifically, when implementing custom operators using Ascend C [14], programmers are required to partition data in global memory and assign specific data partitions to each AI Core during the `Init` function. In contrast, GPUs do not require inter-SM (i.e., Streaming Multiprocessor) data partitioning, as data and kernel functions are launched simultaneously to the device, with CUDA

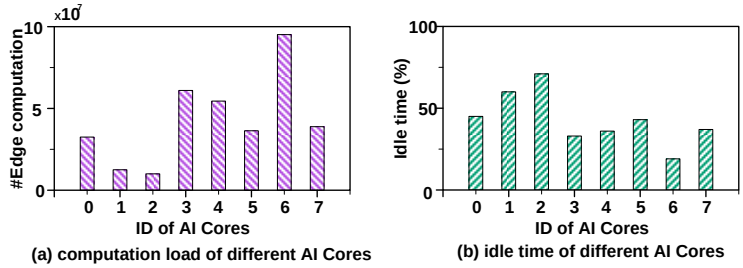

Fig. 4. Training workload on different AI Cores and the proportion of idle time for each AI Core during GNN training. (2-layer GCN on Reddit)

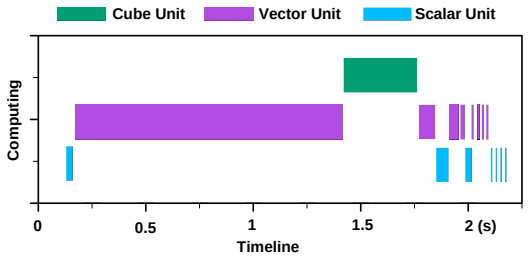

Fig. 5. Active timeline of different computation units in the forward pass of an epoch. (2-layer GCN on Reddit)

automatically managing data distribution. Consequently, manual data partitioning across AI Cores introduces new challenges, it also offers opportunities to enhance overall computational efficiency through fine-grained workload management.

Load balancing is a primary objective in data partitioning. However, as illustrated in Figure 3 (a), the irregular nature of graph data makes it challenging to ensure load balance during the workload partitioning process. Distributed GNN training frameworks [7, 11, 29] and many survey papers [16, 22, 26, 44] highlight workload imbalance as a primary challenge in distributed GNN training. Imbalanced workloads can cause AI cores with lighter loads to remain idle during computation. We evaluate the workload distribution and idle time across different AI Cores using the Mindspore Graph Learning framework. The results are shown in Figure 4. We quantify the computational load by counting the number of edges involved in the computation. Our observations reveal that the computational load can vary by up to 7.4× between AI Cores. Moreover, the imbalance of workload results in AI Cores being idle for as much as 70% of the total runtime, leading to significant resource wastage.

3.2 Challenge 2: Intra-Core Low Resource Utilization

The current task mapping in the MindSpore Graph Learning framework leads to inefficient resource utilization, as the Vector unit is overloaded with intensive graph aggregation operations while the Cube unit remains underutilized during GNN training. As described in Section 2.2, the AI Core features a novel computational architecture, the DaVinci architecture [15], which consists of multiple computation units. The Cube unit serves as the key computation engine of the AI Core, providing efficient matrix multiplication. The Vector unit offers a broader range of computational instructions, supporting most types of computation. The Scalar unit handles loop control, branching, and basic arithmetic operations. To fully exploit the computational capabilities of each unit and mitigate potential bottlenecks in any single unit, it is essential to alllocate the computational processes associated with GNN training to the appropriate units.

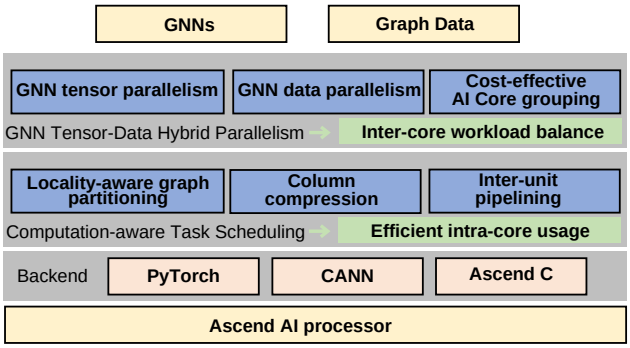

Fig. 6. NeutronAscend system overview.

The GNN training process involves multiple types of computations, making task mapping a non-trivial task. Graph aggregation operations primarily include sparse-matrix dense-matrix multiplication (SpMM), while NN operations mainly consist of purse dense Matrix Multiplication (GEMM), and vector operations. Generally, graph aggregation operations, as edge-level computations, constitute the primary overhead in GNN training, while NN operations are vertex-level computations [34, 38]. As illustrated in Figure 3 (b), in the default implementation of the MindSpore Graph Learning framework, NN operations are assigned to the Cube unit, whereas graph aggregation operations are assigned to the Vector unit. Although the computational power of the Vector unit is lower than that of the Cube unit, it is tasked with more intensive computational operations. This mismatch between computation units and their corresponding computational tasks leads to poor resource utilization of the AI Cores. We evaluate the time distribution across different computation units in MindSpore Graph Learning framework and find that SpMM on the Vector unit is the dominant overhead in the overall training process. The results are shown in Figure 5. The computational cost on the Vector unit accounts for 76% of the total training process, leaving the Cube unit frequently idle, which leads to resource wastage.

4 The NeutronAscend Framework

We present NeutronAscend, an efficient GNN training system specifically designed to leverage the hardware characteristics of the Ascend AI processor. We address the challenges outlined in Section 3 through two critical functions. Figure 6 provides an architectural overview of NeutronAscend.

GNN tensor-data hybrid parallelism. GNN tensor-data hybrid Parallelism combines GNN tensor parallelism for load balancing and GNN data parallelism for improving computational efficiency. GNN tensor parallelism involves evenly partitioning vertex features so that different AI Cores handle training using equal-sized feature slices, maintaining load balance but potentially sacrificing efficiency if feature dimensions are too small. To mitigate these constraints, GNN data parallelism is introduced for graph partitioning, which helps minimize the need for frequent feature partitioning. To further balance the trade-off between computational load imbalance and the benefits of L2 cache reuse, we introduce cost-effective AI Core grouping, which adaptively determines the number of AI Core groups based on a time-normalized cost model. This allows each group to maintain sufficiently large feature slices for compute efficiency while maximizing vertex data reuse within the group to reduce memory access overhead.

Computation-aware task scheduling. Computation-aware task scheduling is specifically designed for the DaVinci architecture of AI Core. This method focuses on the efficient utilization of

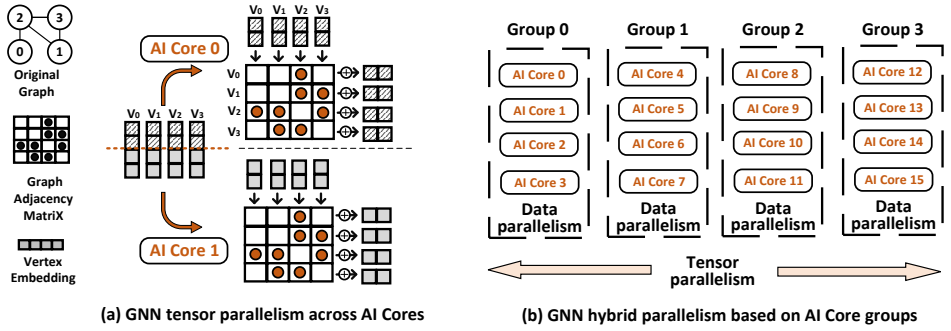

Fig. 7. An illustrative example of GNN tensor parallelism and GNN hybrid parallelism.

the Cube unit, assigning the Cube unit to handle both GeMM and SpMM computations in GNN training, while the Vector unit is dedicated solely to vector operations (such as softmax and ReLU). The task scheduling method consists of two core components: a locality-aware graph compression approach and an inter-unit pipeline. The locality-aware graph compression approach reduces redundant computations and irregular memory access during SpMM processing by the Cube unit. Specifically, we use a greedy algorithm to partition the graph data into multiple chunks, ensuring that vertices with common neighbors are placed in the same chunk whenever possible. We then apply column compression to each chunk, removing non-neighboring indices. The compressed chunks are scheduled sequentially for computation. The inter-unit pipeline concurrently schedules multiple chunks to facilitate the overlap of operations across different computation and transfer units, thereby enhancing overall performance.

5 GNN tensor-data hybrid Parallelism

This section introduces and explains GNN tensor-data hybrid parallelism, discusses its implementation details and advantages, and provides an insightful look into how combining tensor and data parallelism can enhance GNN training performance.

5.1 Tensor Parallelism: Load Balancing Across AI Cores

Avoiding partitioning the graph structure across different AI Cores is crucial to mitigating the workload imbalance issues outlined in Section 3.1. Therefore, we employ GNN tensor parallelism for task concurrency across AI Cores, partitioning vertex features instead of the graph structure. As shown in Figure 7 (a), GNN tensor parallelism vertically partitions vertex features along dimensions, where each AI Core is responsible for the complete graph GNN training of the equal-sized feature slices, ensuring complete load balance. Specifically, vertex features are divided based on the number of AI Cores, with each AI Core handling a feature slice of size $\frac{D}{N}$, where N is the total number of AI Cores and D is the feature dimension. The feature slice assigned to each AI Core is determined by its ID; for example, AI Core 1 is responsible for the feature slice $[\frac{D}{N}, \frac{2D}{N} - 1]$.

In distributed GNN training, GNN tensor parallelism faces two major challenges related to communication and memory overhead [1]. First, distributed GNN tensor parallelism requires frequent collective communication for gathering and splitting embeddings by dimensions between NN computation (requiring the embedding split by vertices) and graph propagation (requiring the embedding split by dimensions), which results in substantial layer-wise data communication overhead. Furthermore, distributed GNN tensor parallelism requires replicating the entire graph topology across all machines, introducing additional memory overhead. In contrast, these challenges can be

Algorithm 1 Cost-effective AI Core Grouping

Input: Number of AI Cores N, Feature Dimension D, Bandwidth Ratio r, Estimate $T_{\text{comp}}, T_{\text{mem}}$
Output: Optimal Number of Groups M^*

$M^* \leftarrow 1$, minCost $\leftarrow \infty$
for $M \in \{1, 2, 4, ..., N\}$ **do**
  $S \leftarrow D/M$
  pad $\leftarrow (16 - (S \bmod 16)) \bmod 16$
  $T_{\text{idle}} \leftarrow \frac{\text{pad}}{S} \cdot T_{\text{comp}}$
  $G \leftarrow N/M$
  $T_{\text{saved}} \leftarrow T_{\text{mem}} \cdot \left(1 - \frac{1}{1+(G-1)(r-1)}\right)$
  Cost $\leftarrow T_{\text{idle}} - T_{\text{saved}}$
  if Cost < minCost **then**
    minCost $\leftarrow$ Cost, $M^* \leftarrow M$
Return M^*

efficiently solved by sharing data in global memory during AI Core parallel training. The computation results of each GNN layer are written back to global memory of the Ascend AI processor, allowing different AI Cores to access the data on demand based on the current computation stage. On the other hand, the graph topology is only stored once in global memory and can be shared by all AI Cores.

5.2 Data Parallelism: Improving Computational Efficiency

When the vertex feature dimension is small, fully partitioning the vertex features across all AI Cores in GNN tensor parallelism may reduce computational efficiency. The Cube unit requires the input tensor dimensions to be 16×16, and any dimension smaller than 16 is padded with zeros [15]. Therefore, when the feature slice dimension cannot be evenly divided by 16, GNN tensor parallelism results in computational waste. This waste occurs in the last scheduling unit, where the dimension is $N/D \mod 16$. When the feature slice dimension is small, such computational waste becomes significant. To address this issue, we combine GNN tensor parallelism with GNN data parallelism, partitioning the graph data simultaneously and adjusting the feature slice dimensions to be as close as possible to multiples of 16.

We implement hybrid parallelism by grouping the AI Cores into M groups, with each group containing an equal number of AI Cores. As shown in Figure 7 (b), tensor parallelism is applied across groups to ensure that the feature slice dimension D/M is as close as possible to a multiple of 16. Data parallelism is employed within each group to enhance computational efficiency. We employ a range-based graph partitioning strategy [1, 32] to partition the graph data among the AI Cores within each group. This lightweight partitioning method divides the graph into subgraphs, where each subgraph consists of a series of consecutively indexed vertices and their associated incoming edges, enabling independent full-neighbor aggregation within each subgraph. All AI Core groups share a unified graph partitioning strategy.

5.3 Cost-effective AI Core Grouping

To balance computation throughput and memory access efficiency, we design a cost-effective AI Core grouping strategy to determine the number of groups M for GNN tensor-data hybrid parallelism. This grouping critically impacts two key aspects: (1) Inter-group tensor parallelism improves load balancing across AI Cores by evenly partitioning vertex features; (2) Intra-group data parallelism enhances memory efficiency by allowing AI Cores within the same group to share

feature and embedding data via L2 cache. While tensor parallelism prefers a larger number of groups (i.e., fewer AI Cores per group) for fine-grained load balancing, data parallelism prefers fewer groups (i.e., more AI Cores per group) to maximize cache reuse of high-volume vertex data.

Design Trade-off In GNN training, vertex-related data, including features, embeddings, and intermediate results, dominate the memory footprint—often accounting for more than 90% of total memory usage [32]. In contrast, graph topology data (e.g., CSR indices or edge lists) typically contributes only a small fraction. Therefore, enabling L2 cache sharing of vertex data across AI Cores in the same group is substantially more beneficial than merely sharing the graph structure. When the number of groups M is small, more AI Cores share a common slice of vertex data, and can efficiently reuse it via L2 cache. The L2 cache on Ascend provides up to 20× bandwidth compared to HBM [14], allowing faster memory access and significantly reducing memory transfer latency. Conversely, when M is large, although graph topology can still be shared, the reduced reuse of high-volume vertex data results in higher memory access overhead. Simultaneously, tensor parallelism introduces another cost when D/M is not divisible by 16. The padding incurred by misaligned feature slices leads to computational imbalance and AI Core idleness, which is equivalent to wasted compute time.

Time-based Cost Model To unify the trade-off between computation imbalance and memory access efficiency under a common metric, we develop a time-based cost model that quantifies both sources of overhead in terms of execution time. Specifically, when the feature slice dimension $S = D/M$ is not divisible by 16, the Cube unit must perform padded computations, resulting in idle compute cycles. We estimate this loss as $T_{\text{idle}}(M) = \frac{(16-(S \bmod 16)) \bmod 16}{S} \cdot T_{\text{comp}}$, where T_{comp} denotes the estimated time for one iteration of ideal computation. On the other hand, increasing the number of AI Cores within a group allows more cores to reuse the same vertex feature and embedding data via L2 cache, thereby reducing the volume of HBM memory access. We quantify this time saving as $T_{\text{saved}}(M) = T_{\text{mem}} \cdot \left(1 - \frac{1}{1+(G-1)\cdot(r-1)}\right)$, where $G = N/M$ is the number of AI Cores in a group, $r = 20$ is the L2-to-HBM bandwidth ratio, and T_{mem} denotes the estimated time of memory transfers. The overall cost is thus expressed as the difference between compute time loss and memory time gain: $\text{Cost}(M) = T_{\text{idle}}(M) - T_{\text{saved}}(M)$.

Algorithm Description Based on the time-based cost model, we design a search algorithm to determine the optimal number of AI Core groups M^*. In practice, the optimal AI Core grouping must be determined individually for each GNN layer, as the dimensionality of vertex features varies across layers. The algorithm iteratively evaluates candidate group sizes, where each group size $M \in \{1, 2, 4, ..., N\}$ evenly divides the total number of AI Cores. For each candidate M, it computes the feature slice size $S = D/M$, estimates the computational time loss T_{idle} caused by padding misalignment, and evaluates the memory access time reduction T_{saved} resulting from vertex data reuse in L2 cache. The total cost is defined as the difference between the two. The configuration that yields the lowest cost is selected as the optimal grouping strategy. This search process efficiently balances the trade-off between inter-core load balancing and intra-group memory reuse, adapting the grouping decision to both model dimensions and hardware characteristics.

5.4 Shared Memory for Data Synchronization

We allocate shared memory in global memory to store the computation results of each GNN layer for data synchronization. The size of the shared memory for each layer is $V \times D \times$ sizeof(float16), where V is the number of vertices and D is the vertex feature or embedding dimension. After each AI Core group completes the graph aggregation operation, the aggregated feature slices are written to the shared memory area. Each AI Core group writes a slice of the feature for all vertices at fixed

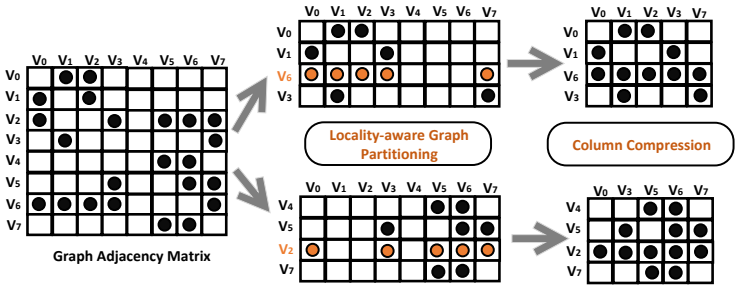

Fig. 8. An illustrative example of Locality-aware Graph Compression.

intervals ($D - D/M$). Once the aggregation results are written, each AI Core group reads the same number of consecutive vertex data for NN operations and then writes the computed embeddings back to the corresponding vertex positions. The computed results of this GNN layer serve as input for the next layer. When the next layer starts the graph aggregation operation, it reads the corresponding vertex data based on the incoming neighbor set of the vertices. In implementation, we use the Ascend C `DataCopy()` function to perform data writing and reading. The `DataCopy()` function supports both continuous and fixed-interval read/write operations. By establishing shared memory for the computation results of each GNN layer, we enable data synchronization across different AI Cores.

6 Computation-aware Task Scheduling

This section presents a computation-aware task scheduling method designed specifically for the DaVinci architecture of AI Cores. The proposed method targets the efficient utilization of the Cube and Vector units, which have distinct functionalities, and aims to optimize data access and computation flow. Our approach leverages two core components: locality-aware graph compression and an inter-unit pipeline, both of which play critical roles in managing the computational workload in GNN tasks.

6.1 Basic Design

To address the computational bottleneck in the Vector unit described in Section 3.2, we offload graph aggregation (sparse matrix multiplication) to the Cube unit, which serves as the primary computational resource. This allows us to reserve the Vector unit for vector operations such as softmax and ReLU. However, our initial studies indicate that naively applying the Cube unit for sparse matrix multiplication can even result in performance degradation compared to the vertex-centric implementation on the Vector unit. This offload of computations introduces two challenges. First, due to the high memory costs ($O(V^2)$, where V is the number of vertices), it is impractical to directly handle sparse matrix multiplication as a dense GEMM operation on the Cube unit. Second, processing sparse matrices filled with zero elements leads to excessive redundant computations and unnecessary memory accesses.

6.2 Locality-aware Graph Compression

To address these challenges, we employ a locality-aware graph compression method that minimizes redundant computations caused by zero elements while also reducing the data transfer volume between local memory and global memory.

Locality-aware Graph Partitioning. NeutronAscend partitions the input graph into smaller execution units suitable for processing with the local memory of single Cube unit. However,

Algorithm 2 Greedy Algorithm of Locality-aware Graph Partitioning

Input: $G = (V, E)$: the input graph, k: the number of chunks
Output: $C = \{C_1, C_2, \ldots, C_k\}$: a partition of V into k chunks
1: Initialize $C \leftarrow \emptyset$, $chunk\_size \leftarrow \lceil |V|/k \rceil$
Phase 1: Initialize one vertex for each chunk
2: Evenly split vertices $V = \{v_1, v_2, \ldots, v_n\}$ into k contiguous intervals $\{I_1, I_2, \ldots, I_k\}$
3: **for** $i \leftarrow 1$ to k **do**
4: Select the vertex $v_i \in V$ such that $\arg\max_{v \in V} (|\text{neighbors}(v) \cap I_i|)$
5: $C_i \leftarrow \{v_i\}$
6: Mark v_i as visited
Phase 2: Assign to each chunk the most neighbor-shared vertices
7: **for** $i \leftarrow 1$ to k **do**
8: $U_i \leftarrow V \setminus (C_1 \cup C_2 \cup \cdots \cup C_k)$
9: $\forall u \in U_i, \quad \text{shared\_neighbors}(u, C_i) = |\text{neighbors}(u) \cap \bigcup_{v_j \in C_i} \text{neighbors}(v_j)|$
10: $U_i' = \text{Top-}(\text{chunk\_size} - 1, \text{shared\_neighbors}(u, C_i))$
11: $C_i \leftarrow C_i \cup U_i'$
12: Mark u as visited for each $u \in U_i'$
13: **if** any unvisited vertices remain **then**
14: Distribute unvisited vertices evenly across unfilled chunks in C
return C

sequentially scheduling subgraphs with duplicated neighbors can result in repeated transmission of the same vertex data. To address this issue, NeutronAscend employs a locality-aware graph partitioning strategy, which divides the graph into multiple chunks, with each chunk comprising vertices that share a common neighborhood. This ensures that most computations within each chunk remain highly localized. By employing this method, we significantly reduce the data transfers between the Cube unit's local memory and global memory, which are typically costly in terms of latency. NeutronAscend employs a greedy algorithm for partitioning the graph. As described in Algorithm 2, the greedy method consists of two phases. The algorithm begins by selecting k vertices as seed vertices, one for each chunk, ensuring that each chunk starts with a vertex that has the most neighbors within its respective interval (lines 2-7). After initializing the seed vertices, the algorithm fills each chunk by selecting the top chunk_size − 1 vertices that share the most neighbors with those already assigned to the chunk (lines 8-13). The unvisited vertices are then distributed evenly across the remaining chunks (lines 15-16). The result is a partition of the graph into k chunks, each containing vertices with high neighbor overlap, improving locality for the GNN training.

Column Compression. After the graph is partitioned, we apply column compression to each chunk. The goal of column compression is to remove the indices of non-neighboring vertices that are irrelevant to the current computation, allowing the Cube unit to focus solely on processing the relevant neighboring vertices, thereby significantly reducing memory usage and computational load. Notably, locality-aware graph partitioning enhances the effectiveness of column compression, as vertices within a chunk tend to share many common neighbors. The total time complexity of the locality-aware graph partitioning and column compression is $O(E + V \log V)$, which scales well with large graphs. We reuse graphs in each training iteration, so the cost of the algorithm is easily amortized. Figure 8 presents an example of locality-aware graph partitioning and column compression applied to a toy graph with only 8 vertices. First, the locality-aware graph partitioning divides the graph data into two chunks, with V_2 and V_6 serving as the seed vertices for each chunk. The remaining vertices, which share the most neighbors with these seed vertices, are assigned to

the corresponding chunk. Next, column compression removes columns without non-zero elements from both chunks, effectively excluding column indices of non-neighboring vertices. The resulting output consists of partitioned and compressed chunks. Compared to the original graph, these chunks exhibit reduced memory usage and enhanced computational locality.

It is worth noting that the locality-aware graph compression and the range-based graph partitioning used in GNN data parallelism can be applied together without any conflict. The purpose of the range-based graph partitioning is to determine the subgraph assigned to each AI Core within a group. On the other hand, the locality-aware graph compression is responsible for providing the scheduling units (the partitioned and compressed chunks) for the computation unit inside the AI Core when processing the graph data. In practice, we first apply locality-aware graph compression to process the graph data, and then use range-based graph partitioning to assign tasks to each AI Core in the data parallelism phase. Therefore, the range-based graph partitioning is performed at the chunk level rather than the vertex level, with each AI core being responsible for an equal number of contiguous chunks.

6.3 Inter-unit Pipelining

Based on locality-aware graph compression, we can further overlap the computation and data transfer overheads of different chunks across various units. For example, while the Cube unit processes one chunk computation, the Vector unit can concurrently begin computing on the previous chunk's result, and the MTE unit can parallelly transfer the next chunk from global memory to local memory. In practice, we use `Queue` to enforce the sequential execution of different tasks within a single chunk, while the concurrency of multiple chunks is managed by setting the parameter `BUFFER_NUM`. The `Queue` employs a blocking mechanism to control the reading and writing of computation results in the local memory of different units, thereby effectively managing synchronization between the transmission and computation units. After organizing the computation and data transfer tasks for a single chunk using the `Queue`, we can configure the `BUFFER_NUM` parameter to specify the number of concurrently working queues. Ascend C will automatically overlaps computation and data transfer of different queues. The recommended implementation scheme is `double buffer`, with `BUFFER_NUM` set to 2. This configuration allows the use of at most half of the local memory for each chunk, enabling the system to process one chunk while simultaneously reading or writing another from local memory.

6.4 Overall Execution Flow in NeutronAscend

Algorithm 3 outlines the overall execution flow. To begin with, NeutronAscend employs a locality-aware graph compression method to partition the original graph data into multiple chunks, with each chunk's training memory usage not exceeding half of the local memory available on an AI Core. Each chunk, denoted as G_k, consists of a set of vertices V_k and their corresponding incoming neighbor set N_k, where k is the chunk ID (line 1). On the other hand, NeutronAscend evenly partitions the vertex features into feature slices $h\_cut_i^0$ according to the number of AI Core groups performing tensor parallelism, where i denotes the group ID (line 2). Subsequently, the forward pass begins, utilizing feature slices for tensor parallelism across groups and performing data parallelism within each group using chunks (lines 3-7). Within each group, each AI Core determines the set of chunks it is responsible for based on a range-based partitioning strategy (line 6). Through L iterations, NeutronAscend aggregates features from the incoming neighbor sets N_k to compute the embeddings for the destination vertices V_k (line 7). After completing the forward pass, the downstream tasks (vertex classification, link prediction) are initialized, and gradients are computed (lines 8-9). The backward pass reverses the forward propagation process, iteratively propagating

Algorithm 3 Workflow of NeutronAscend for a single epoch

Input: Graph $G(V, E)$, Feature $\mathbf{h}^0$, Initial parameterized GNN layers $\{\mathbf{W}^0, \mathbf{W}^1 \dots \mathbf{W}^{L-1}\}$, group number m, AI Core number in each group n, chunk number v
Output: Updated parameterized GNN layers $\{\mathbf{W}^0, \mathbf{W}^1 \dots \mathbf{W}^{L-1}\}$

1: $\{G_k | 0 \le k < v\}$=**Locality-aware_Graph_Compression**(G, v)
2: $\{\mathbf{h\_cut}_i^0 | 0 \le i < m\}$=**Feature_Split**$(\mathbf{h}^0, m)$
3: **for** layer $l = 0$ to $L - 1$ **do**
4: **for** AI Core Group $i \leftarrow 0$ to $m - 1$ **do in tensor parallel**
5: **for** AI Core $j \leftarrow 0$ to $n - 1$ **do in data parallel**
6: **for** chunk $k \leftarrow j * \frac{v}{n}$ to $(j + 1) * \frac{v}{n} - 1$ **do**
7: $\mathbf{h\_cut}_i^{l+1}$ of $V_k \leftarrow$`Group(i).AI Core(j).forward`$(\mathbf{W}^l, \mathbf{h\_cut}_i^l$ of $N_k, G_k)$
8: **loss** = `downstream_task`$(\mathbf{h}^L)$
9: $\nabla\mathbf{h}^L$ = **loss**`.backward()`
10: **for** layer $l = L - 1$ to 0 **do**
11: **for** AI Core Group $i \leftarrow 0$ to $m - 1$ **do in tensor parallel**
12: **for** AI Core $j \leftarrow 0$ to $n - 1$ **do in data parallel**
13: **for** chunk $k \leftarrow j * \frac{v}{n}$ to $(i + 1) * \frac{v}{n} - 1$ **do**
14: $\nabla\mathbf{h\_cut}_i^l$ of $N_k \leftarrow$`Group(i).AI Core(j).backward`$(\mathbf{W}^l, \nabla\mathbf{h\_cut}_i^{l+1}$ of $V_k, G_k)$
15: **for** layer $l = 0$ to $L - 1$ **do**
16: `sync_and_update` $(\mathbf{W}^l)$ //parameter update

Table 1. Dataset description. $|V|$, $|E|$, $\#\mathbb{F}$, $\#\mathbb{L}$, and $\#\mathbb{H}$ represent the number of vertices, edges, features, labels, and hidden layer dimensions respectively.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\#\mathbb{F}$	$\#\mathbb{L}$	$\#\mathbb{H}$
Citeseer [25]	3.32K	9.46k	3703	6	256
Cora [25]	2.71K	10.85k	1433	7	256
Pubmed [25]	19.71K	88.67K	500	3	256
Reddit [8]	232.96K	114.61M	602	41	256
Products [9]	2.4M	61.9M	100	47	256

the gradients from the destination vertices back to the source vertices across L hops (lines 10-14), and finally updates all model layers' weights W (lines 15-16).

7 Evaluation

7.1 Experimental Setup

Environments. We implement NeutronAscend on top of PyTorch v2.1.0, using dynamic graph compilation. The experiments are conducted on a server equipped with an Intel Xeon Platinum 8163 CPU (96 cores and 736 GB main memory) and a Ascend 910B (30 AI Cores and 32GB global memory) NPU. The Ascend 910B NPU is enabled with CANN v8.0 runtime. The server runs Ubuntu 18.04 with Linux kernel 4.13.0.

Datasets and GNN algorithms. Table 1 lists five popular GNN datasets that we used in our evaluation. Cora, Citeseer and Pubmed are citation networks [25] with a relatively small number of nodes and edges but contain rich node embedding information. Reddit [8] is a post-to-post graph. Products [9] is a co-purchasing graph. We use two popular GNN models with different computation patterns, GCN [13] and GAT [28]. All of them are in a 2-layer structure. The vertex

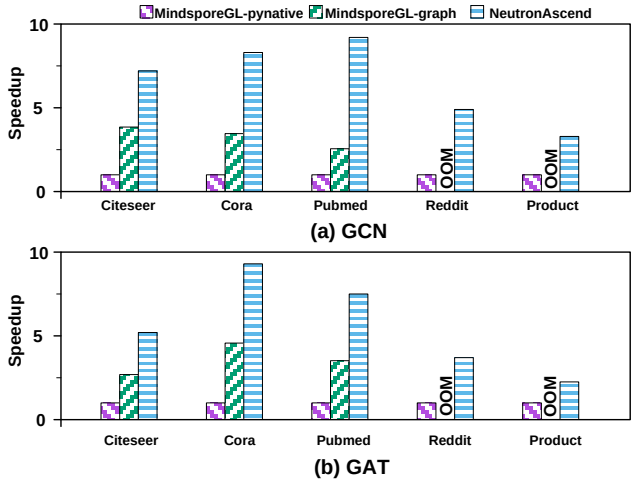

Fig. 9. The end-to-end speedup compared to MindsporeGL. (OOM represents out-of-memory)

feature dimensions, hidden layer dimensions, and the number of labels of datasets are listed in Table 1.

Competitor systems. In performance evaluation, we compare NeutronAscend with MindsporeGL, an Ascend-based GNN training framework deveploped on Mindspore v2.3.1. The first baseline, denoted as MindSporeGL-pynative, uses dynamic graph compilation as PyTorch. The second baseline, MindSporeGL-graph, adopts static graph compilation and applies operator fusion. Unless explicitly stated, the results are reported in per-epoch runtime, i.e., the time to conduct a forward and backward pass for all vertices in the graph. Shorter per-epoch time implies better time-to-accuracy performance. All the per-epoch runtime results are measured by averaging results of 100 epochs.

7.2 Overall Comparison

Compared to MindSporeGL, NeutronAscend consistently demonstrates superior performance across all datasets, achieving an average 4.71× speedup. As described in Section 3, MindSporeGL suffers from imbalanced workloads and inefficient Cube Unit utilization, which limit its performance. Although MindSpore offers static graph compilation optimizations compared to PyTorch, its static graph mode fails to scale to large datasets effectively. Specifically, MindSporeGL allocates a contiguous memory space for storing the neighbor data of each vertex in every model layer. While this reduces the random access overhead caused by graph aggregation, it results in significant redundant storage, as shared neighbors of different vertices are stored multiple times. For large datasets, maintaining neighbor data leads to frequent memory swapping, as the total memory demand exceeds the capacity of the Ascend AI processor. In such cases, the static graph mode of MindSporeGL fails to correctly pre-allocate and manage memory, leading to out-of-memory (OOM) errors. In contrast, NeutronAscend performs graph aggregation using the Cube Unit, storing only a single copy of the graph topology matrix and vertex embedding matrix, while applying column compression to the graph topology matrix to minimize memory overhead. Notably, on small datasets such as Citeseer, Cora, and PubMed, NeutronAscend demonstrates higher speedup due to the reduced overhead of subgraph chunks scheduling between the local buffer and global buffer, resulting in a more favorable compute-to-memory access ratio. In summary, NeutronAscend achieves superior performance through three key factors: hybrid parallelism ensures balanced

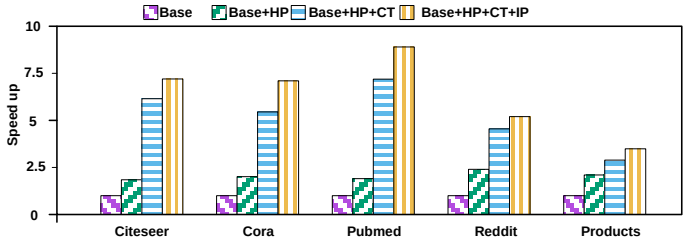

Fig. 10. Performance gain analysis. "HP" indicates the GNN tensor-data hybrid parallelism, "CT" indicates the computation-aware task scheduling, and "IP" indicates the inter-unit pipelining.

workload distribution, task scheduling maximizes Cube Unit utilization, and reduced memory swapping overhead enhances overall efficiency.

7.3 Performance Gain Analysis of NeutronAscend

We analyze the performance gain of GNN tensor-data hybrid parallelism (HP), computation-aware task scheduling (CT), and inter-unit pipelining (IP) on the GCN model with five datasets. To ensure a fair comparison, we start with a data parallelism baseline based on the NeutronAscend codebase and gradually integrate the three optimization methods. The data parallelism baseline employs the same vector unit-based, vertex-centric graph aggregation operator as MindSporeGL. Figure 10 shows the normalized speedups. Compared to the baseline, HP achieves speedups ranging from 1.84× to 2.40× by implementing a more balanced workload. On the Reddit dataset, HP achieves the highest speedup, attributed to its inherent power-law distribution as a social network graph. The data parallelism strategy exacerbates severe workload imbalances in such graphs. Compared to the baseline+HP, CT achieves speedups ranging from 1.37× to 3.33× by fully utilizing the Cube unit. CT achieves only a 1.37× speedup on the product dataset, which contains the largest number of vertices, as the substantial overhead of subgraph chunks scheduling partially offsets the performance gains brought by utilizing the Cube units. After transferring subgraph chunks (stored in CSR format) to the L2 cache, CT performs two preprocessing steps: matrix densification and vertex embedding gathering. Specifically, to support matrix multiplication on the Cube units, the CSR format is first converted to COO format for dense computation, and the gather operation reorganizes the randomly scattered vertex embeddings (after column compression) into a contiguous memory layout. Lastly, IP provides speedups ranging from 1.14× to 1.30× by overlapping the computational overhead across different computation units and the transmission overhead of the MTE (Memory-Transfer Engine) unit.

7.4 Sensitivity Study

Performance with varying hidden layer dimensions. In this experiment, we compare NeutronAscend with MindsporeGL when training GCN with different hidden layer dimensions over two datasets. The results are shown in Figure 11. We observe that the performance advantage of NeutronAscend over MindsporeGL gradually increased with the dimensions of the hidden layer. For hidden layer dimensions of 32, NeutronAscend achieves an average 1.97× speedup compared to MindSporeGL. At 64, 128, and 256 dimensions, NeutronAscend achieves an average speedup of 3.51×, 3.75×, and 4.08×, respectively. This is because, as the hidden layer dimension increases, the memory overhead for storing vertex neighbor data in MindSporeGL also grows. Once it exceeds the Ascend AI processor's memory capacity, memory swapping overhead occurs, further degrading performance. In practice, when the hidden layer dimension exceeds 32, the Ascend 910B processor's

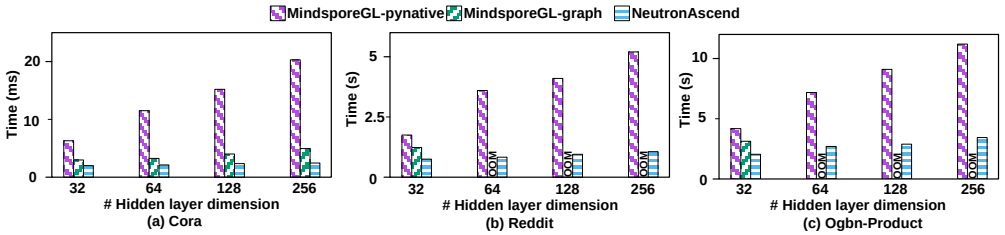

Fig. 11. Per-epoch runtime of different systems with different hidden layer dimensions. (OOM represents out-of-memory)

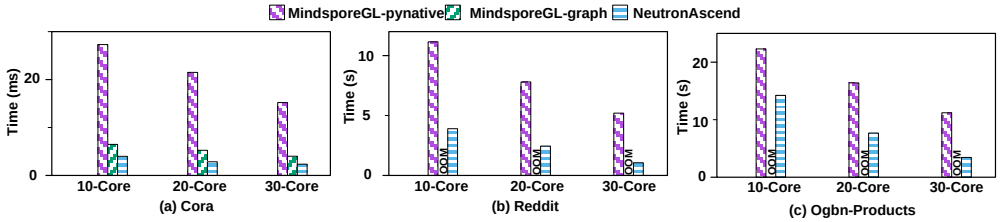

Fig. 12. Per-epoch runtime of different systems with different AI Cores. (OOM represents out-of-memory)

32GB memory becomes insufficient to store all vertex neighbor data, as shared neighbors of different vertices are redundantly stored multiple times. Additionally, as the hidden layer dimension increases, the total runtime of NeutronAscend does not exhibit a significant rise. This is because larger dimensions improve the compute-to-memory access ratio of the Cube Unit, enhancing data reuse efficiency and overall computational performance.

Performance with varying AI Cores. In this experiment, we compare NeutronAscend with MindsporeGL when training GCN with different AI Cores over two datasets. We adjust `BLOCK_NUM` to control the number of AI Cores participating in the computation. The results are shown in Figure 12. We observe that the performance advantage of NeutronAscend over MindsporeGL gradually increased with the number of AI Cores. When 10 AI Cores are used, NeutronAscend achieves an average 3.22× speedup. With 20 and 30 AI Cores, the speedups increase to 3.67× and 4.08×, respectively. This improvement is due to the imbalance in MindSporeGL's vertex-centric graph aggregation computation pattern, where each vertex has a highly variable aggregation workload (i.e., different numbers of neighbors), making it difficult to evenly distribute workloads across AI Cores. In contrast, NeutronAscend's tensor-data hybrid parallelism evenly partitions vertex data across AI Core groups, achieving effective load balancing and preventing some AI Cores from remaining idle.

7.5 AI Core Utilization

We evaluate the AI Core utilization during the training of GCN on five datasets for NeutronAscend and MindsporeGL-pynative. We use the `msprof` profiling tool to evaluate AI Core utilization. By default, `msprof` reports the utilization of Cube units as a proxy for overall AI Core utilization, since Cube units account for more than 90% of the total compute capacity. Figure 13 shows the results. NeutronAscend exhibits higher AI Core utilization (88.92% on average) compared to MindsporeGL (16.75% on average). Compared to MindSporeGL, which utilizes only the Cube Unit for NN computation, NeutronAscend fully leverages the Cube Unit to accelerate both graph

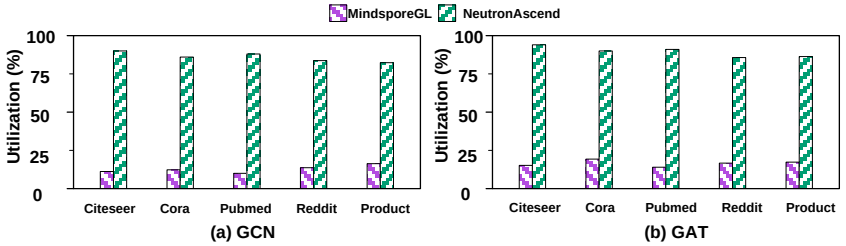

Fig. 13. AI Core utilization comparison.

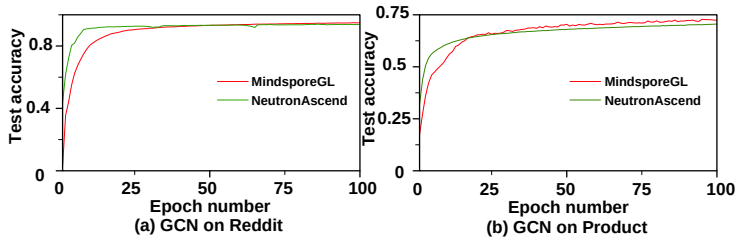

Fig. 14. Epoch-to-accuracy.

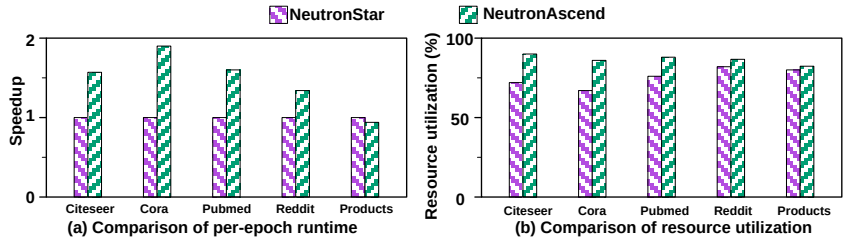

Fig. 15. Comparison with GPU-based GNN Training Framework.

aggregation and NN computation. Additionally, NeutronAscend incorporates parallel and task scheduling strategies to balance workloads across different AI Cores, while mitigating irregular memory access and redundant computations in the Cube Unit during SPMM.

7.6 Accuracy Comparisons

We plot the epoch-to-accuracy curve on different systems for a GCN model over two datasets. The results are shown in Figure 14. After 100 epochs, the test accuracy reaches a stable state, NeutronAscend and MindsporeGL achieve nearly identical test accuracy, as the techniques of NeutronAscend do not alter the original full graph training semantics (i.e., aggregating embeddings from all neighbor vertices for all vertices in each epoch).

7.7 Comparisons with GPU-based GNN training framework

In this experiment, we compare end-to-end performance and resource utilization between NeutronStar running on the NVIDIA A5000 and NeutronAscend running on the Ascend 910B. We use the `nvidia-smi` command to evaluate GPU utilization and the `msprof` tool to measure AI Core utilization. Figure 15 shows the results. NeutronAscend achieves a 1.47× end-to-end performance

improvement, along with approximately a 11.1% increase in resource utilization. The performance advantage becomes more pronounced on smaller datasets (Citeseer, Cora, Pubmed) and denser graphs such as Reddit, where the average node degree is higher. These datasets are more likely to benefit from the Ascend AI processor's higher local buffer bandwidth and NeutronAscend's data reuse optimizations.

8 Related Work

Frameworks for GNN training. The topic of effectively enabling GNN training has gained increasing attention [1–3, 6, 7, 10, 11, 17, 19, 23, 31, 33, 37, 40, 41, 45–47]. The widespread industrial deployment of GNNs heavily relies on robust framework support. Large-scale graph data and training models must be deployed across various hardware environments (e.g., CPU/GPU/NPU; PCIe/NvLink/CXL) to leverage device resources for accelerating GNN training. The framework provides programming interfaces for users and manages the computation, storage, and networking resources of the underlying hardware. Existing frameworks often focus on GPU-centric hardware environments and introduce a series of key technologies, such as graph sampling [31, 41], graph partitioning [1, 7, 11, 47], feature caching [3, 17, 37, 45], and asynchronous communication [10, 23]. On the other hand, Ascend AI processors, as a new hardware platform, offer computational power comparable to GPUs while consuming less energy [14]. We believe that framework development for Ascend AI processors is a promising research direction. This will not only foster the growth of the Ascend ecosystem but also enhance the portability of GNN training framework across different hardware environments.

Researches on the Ascend AI processor The Huawei group introduces and designs the Ascend AI processors, providing new opportunities for accelerating AI applications [14]. They also launch a series of frameworks centered around MindSpore to support various AI applications, such as MindSpore Transformers, MindSpore NLP, and MindSpore CV. Several benchmarking studies evaluate the performance of Ascend AI processors across different computational tasks within the MindSpore framework [30, 39, 50]. Experimental results show that Ascend AI processors perform comparably to GPUs running the PyTorch framework for compute-intensive AI tasks, such as traditional DNNs. However, their performance is limited in sparse AI tasks, particularly those represented by GNNs, which is the focus of this work. Additionally, several research efforts have explored development on Ascend AI processors. For instance, Ruohan Wu et al. propose an ML-enabled performance model to predict the execution time of different DNN tasks on Ascend AI processors tasks within the MindSpore framework [36]. Salli M. et al. present a custom SPMM operator based on the AI Core, achieving an order-of-magnitude speedup over the AI CPU version [24]. XY-Serve is a Ascend-native LLM inference system that provides hardware-friendly attention and GEMM computation methods [27].

9 Conclusion

We present NeutronAscend, a high-performance GNN training system optimized for the Ascend AI processor. NeutronAscend proposes two key designs around the AI Core, the heart of the Ascend AI processor. First, NeutronAscend employs GNN tensor-data hybrid parallelism to manage task concurrency across AI Cores, combining tensor parallelism for load balancing and data parallelism for improved computational efficiency. Second, NeutronAscend employs computation-aware task scheduling to optimize the allocation of tasks to the heterogeneous computing units within the AI Core, fully leveraging their capabilities while overlapping their execution. Extensive experiments demonstrate that NeutronAscend achieves significant performance improvements compared to the MindSpore Graph Learning framework.

References

[1] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proc. VLDB Endow.* 18, 2 (2024), 173–186.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Sixteenth European Conference on Computer Systems, EuroSys '21, Online Event, United Kingdom*. ACM, 130–144.

[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] Bingqian Du, Jun Liu, Ziyue Luo, Chuan Wu, Qiankun Zhang, and Hai Jin. 2023. Expediting Distributed GNN Training with Feature-only Partition and Optimized Communication Planning. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*. IEEE, 1–10.

[5] 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 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 417–426.

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

[7] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568.

[8] 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.

[9] 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 2020, December 6-12, 2020, virtual*.

[10] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. 2024. FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training. *Proc. VLDB Endow.* 17, 6 (2024), 1473–1486.

[11] 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*. mlsys.org.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12.

[13] 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.

[14] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 789–801.

[15] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*. IEEE, 1–44.

[16] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2023. A Comprehensive Survey on Distributed Training of Graph Neural Networks. *Proc. IEEE* 111, 12 (2023), 1572–1606.

[17] 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*. 401–415.

[18] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 393–405.

[19] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, ATC'19, Renton, WA, USA*. USENIX Association, 443–458.

[20] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: scalable distributed training for large-scale graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'21, St. Louis, Missouri, USA*. ACM, 76.

[21] Eitan Medina and Eran Dagan. 2020. Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor. *IEEE Micro* 40, 2 (2020), 17–24.

[22] Nikolai Merkel, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. 2023. An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training. *CoRR* abs/2308.15602 (2023).

[23] 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.

[24] Moustafa S. 2023. Accelerating Sparse Matrix Matrix Multiplication with the Ascend AI Core. (2023).

[25] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[26] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2022. Distributed Graph Neural Network Training: A Survey. *CoRR* abs/2211.00216 (2022).

[27] Mingcong Song, Xinru Tang, Fengfan Hou, Jing Li, Wei Wei, Yipeng Ma, Runqiu Xiao, Hongjie Si, Dingcheng Jiang, Shouyi Yin, Yang Hu, and Guoping Long. 2024. Tackling the Dynamicity in a Production LLM Serving System with SOTA Optimizations via Hybrid Prefill/Decode/Verify Scheduling on Efficient Meta-kernels. *CoRR* abs/2412.18106 (2024).

[28] 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.

[29] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23. doi:10.1145/3589288

[30] Bangchuan Wang, Chuying Yang, Rui Zhu, Xiao Liu, Mingyao Zhou, and Nenggan Zheng. 2023. Analysis of Performance and Optimization in MindSpore on Ascend NPUs. In *29th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2023, Ocean Flower Island, China, December 17-21, 2023*. IEEE, 1701–1708.

[31] 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).

[32] 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).

[33] 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*. ACM, 1301–1315.

[34] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 149–164.

[35] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 6861–6871.

[36] Ruohan Wu, Mingfan Li, Hanxi Li, Tianxiang Chen, Xinghui Tian, Xiaoxin Xu, Bin Zhou, Junshi Chen, and Hong An. 2022. Machine Learning-enabled Performance Model for DNN Applications and AI Accelerator. In *24th IEEE Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application, HPCC/DSS/SmartCity/DependSys 2022, Hainan, China, December 18-20, 2022*. IEEE, 25–34.

[37] 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).

[38] 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.

[39] Yang Xiao and Zeke Wang. 2024. AIbench: a tool for benchmarking Huawei ascend AI processors. *CCF Trans. High Perform. Comput.* 6, 2 (2024), 115–129.

[40] 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.

[41] 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.

[42] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph Convolutional Networks for Text Classification. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 7370–7377.

[43] 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.

[44] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2023. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *CoRR* abs/2311.13279 (2023).

[45] 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.

[46] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242.

[47] 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 2020, Atlanta, GA, USA, November 11, 2020*. IEEE, 36–44.

[48] 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.

[49] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105.

[50] Zeling Zhu, Bangchuan Wang, Chuying Yang, Rui Zhu, Mingyao Zhou, and Nenggan Zheng. 2023. Performance Evaluation of MindSpore and PyTorch Based on Ascend NPU. In *29th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2023, Ocean Flower Island, China, December 17-21, 2023*. IEEE, 1826–1832.