Algorithm and System Co-design for Efficient and Scalable Graph Learning

Meng Zhang

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2026

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

04/06/2025

Date

Zhang Meng

Meng Zhang

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

04/06/2025

Date

Prof. Tianwei Zhang

Authorship Attribution Statement

This thesis contains material from 2 papers published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as *Meng Zhang, Qinghao Hu, Cheng Wan, Haozhao Wang, Peng Sun, Yonggang Wen, Tianwei Zhang*. ***Sylvie: 3D-adaptive and Universal System for Large-scale Graph Neural Network Training***. *In IEEE International Conference on Data Engineering, ICDE '24, 2024.*

The contributions of the co-authors are as follows:

- I was the lead author. The idea was proposed by me, followed by the implementation of the system framework, execution of the experiments, and writing of the manuscript draft.
- Prof. Tianwei Zhang, Prof. Yonggang Wen, and Dr. Peng Sun were instrumental in guiding the project's early direction, providing critical insights, participating in thoughtful discussions on system design, and contributing substantially to revising the manuscript draft.
- Dr. Qinghao Hu assisted me much in locking research direction, shaping the initial ideas and writing an integrated academic paper.
- Dr. Haozhao Wang and Dr. Cheng Wan made significant revisions to the paper manuscript. Dr. Cheng Wan helped me to improve and correct my initial design ideas. He also assisted in manuscript checking and formula writing.

Chapter 4 is published as *Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, Tianwei Zhang*. ***TorchGT: A Holistic System for Large-scale Graph Transformer Training***. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '24, 2024.*

The contributions of the co-authors are as follows:

- I was the lead author. The idea was proposed by me, followed by the implementation of the system framework, execution of the experiments, and writing of the manuscript draft.
- Jie Sun played a key role in assisting me in conducting ablation experiments and collecting some experimental data.
- Prof. Tianwei Zhang, Prof. Yonggang Wen, and Dr. Peng Sun were instrumental in guiding the project's early direction, providing critical insights, participating in thoughtful discussions on system design, and contributing substantially to revising the manuscript draft.
- Dr. Qinghao Hu provided thoughtful discussions on my system designs, and provided many suggestions on the manuscript writing.

04/06/2025 .... Zhang Meng ....

Date Meng Zhang

Acknowledgements

Looking back on my arrival in Singapore, I could never have imagined that one of the most significant chapters in my life would come to such a fulfilling close here — that my doctoral journey at Nanyang Technological University would conclude today. These past few years as a Ph.D. student have been among the most transformative and enriching periods of my life. What this experience has bestowed upon me extends far beyond mere knowledge; it is an enduring legacy that I will cherish for a lifetime.

I am deeply grateful to the mentors and friends who enriched my journey. Their wisdom and support propelled my research and brightened my time abroad with warmth and strength. Firstly, I owe my deepest gratitude to my advisor, Prof. Tianwei Zhang, whose guidance has profoundly shaped my academic journey. Looking back to the start of my Ph.D., I consider myself truly fortunate to have joined his group of doctoral students. Despite my limited research background at the time, Tianwei's patience, encouragement, and kindness gave me the confidence to take my first steps into the world of research — a support I will always remember with appreciation. Under his mentorship, I was introduced to the field of graph learning optimization. He encouraged me to begin with foundational graph neural network studies and then gradually explore at my own pace. I still remember vividly when I was preparing my first research manuscript, Sylvie, Prof. Zhang printed out my draft and went through it meticulously, marking every section that needed refinement. At that time, I was still inexperienced and sometimes struggled with gaps in my knowledge. Yet Prof. Zhang's unwavering encouragement consistently lifted my spirits and motivated me to keep going. Beyond that, Tianwei generously created opportunities for me to collaborate with industry professionals and supported my participation in several conferences, where I had the privilege to present my work and broaden my perspective. In short, my experience in his group has been transformative, and I have no doubt it will continue to benefit my future in many meaningful ways.

Moreover, I received a lot of guidance and advice from a senior collaborator in our group, Dr. Qinghao Hu. Drawing on his academic expertise and experience, he has offered me selfless and invaluable guidance that has profoundly shaped my research

direction and illuminated the path of my academic journey. Whether it was identifying the next research direction, designing experiments, or writing papers, he consistently offered me thoughtful and practical advice. Whenever I was preparing my research manuscript, he always reviewed the content carefully, providing thoughtful feedback and guiding me in polishing the work. He also actively introduced me to industry-related opportunities, allowing me to learn from many remarkable individuals. Beyond academia, he also shared much practical advice about living in Singapore, which enriched my years as a Ph.D. student here and made my experience all the more vibrant. All in all, I am deeply and sincerely grateful for his tremendous support, both academically and personally, throughout my doctoral journey — support that has greatly contributed to my growth and development.

I would also like to express my heartfelt gratitude to my industrial mentor, Dr. Peng Sun. His extensive practical experience at Shanghai AI Laboratory offered me valuable insights into the real-world challenges inherent in machine learning systems. Notably, his support in securing experimental hardware resources from the company greatly contributed to my research work, for which I am deeply appreciative. The guidance I received from him has had a significant impact on my work, particularly in the development of Sylvie and TorchGT.

In addition, I would like to extend my sincere thanks to Prof. Zeke Wang and Prof. Yonggang Wen for their generous help and guidance. Their insightful feedback and thoughtful suggestions have greatly enhanced the quality of my work.

The research works would not have been possible without the collective efforts of the talented members. I feel fortunate to have collaborated with such remarkable co-authors: Dr. Haozhao Wang, Dr. Zhisheng Ye, Qiaoling Chen, and Cheng Wan. I truly cherish every stimulating discussion and creative brainstorming session we shared along the way.

In our group, I am grateful to work alongside many amiable fellow members: Dr. Wei Gao, Dr. Guanlin Li, Shengwei Li, Shenggui Li, Dr. Kangjie Chen, Dr. Xiaoxuan Lou, Dr. Xingshuo Han, Dr. Gelei Deng, Xiaobei Yan, Hui En Pang, Dr. Meng Hao, Dr. Hanxiao Chen and Jiale Meng. Their enthusiastic assistance and warmth when I first joined the group remain etched in my heart, transforming my apprehension into belonging. They have cultivated a vibrant and collegial atmosphere that encourages open dialogue and fosters genuine collaboration among students. I also became acquainted with many talented and friendly colleagues in S-Lab where I spent my Ph.D. life: Dr. Chenxi Liu, Dr. Haoying Li, Zhexin Liang, Dr. Yuen Fei Wong, Dr. Geong Sen Poh, Wei Tang, Chee Guan Koh, Eunice Yeo and Ng Jin Boon. Furthermore, I

was grateful for the chance to collaborate with exceptional colleagues at Shanghai AI Laboratory: Yingtong Xiong, Guoteng Wang, Xun Chen, and Ting Huang.

Beyond my academic and professional journey, my heartfelt thanks go to my dearest friends who have always stood by my side, whether in Singapore or across the world. It is their unwavering and unconditional support over the years that has sustained and encouraged me, no matter where life took them or me. Every reunion with them has been a solace to my soul, a cherished respite that fuels my spirit amidst the relentless pursuit of my academic endeavors.

Finally, and most importantly, I owe my deepest gratitude to my family, especially my parents and my beloved husband. To my parents, for their boundless support and steadfast encouragement that sustained me during my entire Ph.D. journey despite the geographical distance. With profound tenderness, I dedicate special thanks to my soulmate and life's companion, Yuxin Liu, for his steadfast presence through the long way. Many times when I doubted and research faltered, it was his quiet strength and enduring patience that kept me persevering. At every crossroad, he stood beside me not merely as a supporter, but as the wind beneath my wings, instilling in me the audacity to soar beyond perceived limits. This dissertation bears his fingerprints as surely as mine—every page a testimony to our shared triumph.

With heart full, I thank every soul who made this journey meaningful.

Meng Zhang, Jul 2025

Abstract

In today's digital era, graph data has become a cornerstone of information representation and analysis. By modeling relationships through nodes and edges, graphs provide a powerful framework for understanding complex systems. Beyond revealing hidden patterns and insights, graphs enable predictive modeling and data-driven decision-making. As organizations seek deeper intelligence from interconnected datasets, mastering graph data has become essential, transforming the future of modern analytics.

In modern applications, real-world graphs often contain millions of nodes, leading to prohibitively long sequence lengths. For instance, processing the ogbn-papers100M dataset, a citation graph with over 100 million nodes, requires high dimensional inputs with prohibitive memory demand. Efficient machine learning methods for large-scale graphs are thus critical for model performance and enabling diverse graph learning applications. Besides, deep graph learning models are able to achieve state-of-the-art (SOTA) performance over shallow ones on large graphs. However, existing research on graph learning is confined to small scales due to system limitations and excessive memory requirements. Hence, current graph processing systems primarily suffer from several deficiencies: (1) *low efficiency*, (2) *sacrificing accuracy*, (3) *incompatible for systematic optimizations*. To bridge these gaps, this thesis focuses on efficient systems to advance various graph learning methodologies from the algorithm and system co-design perspective.

This thesis bridges the gap between advanced graph learning algorithms and practical system scalability through a unified framework of system-algorithm co-design. We argue that efficient large-scale graph learning requires moving beyond generic optimizations to exploit intrinsic graph topology-induced sparsity, communication patterns, and heterogeneous computation dynamics. We substantiate this thesis through three progressive system contributions that tackle distinct frontiers of graph learning.

First, we address GNN training scalability by introducing Sylvie, a system that exploits input-level graph properties to optimize training across data, time, and execution dimensions. By pioneering node-aware communication quantization and dynamic scheduling, Sylvie achieves up to 17.2× speedup while strictly preserving theoretical convergence guarantees for deep GNNs.

Second, we extend these principles to long-range dependency modeling with TorchGT, the first system capable of scaling Graph Transformers to billion-edge graphs. TorchGT overcomes the quadratic complexity of attention mechanisms by introducing topology-induced sparse attention and communication-light parallelism. This co-design approach enables the processing of sequence lengths exceeding one million nodes, achieving a 62.7× speedup over existing baselines.

Third, we tackle the multimodal convergence of texts and graphs with UniTG, a unified system for graph learning with Language Models (LM4Graph). Addressing the computational mismatch between LMs and graph operations, UniTG utilizes "computation bubbles" and graph affinity-based scheduling to seamlessly integrate textual and graph structural learning phases, reducing training time by 17.3×.

Together, these works demonstrate that aligning system architecture with algorithmic properties allows for massive scalability without compromising model quality. This thesis provides a comprehensive methodology for building high-performance, practical graph learning infrastructure, paving the way for the deployment of sophisticated graph foundation models in real-world applications.

Contents

Acknowledgements **v**

Abstract **viii**

List of Publications **xiii**

List of Figures **xiv**

List of Tables **xvii**

1 Introduction **1**
1.1 Background and Challenges . . . 1
1.2 Motivation and Contribution . . . 4
1.3 Outline . . . 6

2 Preliminary and Literature Review **7**
2.1 Graph Learning Basics . . . 7
2.1.1 Graph Data Formulation . . . 7
2.1.2 Graph Learning Tasks . . . 9
2.2 Graph Learning Algorithms . . . 10
2.2.1 Graph Neural Network . . . 10
2.2.2 Graph Transformer . . . 11
2.2.3 LM4Graph . . . 12
2.3 Graph Training System . . . 13
2.3.1 Distributed GNN Training . . . 13
2.3.2 Large-Scale Graph Transformer Training . . . 14
2.3.3 LM4Graph Training . . . 15
2.4 Literature Review . . . 16
2.4.1 Graph Neural Network Optimization . . . 16
2.4.2 Graph Transformer Optimization . . . 18
2.4.3 LM4Grah Optimization . . . 19

3 Sylvie: Efficient Graph Neural Network Training System **21**
3.1 Introduction . . . 21
3.2 Background and Motivation . . . 25
3.2.1 Communication Bottleneck . . . 25

3.2.2 Quantization for GNNs . . . 27
3.2.3 Pipeline of Distributed GNN Training . . . 29
3.3 System Overview . . . 30
3.4 Offline Stage of Sylvie . . . 32
3.4.1 Graph Extractor . . . 34
3.5 Online Stage of Sylvie . . . 35
3.5.1 Quant Orchestrator . . . 35
3.5.2 Pipeline Adaptor . . . 38
3.5.3 Coordinator . . . 40
3.6 Evaluation . . . 40
3.6.1 End-to-end Experiments . . . 42
3.6.2 Ablation Studies . . . 45
3.6.3 More Evaluation . . . 46
3.6.4 Scalability Analysis on More Servers . . . 48
3.7 Summary . . . 49

4 TorchGT: Large-scale Graph Transformer Training System 50
4.1 Introduction . . . 50
4.2 Challenges and Motivation . . . 54
4.2.1 Long Sequence for Graph Transformers . . . 54
4.2.2 Issues and Opportunities . . . 54
4.3 System Design . . . 56
4.3.1 System Overview . . . 57
4.3.2 Dual-interleaved Attention . . . 58
4.3.3 Cluster-aware Graph Parallelism . . . 60
4.3.4 Elastic Computation Reformation . . . 63
4.4 Evaluation . . . 66
4.4.1 End-to-end Training Throughput . . . 68
4.4.2 Model Convergence . . . 69
4.4.3 System Scalability . . . 70
4.4.4 Micro-benchmarks . . . 72
4.4.5 Pre-processing Cost . . . 74
4.5 Summary . . . 74

5 UniTG: Unified System for Textual Graph Learning in One Step 76
5.1 Introduction . . . 76
5.2 Motivation and Opportunities . . . 80
5.2.1 Large Models for LM4Graph . . . 80
5.2.2 Opportunities for Efficient Graph Learning . . . 81
5.3 System Design . . . 84
5.3.1 UniTG Overview . . . 84
5.3.2 Affinity-aware Flow Parallelism . . . 85
5.3.3 Dual-modality Collaborative Learning . . . 89
5.3.4 Streamlined Pipeline Schedule . . . 91
5.4 Evaluation . . . 93

5.4.1 End-to-end Performance 94
5.4.2 Effect of Dependency-aware Flow Parallelism 96
5.4.3 Effect of Dual-modality Collaborative Learning 97
5.4.4 Effect of Streamlined Pipeline Schedule 99
5.5 Conclusion 100

6 Conclusion and Future Work 101
6.1 Conclusion 101
6.2 Limitations 103
6.3 Future Work 104

Bibliography 107

List of Publications

[1] Meng Zhang*, Jie Sun*, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, Tianwei Zhang. **TorchGT: A Holistic System for Large-scale Graph Transformer Training**. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* , SC '24, 2024.

[2] Meng Zhang, Qinghao Hu, Cheng Wan, Haozhao Wang, Peng Sun, Yonggang Wen, Tianwei Zhang. **Sylvie: 3D-adaptive and Universal System for Large-scale Graph Neural Network Training**. In *IEEE International Conference on Data Engineering*, ICDE '24, 2024.

[3] Qinghao Hu*, Zhisheng Ye*, Zerui Wang*, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, Tianwei Zhang. **Characterization of Large Language Model Development in the Datacenter**. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, 2024.

[4] Haozhao Wang, Yabo Jia, Meng Zhang, Qinghao Hu, Hao Ren, Peng Sun, Yonggang Wen, Tianwei Zhang. **Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters**. In *The Web Conference*, WWW '24, 2024.

[5] Qinghao Hu*, Meng Zhang*, Peng Sun, Yonggang Wen, and Tianwei Zhang. **Lucid: A Non-Intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs**. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, 2023.

[6] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. **Hydro: Surrogate-Based Hyperparameter Tuning Service in Datacenters**. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, 2023.

* Equal Contribution.

List of Figures

1.1 The development in graph data science. 2

2.1 Examples of real-life graph data. 8
2.2 Illustration of large-scale graphs. 8
2.3 (a) The architecture of LM4Graph methods. (b) LM4Graph methods outperform classical GNNs on TAG-related tasks by a large margin. . . 13
2.4 Example of vanilla distributed GNN training. For the partition on GPU-1, node 4 requires extra features of node 7 from GPU-0 and node 1 from GPU-2 to update its embedding in each layer. 14
2.5 The test accuracy of graph transformers when trained with different sequence lengths S. 16

3.1 Training time per epoch in vanilla distributed training with DGL on a single server (8 GPUs). 26
3.2 Training time per epoch and its breakdown when using different quantization bit-widths to train GraphSAGE on Yelp over 8 GPUs. 28
3.3 Overview of Sylvie architecture and workflow. 31
3.4 The training process with Sylvie. Orange circles and rectangles represent nodes allocated to GPU-1 and their corresponding messages. The others in gray represent nodes/messages on other GPUs. 32
3.5 Training loss with respect to different quantization bit-widths on GAT (Yelp dataset). 36
3.6 The process of how *Quant Orchestrator* jointly orchestrates the quantization from time- and data-dimension. 38
3.7 Illustration of how *Pipeline Adaptor* adjusts execution mode between synchronous and asynchronous training. 39
3.8 Training throughput of different methods (normalized to that of DGL, shown in the dashed line) when training three representative models on four datasets on two 3090 servers. Sylvie outperforms DGL by up to 16.0×. 43
3.9 The convergence curve comparisons of Sylvie and baselines on different models and datasets over single-server. 44
3.10 Sensitivity experiments of comparing range δ on Reddit (N=8, single server). 47
3.11 Wall-clock time of DGL and Sylvie with overhead. 47
3.12 Ratios of different components in epoch time when training GraphSAGE with Sylvie on Reddit over single server and two servers. Both quantization and coordination take up negligible overhead. 48

3.13 Normalized training throughput on multiple 3090 servers for GraphSAGE on Amazon. . . . 49

4.1 Training iteration time breakdown when training Graphormer on ogbn-products in different sequence lengths on two types of GPUs: RTX 3090 and A100. . . . 55
4.2 Overview of TorchGT architecture and workflow. . . . 58
4.3 Detailed training process on one worker with TorchGT, which includes three key components. . . . 62
4.4 Three attention layouts after *Dual-interleaved Attention*, *Cluster-aware Graph Parallelism* and *Elastic Computation Reformation* respectively. (c) is obtained by compacting elements to adjacent neighbors inside clusters. . . . 64
4.5 (a) Profiled hardware statistics of GPU when computing in different sub-block size d_b. The ideal d_b considers both load balance and cache hit rate. (b) Computation throughput of the indexing kernel in different d_b normalized on that of d_b=2. . . . 66
4.6 The convergence curve comparisons of TorchGT and GP-Flash on different models and datasets. . . . 69
4.7 Scalability results of TorchGT in training GPH_{Slim} on ogbn-products on many A100 servers. (a) With fixed sequence length, throughput reduces almost linearly. (b) With fixed computational load per GPU, throughput remains well. . . . 71
4.8 Scalability experiments of training GPH_{Slim} on ogbgn-products. (a) The supported maximum sequence length w.r.t. GPU number. (b) Training throughput w.r.t. sequence length. In both cases TorchGT shows greater scalability than others. . . . 71
4.9 Convergence comparisons of different attentions: our interleaved attention, FlashAttention and sparse attention. . . . 72
4.10 Convergence curves of different attentions: our interleaved attention, full and sparse attention on small graphs. . . . 73
4.11 Computation time of attention modules when training Graphormer on ogbn-products with different (a) sequence lengths and (b) model hidden dimensions when S=256K. . . . 73

5.1 The test score of GNNs when trained with different numbers of layers. Deep layers show better performance. . . . 80
5.2 The profiled time and memory consumption of a LM4Graph method on different GPUs. . . . 81
5.3 Time breakdown of each phase when training different methods on 3 datasets on GPU-A (details in §5.4). . . . 83
5.4 Overview of UniTG architecture and workflow. . . . 85
5.5 Design of Affinity-aware Flow Parallelism. (a) Original graph topology, (b) affinity-aware topology and (c) Flow Parallelism for a GNN model with 4 pipeline stages and 4 graph microbatches. . . . 86

5.6 The test accuracy of GNNs in Flow Parallelism when trained with different numbers of microbatches M. Large M results in degraded model accuracy. 88
5.7 Illustration of the proposed Centrality-guided Lazy Tuner, and how it interacts with the GNN phase. Each grey bar represents a sequence of model layers. Both phases are trained in the collaborative learning mechanism. 89
5.8 Illustration of Bubble Interleaver with 4 pipeline stages and 4 micro-batches. Right-side is the memory consumption of each GPU. 91
5.9 Detailed time breakdown of LM and GNN phases for all methods on two models. UniTG far exceeds baselines in training time within one integral procedure. 95
5.10 Ablation study of Affinity-aware FP on ogbn-arxiv on one type-A GPU server. (a) Effect of each technique. (b) The supported maximum GNN size w.r.t. GPU count. 97
5.11 Effect of Lazy Tuner on DeBERTa-base. (a) Total time effect on LM fine-tuning. (b) Loss curve comparisons with some highlighted exit points. 98
5.12 The convergence curve comparisons of UniTG and TAPE on different models and datasets on one type-B server. 98
5.13 Visualization of GPU utilization via DCGM on (a) RevGCN and (b) RevGAT. Several iterations of the first pipeline stage are presented. The execution periods of the interleaved LM phase are highlighted by red arrows. 100
5.14 GPU utilization on the second and third stage when training RevGCN in FP. 100

List of Tables

2.1 Comparison of prior works on GNN quantization. 17
2.2 Comparison of existing graph transformer methods and TorchGT. . . . 19
2.3 Comparison of existing LM4Graph methods and UniTG. 19

3.1 Test accuracy of training GraphSAGE on the Ogbn-products dataset with different sample sizes. 22
3.2 Data volume of communicated messages (embeddings & embedding gradients) and weight gradients of three models on the Ogbn-products dataset over 4 GPUs. 4-GCN-256 means a 4-layer GCN with the hidden size of 256. 26
3.3 The test accuracy (%) of three models trained with different quantization bit-widths. 4-GCN denotes a GCN with 4 layers and the other models are similar. 29
3.4 Detailed information of datasets used in evaluation. 41
3.5 Model architecture and detailed hyperparameters. ***Arch.***: Number of layers × Number of hidden neurons in each layer. ***HP.***: (Epoch, Dropout) 42
3.6 Detailed comparison of training throughput and test accuracy between SYLVIE and other baselines when training on two 3090 servers, where the best performance is highlighted in bold. Dash line '-' means the method does not converge. SYLVIE always outperforms others in throughput on all the models and datasets while still achieving high accuracy. . 43
3.7 Training epoch time comparison between SYLVIE and other methods on GraphSAGE on A100 servers with NVLink. 44
3.8 Throughput when training on a single 3090 server. 45
3.9 Training GraphSAGE on Yelp with different b values and fixed execution on single A100 server. 46
3.10 Training GraphSAGE on Yelp with different execution modes and fixed b values on single A100 server. 46
3.11 Epoch communication volume and time breakdown of training GraphSAGE over two servers. 47
3.12 Epoch time of GraphSAGE on Ogbn-products under different A100 server settings. 49

4.1 Graph transformers outperform classical GNNs on graph-level and node-level (Flickr) tasks. 51
4.2 Backward (BW) time of topology-pattern & dense counterpart when training Graphormer on ogbn-products. 56

4.3 Detailed information of datasets in evaluation. 67
4.4 Detailed information of graph transformer models. 67
4.5 Detailed comparison of training speed and test accuracy of methods when training on one 3090 GPU server. OOM means the method runs out of memory. TorchGT always outperforms GP-FLASH in throughput and accuracy on all the models and datasets. GP-RAW with full attention runs out of memory in all cases. 68
4.6 Training time per epoch of trianing GPH_{Slim} on one A100 server. TorchGT can still improve training efficiency compared with GP-FLASH. 68
4.7 Training throughput and test accuracy of methods. The accuracy of GP-FLASH decreases because of BF16. 70
4.8 Training time per epoch and test accuracy on ogbn-arxiv dataset regarding different transfer threshold β_{thre}. 74
5.1 LM4Graph methods outperform classical GNNs on TAG-realted tasks by a large margin. 77
5.2 Epoch time of training GNN in affinity-aware way and the original counterpart on ogbn-products. 82
5.3 Detailed information of evaluated datasets, where ogbn-products* stands for a subset of the original dataset. 94
5.4 Hyperparameters for GNNs. For cells with two values, the first value corresponds to the first dataset and the rest are similar. Otherwise datasets share the same value. 94
5.5 Detailed comparison of total time and test accuracy of methods when training on one GPU-A server. OOM means the method runs out of memory. UniTG always outperforms baselines in makespan and achieves comparable accuracy on all models and datasets. 95
5.6 GPU energy consumption comparisons of UniTG and baselines (units: J). 96

Chapter 1

Introduction

1.1 Background and Challenges

Graph-structured data has long been prevalent and indispensable in many real-life applications such as social network construction and molecular analysis. Graphs, consisting of nodes and edges that denote relationships between them, offer a versatile framework for modeling complex systems across various domains. From social networks and recommendation engines to biological networks and supply chains, the prevalence of graph data is undeniable. This surge is fueled by the exponential growth of interconnected data sources and the need for sophisticated data analysis techniques. Graph data not only enables us to uncover hidden patterns and insights but also facilitates predictive modeling and decision-making.

As organizations strive to extract meaningful intelligence from interconnected datasets, the importance of understanding and harnessing graph data continues to escalate, reshaping the landscape of modern data analytics. As shown in Figure 1.1, at the very beginning, we have knowledge graphs, which allow people to prioritize relationships that might be more important. Graph feature engineering incorporates some machine learning to add graph features into the applications. Graph embeddings are learned representations of the graph, which is a required step for deep learning. Then we have graph neural networks popularized in 2016, performing native learning and training multi-layer neural networks using gradient descent. From 2019, transformer-based models have emerged, and they show promising power in capturing the inter-dependencies among nodes, attracting surging attention in recent years.

Real-world graphs can easily involve millions of nodes [1, 2], making the memory demands enormously large. For example, in the current graph transformers' operation way, processing the citation graph dataset ogbn-papers100M from Open Graph Benchmark [1] (including more than 100 million nodes) requires high dimensional inputs with prohibitive memory footprints. Therefore, training machine learning models

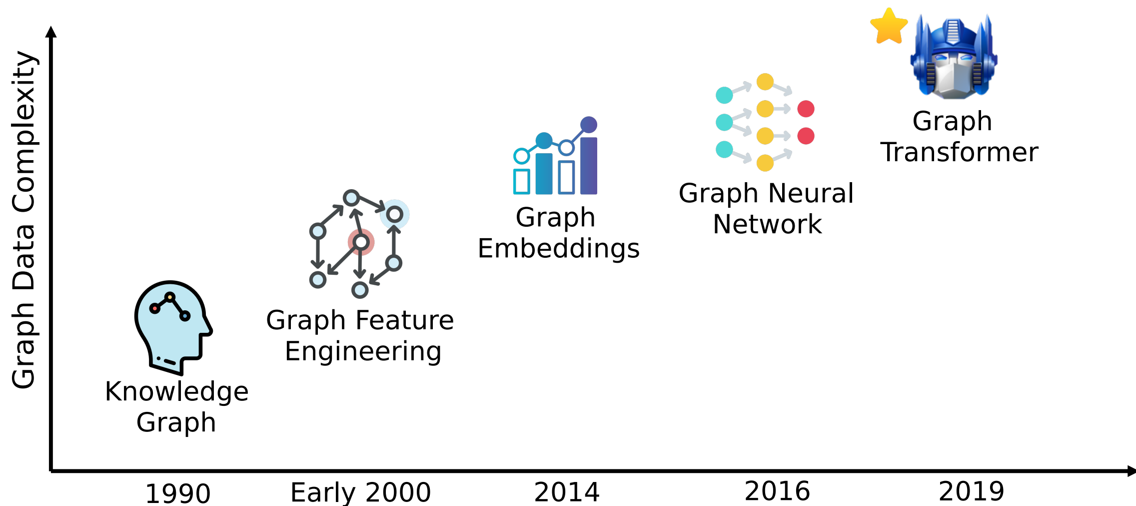


FIGURE 1.1: The development in graph data science.

targeted at large-scale graphs is crucial for model quality and the development of versatile graph learning application scenarios. However, we find most existing research works [3–6] are only limited to small graphs due to a lack of compatible systems, extensive memory demand, model quality degradation and so on. More specifically, there are several deficiencies in current works.

First, the model quality may be compromised. For GNNs, though sampling-based methods [3, 7–9] can reduce the memory footprint, they forfeit crucial neighborhood information, ultimately resulting in obvious model accuracy degradation ($> 2\%$) compared with full-graph training. This line of work typically assumes that a small sampled neighborhood can sufficiently approximate the informative context of each node during training. Consequently, sampling introduces an explicit trade-off: it improves scalability (memory/compute) but may increase estimation bias/variance and slow convergence, which can manifest as degraded accuracy compared with full-graph training. While these methods successfully address memory constraints through neighborhood sampling strategies, they inherently sacrifice critical structural information that is essential for optimal model performance. The accuracy drop represents more than just a numerical difference. It often corresponds to critical failures in real-world applications. In recommendation systems, this might mean missing important long-tail items; in fraud detection, it could translate to overlooking sophisticated attack patterns that only become visible in complete neighborhood contexts. The degradation tends to worsen with more complex multi-relational graphs, growing importance of long-range dependencies and deeper GNN architectures.

Second, extensive memory demand is required. The quadratic memory complexity inherent in standard graph transformer architectures presents a fundamental scalability barrier that severely limits their practical application. For example, graph

transformers with standard attention calculate attention for all node pairs, resulting in the computation and memory complexity of $O(N^2)$, quadratic on the number of nodes (N) in a graph [10–13]. At the core of this challenge lies the all-pairs attention mechanism, which requires maintaining an N×N attention matrix where N represents the number of nodes - a requirement that quickly becomes prohibitive as graph size increases. For instance, a moderately-sized graph with just 1 million nodes would demand terabytes of memory for the attention matrix alone, far exceeding the capacity of modern GPUs. This explosive memory scaling creates cascading system challenges in the entire training pipeline. Hardware limitations force drastic reductions in batch sizes, trigger frequent out-of-memory errors, and necessitate inefficient memory swapping operations that degrade performance by orders of magnitude. Practitioners face an impossible obstacle: either accept severely truncated graph sizes, invest in prohibitively expensive hardware infrastructure, or compromise model quality through approximations - none of which are satisfactory solutions. The problem intensifies when considering real-world requirements like high-dimensional node features, edge attributes, and deeper architectures, all of which further strain memory resources. Compared to alternative approaches like traditional GNNs with their $O(E)$ complexity or CNNs with localized receptive fields, graph transformers' memory hunger appears particularly acute.

Third, there is a lack of systematic optimizations. The lack of systematic optimizations for graph-based attention mechanisms creates fundamental performance barriers that existing solutions fail to adequately address. At the hardware level, the sparse and irregular nature of graph-structured computations leads to catastrophic memory access patterns that existing architectures handle poorly. Unlike the regular, predictable memory accesses of dense matrix operations or even traditional transformer attention, graph attention must navigate highly variable neighborhood structures, resulting in memory access latencies that can reach 33.2× slower than their dense counterparts. This performance gap stems from multiple compounding factors: the inherent data-dependent nature of graph traversals, poor cache utilization due to irregular access patterns, and inefficient memory bandwidth usage when fetching small, scattered fragments of node data. The challenge intensifies when considering parallelization strategies - while large language models have benefited from sophisticated parallelism techniques like tensor parallelism, pipeline parallelism, and various attention optimizations, these approaches translate poorly to graph learning scenarios. Moreover, although there have been many parallelism methods for large language models (LLMs) [14–17], they cannot be directly transplanted on graph learning methods due to the extra graph encodings and neglect of structure properties. Existing parallelism methods often make assumptions about data regularity and uniform computation patterns that graphs routinely violate, leading to severe load imbalance and underutilized compute resources. Moreover, the interplay between graph structure

and attention mechanisms creates novel bottlenecks not seen in other domains, such as the tension between global attention patterns and local graph neighborhoods, or the challenge of efficiently aggregating structural information across devices. These limitations become particularly acute at scale, where the combination of sparse computation, irregular memory access, and inefficient parallelism can render theoretically efficient algorithms practically unusable on real-world graph problems.

1.2 Motivation and Contribution

The intersection of graph learning and systems has unlocked transformative opportunities, heralding a golden era for interdisciplinary research in machine learning and systems. This thesis establishes a unified framework for system-algorithm co-design to address the scalability bottlenecks of modern graph learning. We introduce a progressive suite of optimizations that evolve alongside model complexity: GNNs, Graph Transformers and LM-based graph learning. Together, these contributions establish a comprehensive methodology for developing high-performance, practical systems capable of deploying advanced graph learning algorithms in real-world, resource-constrained environments.

To clarify the scope of our contributions, all three proposed systems are designed to address the above challenges: *model quality preservation*, *memory/scalability constraints*, and *the lack of systematic optimizations*. However, since the three systems target different graph learning paradigms and application scenarios, these challenges manifest differently in each system; therefore, they require corresponding, differentiated solutions with explicit design trade-offs.

Sylvie [18]: **Efficient Graph Neural Network Training System.** It differs from all existing works in that it is model-agnostic and exploits input-level information on full-graph training. It optimizes arbitrary GNN training from **data, time, and execution** dimensions. SYLVIE leverages previously unexploited input-level graph properties to optimize training across three synergistic dimensions: (1) **data-level** optimizations that intelligently exploit inherent graph sparsity patterns and community structures, (2) **time-domain** optimizations through dynamic computation scheduling that adapts to training dynamics, and (3) **execution-level** communication and computation overlapping. The system pioneers three groundbreaking techniques: a universal model-agnostic architecture that maintains theoretical guarantees across diverse GNN variants (from classic GCN to advanced GraphSAGE and deep GNNs). Supported by both theoretical proofs and extensive experiments, SYLVIE can improve training on versatile GNN models, even on deeper GNNs. SYLVIE is the *first* to explore the input-level properties on expediting and guaranteeing large-scale full-graph

training. Besides, we pioneer in identifying the unique opportunities of quantizing communicated messages in GNNs. We coordinate a set of system optimizations to substantially facilitate the training efficiency (up to 17.2×) while preserving the model quality. Notably, SYLVIE introduces the novel insight that message communication in GNNs presents unique quantization opportunities distinct from other neural networks, enabling additional efficiency gains through its coordinated set of system optimizations that collectively push the boundaries of practical GNN training scalability.

TorchGT [19]: **Large-scale Graph Transformer Training System.** We design TorchGT, the first distributed training system that scales graph transformer model to large graphs with billions of edges. As the first system of its kind, TorchGT establishes four fundamental design principles: (1) Scalability: supporting near-linear performance scaling across distributed compute nodes; (2) Efficiency: maximizing hardware utilization through novel optimization techniques; (3) Convergence-preservation: ensuring training stability and model quality comparable to full-batch training; and (4) Task-agnostic: maintaining flexibility across diverse graph learning tasks. TorchGT is the *first* graph transformer system that facilitates efficient, scalable, and accurate training on large-scale graphs as well as universal graph learning tasks. We are the *first* to identify the major challenges that hinder existing graph transformers from scaling to large graphs and explore the graph-specific optimization opportunities which are neglected previously. We propose three key techniques to meet all design goals from algorithm and system co-design perspectives. Extensive experiments demonstrate TorchGT's remarkable performance, achieving up to 62.7 × speedup compared to existing implementations while supporting graph sequence lengths exceeding one million nodes. The system shows near-perfect linear scaling across GPU clusters, enabling training on graph sizes previously considered impractical. Importantly, these efficiency gains come without sacrificing model quality - TorchGT maintains competitive accuracy across standard graph benchmarks while opening new possibilities for large-scale graph learning applications. Our work fundamentally advances the field by bridging the gap between theoretical graph transformer models and their practical realization at scale.

UniTG: Unified System for Textual Graph Learning in One Step. While existing LM-based graph learning approaches have demonstrated promising algorithmic breakthroughs, their practical deployment remains severely constrained by four fundamental limitations: (1) prohibitive computational costs stemming from inefficient joint processing of textual and graph data, (2) rigid architectural designs that force separate handling of different modalities, (3) scalability barriers preventing the use of state-of-the-art large language models, and (4) inadequate incorporation of essential graph structural properties. To overcome these challenges, we introduce UniTG, the first truly unified system for end-to-end LM-graph learning that seamlessly integrates

both modalities in a single computational framework. Inspired by the unique features of LM4Graph, UniTG exploits the bubble resources to improve hardware utilization and holistic system efficiency. We summarize the current challenges and identify the graph-specific opportunities for both LM and GNN phases' optimizations, including guiding fine-tuning with node centrality and computing nodes based on graph affinity. It substantially improves the total training time by up to 17.3 ×.

1.3 Outline

The research problem studied in this thesis can be categorized according to the graph learning method: GNN training optimization, graph transformer optimization, and LM4Graph optimization. We focus on these three graph learning methods and improve their training performance. This thesis is organized as follows:

- Chapter 2 presents the preliminary of various graph learning algorithms, distributed training, and relevant studies to this thesis.
- Chapter 3 presents a full-graph GNN training system that not only improves the training throughput substantially but also maintains the model quality for universal GNNs by harnessing the inherent information embedded in the graph data and model structure.
- Chapter 4 proposes the first distributed training system that scales graph transformer model to large graphs with billions of edges, whose designs abide four goals: scalable, efficient, convergence-maintained, and task-agnostic.
- Chapter 5 designs a compute-efficient and accuracy-maintained system from both system and algorithm perspectives to support portable LM-based graph learning in one step.
- Chapter 6 summarizes the thesis and discusses future research directions.

Chapter 2

Preliminary and Literature Review

2.1 Graph Learning Basics

2.1.1 Graph Data Formulation

Graph data are indispensable for many real-life applications and appear ubiquitously in modern society. They naturally capture complex relationships and interactions between entities, making them a powerful tool for modeling structured data. For example, as shown in Figure 2.1, social networks can be represented as graphs where nodes correspond to users and edges capture friendships or interactions [20]. In chemistry, molecules are modeled as graphs with atoms as nodes and chemical bonds as edges [1]. Similarly, product recommendation systems often construct graphs to represent users and items, linking them based on preferences or co-purchasing behaviors [2]. Even large-scale infrastructures, such as power grids or transportation networks, can be effectively modeled as graphs. Formulating data in this way enables the development of advanced machine learning methods, like Graph Neural Networks, which can directly leverage the graph structure to capture rich and meaningful patterns.

Let $\mathcal{G} = (V, E)$ be a graph defined by a vertex set V and an edge set E, where each edge connects pairs of vertices. In the context of graph representation learning, we consider a graph $\mathcal{G} = (V, E)$ where V denotes the set of vertices and E represents the set of edges connecting these vertices. For any given vertex $v \in V$, its neighborhood structure $N(v)$ can be formally defined in two distinct ways: (1) through the explicit connectivity patterns established by the edge set E, where $N(v) = u \in V|(v, u) \in E$, or (2) via sophisticated sampling strategies employed in modern graph neural network implementations, which may include random walks, importance sampling, or other neighborhood aggregation techniques. Each vertex $v \in V$ is associated with a feature representation $h_v \in \mathbb{R}^d$, where d denotes the dimensionality of the vertex feature space. Similarly, every edge $e \in E$ is characterized by its own feature vector $g_e \in \mathbb{R}^{d'}$, with d'

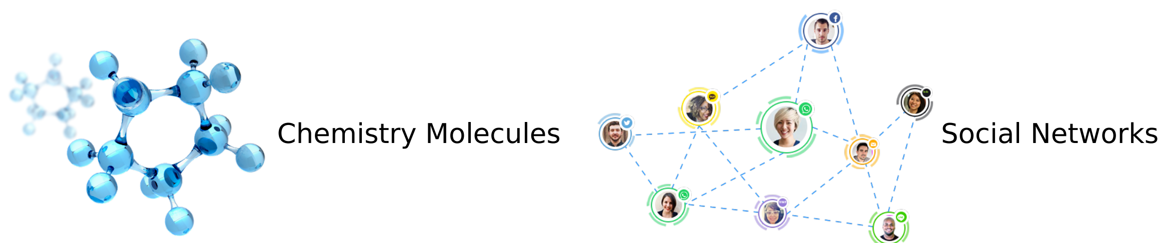
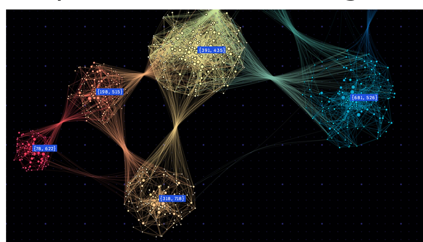


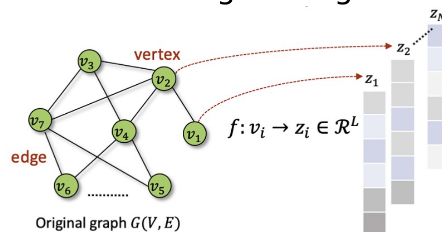
FIGURE 2.1: Examples of real-life graph data.

1. Graph structure scaling



Ogbn-paper100M has 111M nodes and 1.6B edges

2. Embedding scaling



Reddit graph has 602 embedding dimension

FIGURE 2.2: Illustration of large-scale graphs.

representing the edge feature dimensionality. These feature vectors typically encode either fundamental scalar attributes or more complex, high-dimensional descriptors that capture the rich information contained in the graph structure. At the graph level, we may define a global feature vector $y \in \mathbb{R}^{d''}$ that encapsulates macroscopic properties and holistic characteristics of the entire network.

Real-world graphs can easily involve millions of nodes [1, 2], making the sequence length enormously large for graph transformers and LM4Graph algorithms. For example, Figure 2.2 shows in the current graph learning operation way, processing the citation graph dataset ogbn-papers100M from Open Graph Benchmark [1] (including more than 100 million nodes) requires high dimensional inputs with prohibited sequences. The dimension of graph embeddings can also be large. Reddit graph dataset has an embedding dimension of 602, resulting in substantial input and activation memory demand. Moreover, learning on large graphs is crucial for model quality and the development of versatile graph application scenarios. However, we find most modern GNN [3, 4] or graph transformer research works [5, 6, 21, 22] are only limited to small graphs due to a lack of compatible systems tailored for the graph model training with large graphs or long sequences.

2.1.2 Graph Learning Tasks

In graph learning tasks, according to the graph size and properties, predictions are typically organized into three main categories: node-level prediction, graph-level prediction, and link prediction. These tasks illustrate the flexibility and broad applicability of GNNs and other graph-based models.

Node (vertex)-level classification focuses on predicting labels or properties of individual vertices within a single graph. Each vertex is an instance with a label and one can treat all the nodes as non-independent and identically distributed (i.i.d.) generated due to the inter-dependence. Therefore, node-level classification tasks usually involve a single large graph such that scalability is paramount, especially if we are to consider arbitrary relationships across all nodes. For instance, in a citation network, this could involve classifying the subject area of each paper (vertex) based on its textual content and its connections to other papers [1, 23]. The goal is to learn vertex embeddings that combine structural and attribute information to enable accurate per-vertex downstream predictions [4, 24, 25].

Graph-level classification aims to assign labels or predict properties for the entire graphs. Distinct from the node-level counterpart, for graph-level tasks, each i.i.d. instance is a small graph itself and the connected nodes within each graph are computationally inexpensive. This task often arises when each graph in the dataset represents a distinct object, such as a chemical compound or protein in ogbg-molpcba dataset [1] and MalNet [26]. The objective is to predict whether a molecule is active against a disease target or whether a protein has a certain function. This graph-level scenario has been explored in the context of graph structure learning [27] and all-pair message passing design, e.g., graph transformers [5, 6]. Models must aggregate information from all vertices and edges to learn meaningful global representations for graph-level inference.

Link prediction, sometimes called edge prediction, focuses on predicting the existence or likelihood of an edge between two vertices. This task is common in recommendation systems and knowledge graphs, where the goal is to infer missing connections (e.g., suggesting new friendships in a social network) or discover hidden relationships between entities [28]. For instance, one work [29] designs a link prediction task to further refine the understanding of relationships between entities and capture graph knowledge in a self-supervised manner.

Together, these tasks highlight the power of graph-based learning: models can reason about local vertex properties, global graph characteristics, and pairwise relationships between vertices, making them suitable for a wide range of real-world applications. In this thesis, we mainly focus on node-level and graph-level classification tasks.

2.2 Graph Learning Algorithms

2.2.1 Graph Neural Network

GNNs are machine learning algorithms that learn from the graph connectivity and model the relationship between nodes. To update nodes, a GNN model first aggregates the feature vectors from the nodes' neighbors and then combines them together, which is called message passing [30]. In general, the iterative learning process contains two important steps in each layer: feature aggregation and update. Intuitively, consider a graph $\mathcal{G} = (V, E)$ with nodes $V = \{v_1, \cdots, v_{|V|}\}$, edges $E = \{e_1, \cdots, e_{|E|}\}$ and node feature matrix $\boldsymbol{X} \in \mathbb{R}^{|V| \times d}$. For an arbitrary layer $l \in [1, L]$, the aggregation and update steps can be expressed as:

$$\boldsymbol{z}_v^{(l)} = \rho^{(l)}\left(\left\{\boldsymbol{h}_u^{(l-1)} \mid u \in \mathcal{N}(v)\right\}\right) \quad (2.1)$$

$$\boldsymbol{h}_v^{(l)} = \phi^{(l)}\left(\boldsymbol{z}_v^{(l)}, \boldsymbol{h}_v^{(l-1)}\right) \quad (2.2)$$

where $\mathcal{N}(v)$ means the neighboring nodes of node v. The aggregation function $\rho^{(l)}$ takes the embeddings of neighboring nodes $\boldsymbol{h}_u^{(l-1)}$ to get an intermediate aggregated result $\boldsymbol{z}_v^{(l)}$, which then serves as the input to update function $\phi^{(l)}$ together with the feature embedding $\boldsymbol{h}_v^{(l-1)}$ of node v itself to obtain the learned embedding $\boldsymbol{h}_v^{(l)} \in \boldsymbol{H}^{(l)}$ at the l-th layer, where $\boldsymbol{H}^{(l)}$ is the matrix consisting of all nodes' embeddings at the l-th layer. Different GNNs vary in their aggregation and update functions.

Deep GNNs. Despite the enormous success of classical shallow GNNs, deeper models are capable of extracting information from higher-order neighbors by solving the over-smoothing problem. JKNet [31] uses dense skip connections to combine the output of each layer to preserve the locality of the node representations. SGC [32] attempts to capture higher-order information in the graph by applying the power of the graph convolution matrix in a single neural network layer. In this work, we highlight our system not only outperforms on shallow and classical GNN models, but can also be easily extended to other deeper and more advanced GNN models. Most of the existing systems, such as NeutronStar [33], all optimize with 2-layer GNN models, while ignoring the mainstream of adopting deeper models to extract information from high-order neighbors.

In our thesis, we classify GNN architectures into three types: shallow, deep, and special GNNs. For each kind, we list one example of the update rule as below: GCN [4], DAGNN [34], and GAT [24].

- Graph Convolution Network (GCN):

$$\boldsymbol{z}_v^{(l)} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{d_v d_u}} \mathbf{W}^l \boldsymbol{h}_u^{(l-1)}, \quad \boldsymbol{h}_v^{(l)} = \sigma\left(\boldsymbol{z}_v^{(l)}\right)$$

where d_v refers to the degree of node v, σ is an activation function, and $\mathbf{W}^l$ is the weight matrix at layer l.

- Deep Adaptive Graph Neural Network (DAGNN):

$$\boldsymbol{Z} = MLP(\boldsymbol{X})$$
$$\boldsymbol{H}^{(L)} = stack(\boldsymbol{Z}, \widehat{\boldsymbol{A}}^1\boldsymbol{Z}, \widehat{\boldsymbol{A}}^2\boldsymbol{Z}, ..., \widehat{\boldsymbol{A}}^L\boldsymbol{Z})$$

where $\widehat{\boldsymbol{A}} = \widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}$, $\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$. $\widetilde{\boldsymbol{D}}_{v,v} = \sum_u \widetilde{\boldsymbol{A}}_{v,u}$ is the diagonal node degree matrix, $\boldsymbol{I}$ is the identity matrix.

- Graph Attention Network (GAT):

$$\boldsymbol{z}_v^{(l)} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{v,u} \mathbf{W}^l \boldsymbol{h}_u^{(l-1)}, \quad \boldsymbol{h}_v^{(l)} = \sigma\left(\boldsymbol{z}_v^{(l)}\right)$$

where α represents the attention coefficients.

2.2.2 Graph Transformer

Graph transformer model has brought surging interest in graph representation learning recently [35]. Current representative graph transformers integrate graph structural encodings into the input and attention map in the Transformer architecture. The input sequence is built by tokens generated with graph attributes. Specifically, some works [5, 6, 36–40] calculate node positional encodings beforehand and add them to the inputs before the attention module. Other works [5, 6, 25, 41, 42] add graph topology information on top of the attention layout as bias terms. Several works [21, 22, 43] combine message-passing GNNs and the attention mechanism together. Here we only focus on the former two types of graph transformers since they are currently most representative.

A basic Transformer consists of multi-head attention (MHA) and feed-forward network (FFN) which contains two linear layers. Given an input sequence $\boldsymbol{H} = [h_1, \cdots, h_S]^\top \in \mathbb{R}^{S\times d}$ where S denotes the sequence length and d is the hidden dimension, MHA first projects its input $\boldsymbol{H}$ to three subspaces: $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ with projection weight matrices $\boldsymbol{W_Q} \in \mathbb{R}^{d\times d_{\boldsymbol{K}}}, \boldsymbol{W_K} \in \mathbb{R}^{d\times d_{\boldsymbol{K}}}, \boldsymbol{W_V} \in \mathbb{R}^{d\times d_{\boldsymbol{V}}}$. The MHA output can be obtained:

$$\boldsymbol{H}' = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_{\boldsymbol{K}}}}\right)\boldsymbol{V} \tag{2.3}$$

where $d_{\boldsymbol{K}}$ is the second dimension of $\boldsymbol{K}$. In MHA, each attention head learns to measure the pairwise relationships between tokens in the sequence.

For a graph $G = (V, E)$ with nodes $V = \{v_1, \cdots, v_N\}$ and edges $E = \{e_1, \cdots, e_{|E|}\}$, here we list the formulation of Graphormer [6] as an example:

$$h_i^{(0)} = x_i + z^-_{\deg^-(v_i)} + z^+_{\deg^+(v_i)} \tag{2.4}$$

$$\boldsymbol{A}_{ij} = \frac{(h_i \boldsymbol{W_Q})(h_j \boldsymbol{W_K})^\top}{\sqrt{d_{\boldsymbol{K}}}} + bias_{\phi(v_i, v_j)} \tag{2.5}$$

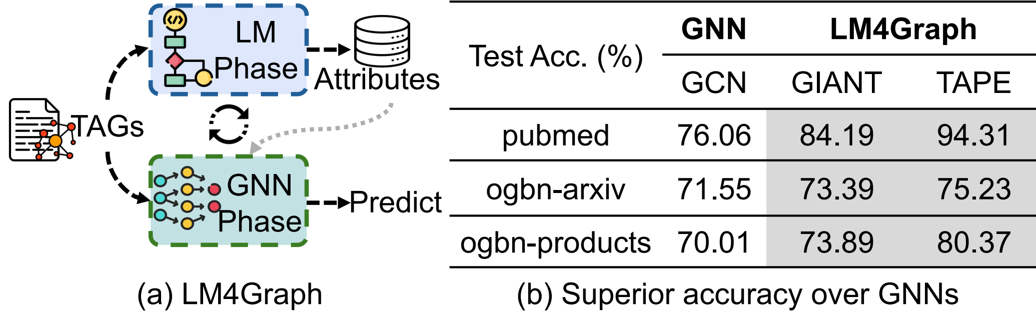
where $h_i^{(0)}$ is the beginning attribute of node i, x_i is the node feature, and $z^-, z^+ \in \mathbb{R}^d$ are learnable embeddings specified by the in-degree $\deg^-(v_i)$ and out-degree $\deg^+(v_i)$. The encodings in Equation 2.4 allow the attention to capture the node importance. $\boldsymbol{A}_{ij}$ is the $(i,\ j)$-element of Query-Key product matrix, namely the attention coefficient. $bias_\phi$ is a learnable scalar shared across all layers, and $\phi(v_i, v_j)$ is the distance of the shortest path (SPD) between node v_i and v_j, which contains the smallest number of hops that v_i needs to pass to reach v_j.

2.2.3 LM4Graph

Recent years have witnessed a surge of interest in LM4Graph, or language model-based graph learning, particularly for node classification tasks on text-attributed graphs (TAGs) [44]. As illustrated in Figure 2.3(a), existing LM4Graph approaches generally operate in two distinct phases: (1) the LM phase, where pre-trained language models serve as text encoders to generate node embeddings from textual attributes, and (2) the GNN phase, where message passing is performed over these embeddings to refine node representations. In summary, LMs leverage the local textual information of each node, while GNNs utilize the global structural connections among nodes. By leveraging rich textual semantics, LM4Graph methods have demonstrated superior performance compared to traditional message-passing graph neural networks (e.g., GCN [4]) across various TAG-related benchmarks, as evidenced in Figure 2.3(b). In this thesis, we focus on advancing node classification in TAGs by addressing these limitations and proposing a more integrated and efficient framework.

Formally, a TAG can be represented as $G = (V, A, \boldsymbol{s}_V)$ with nodes $V = \{v_1, \cdots, v_N\}$, adjacency matrix $A \in \mathbb{R}^{N \times N}$, and a sequential text attribute $\boldsymbol{s}_v \in D^{L_v}$ for each node $v \in V$. D is the words dictionary and L_v is the text length.

LM Phase. An LM is employed as deep embedding techniques to encode text attributes and capture local textual information of each node. Given a pre-trained network $\mathcal{LM}$, such as BERT [45] or DeBERTa [46], the encoded embedding of node



Test Acc. (%)	GNN	LM4Graph	
	GCN	GIANT	TAPE
pubmed	76.06	84.19	94.31
ogbn-arxiv	71.55	73.39	75.23
ogbn-products	70.01	73.89	80.37

(b) Superior accuracy over GNNs

FIGURE 2.3: (a) The architecture of LM4Graph methods. (b) LM4Graph methods outperform classical GNNs on TAG-related tasks by a large margin.

v is as follows:

$$\boldsymbol{h}_v = \mathcal{LM}(\boldsymbol{s}_v) \in \mathbb{R}^d \tag{2.6}$$

where $\boldsymbol{h}_v$ is the encoded embedding of the LM, and d is the dimension of the embedding vector. To perform downstream node classification, the embedding is treated as input features to train the GNN.

GNN Phase. A GNN updates via the message passing mechanism [30], which first aggregates the feature vectors from a node's neighbors and then utilizes the aggregated information to update the node's representation. The l-th layer of a GNN can be expressed as:

$$\boldsymbol{h}_v^l = \phi^l \left(\rho^l \left(\left\{ \boldsymbol{h}_u^{l-1} \mid u \in \mathcal{N}(v) \right\} \right), \boldsymbol{h}_v^{l-1} \right) \in \mathbb{R}^d \tag{2.7}$$

where $\boldsymbol{h}_v^l$ is the representation of node v at layer l and $\mathcal{N}(v)$ is the neighboring nodes of node v. ρ^l is the aggregation function (e.g., sum, mean, etc.) that takes the embeddings of node v's neighboring nodes to get an intermediate aggregated result. ϕ^l is the update function that takes the aggregated results and the previous-layer representation to obtain the learned representation. The final representation goes through a fully connected layer and a softmax function for the label prediction.

2.3 Graph Training System

2.3.1 Distributed GNN Training

Existing solutions to this problem can be categorized into two directions. First, some works [9, 47–50] utilize *sampling-based training*, which only selects a subset of nodes and edges to be trained at each iteration. Although this method can reduce memory consumption, it requires careful consideration of the sampling strategy and may lead to the loss of important neighborhood information, suffering from model accuracy loss [3, 51, 52]. Second, distributed *full-graph training* [52–57] allows training over

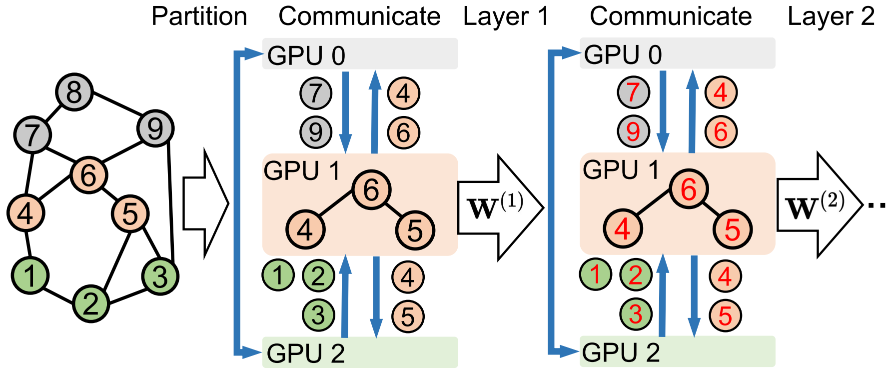


FIGURE 2.4: Example of vanilla distributed GNN training. For the partition on GPU-1, node 4 requires extra features of node 7 from GPU-0 and node 1 from GPU-2 to update its embedding in each layer.

full graphs on multiple devices or servers to reduce the computing time and memory demand on each GPU. Therefore, it preserves the complete input graph information so that model accuracy can be better preserved. Due to its promising features, our focus in this thesis lies in developing full-graph training systems.

Figure 2.4 shows the vanilla distributed GNN training on full graphs. The whole input graph is first partitioned via a graph partitioning algorithm (e.g., METIS [58]) on the host side to fit into a single GPU. Since each node and its features will only be assigned to one GPU, there exist nodes that are connected to the local partition but reside on other partitions, dubbed **boundary nodes**. For instance, node 4 requires the embeddings of boundary node 7 from GPU-0 and node 1 from GPU-2 to update itself during message passing. In the backward pass, the embedding gradients of boundary nodes are also transferred. Therefore, both embeddings and embedding gradients of boundary nodes, denoted as *messages*, will be transferred in each layer.

2.3.2 Large-Scale Graph Transformer Training

Large-scale graph transformer training relies on long sequences, which is crucial for model quality and the development of versatile graph transformer application scenarios. However, existing graph transformer research works [5, 6, 21, 22] fail on long sequence training due to missing training adaptations, which we will elaborate in the following. For better illustration, we categorize current graph learning tasks into two types to explain the process and necessity of training in long sequences: (1) graph-level training, and (2) node-level training.

Long Sequence for Graph-level Training. For graph classification tasks, long sequences are necessary when training large-scale graphs. For such tasks, the input sequences represent a set of graphs while the output is a set of labels representing the

types of corresponding graphs. When processed by graph transformers, all nodes in each input graph need to be encoded as input tokens and are concatenated into an input sequence. As such, the length of each sequence equals the number of nodes in each graph. In this task, if the graph size, i.e., the number of nodes, grows very large, the input sequence can be too long to be trained by current methods. For instance, the MalNet [26] dataset contains graphs with up to 552K nodes, indicating a single sequence length of prohibitive 552K.

Long Sequence for Node-level Training. These tasks classify each node in an input graph with a specific label. In the node classification task, the input sequences can either encode all nodes in the graph or a mini-batch of nodes. For the former case, the input sequence can be enormously long for large-scale graphs, which is not supported by most models. For the latter, with a larger batch size, both the training throughput and the trained model quality can be improved. Both cases validate the necessity and advantages of long sequence training.

However, existing graph transformers have some inherent constraints in performing the above tasks. While graph-level scenario has been explored in [5, 6], existing endeavors do not generalize to large-scale graphs endemic to node-level prediction. Our TorchGT strives to include both tasks by joint algorithm-system design. The scale of graphs applicable to current models is still limited, thus leaving long sequence training still an urgent necessity. Besides, training large-scale graphs in short sequence suffers from lower training throughput, downgraded model quality and limited graph transformer applications. Figure 2.5 illustrates the impact of sequence length on the test accuracy of two representative models Graphormer [6] and NodeFormer [25] on two datasets. Both models show superior performance on longer sequences. On the AMiner-CS dataset, Graphormer with a 4K sequence length improves the test accuracy by up to 0.9% compared to the short sequences. On the Pokec dataset, sampling-based NodeFormer with 100K sequence length outperforms the case with 10K sequence length by a staggering 12% accuracy. These results necessitate the need for long sequence training of graph transformers.

2.3.3 LM4Graph Training

Current LM4Graph methods typically adopt a decoupled training process comprising two separate phases: the LM phase and the GNN phase. In the LM phase, to overcome the limitations of shallow embedding approaches, researchers leverage deep embedding techniques by fine-tuning pre-trained language models on the text data associated with the graph. This step generates node embeddings that are specifically adapted to the context of text-attributed graphs. The LM phase often requires substantial

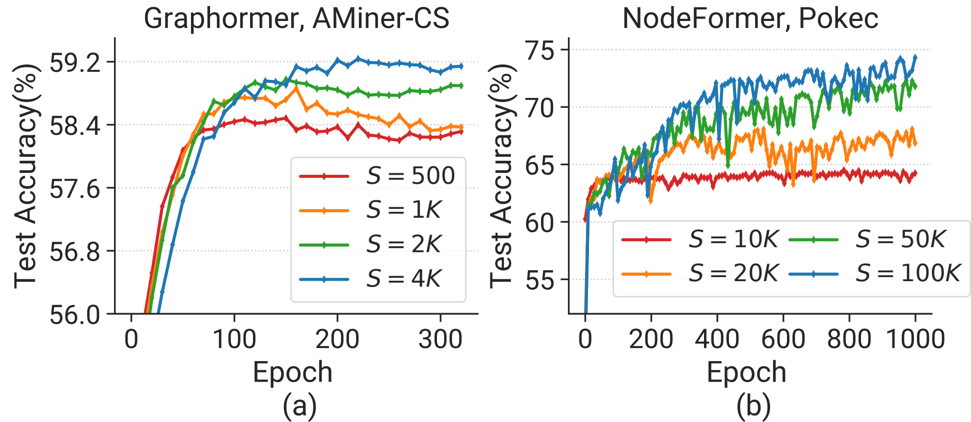


FIGURE 2.5: The test accuracy of graph transformers when trained with different sequence lengths S.

computational resources, as training is usually implemented using data parallelism across multiple GPUs to handle the large-scale language model [59, 60].

Following this, the GNN phase uses the generated node embeddings as input to train a GNN model, which is typically much smaller in scale. As a result, the GNN training can be efficiently performed on a single GPU with limited memory requirements. However, because these two phases are trained independently, the overall process becomes fragmented. This decoupled design leads to lower training efficiency and poses challenges for end-to-end optimization, as the language model and graph neural network cannot be jointly fine-tuned to directly capture graph-specific semantics.

2.4 Literature Review

This chapter provides the related works about training optimizations of each graph learning method.

2.4.1 Graph Neural Network Optimization

Prior works propose new frameworks to accelerate distributed GNN training [61], e.g., AliGraph [55] and NeuGraph [62]. However, these methods all store the partitions in CPUs, which inevitably incur frequent CPU-GPU swapping and largely impair the benefits of distributed training. DistDGL [54] provides the scaling results but only on sampling-based methods. LLCG [63] totally drops dependent information between partitions and adds a global correction server to compensate for the error with redundant overhead. Moreover, those works only support mini-batch training on graphs rather than full-graph training. Different from the above sampling-based works, ROC [51] accelerates distributed full-graph training, but it also stores the partitions in

TABLE 2.1: Comparison of prior works on GNN quantization.

Features	EC-Graph [71]	BiFeat [73]	Degree-Quant [72]	SYLVIE [18]
Distributed Environment	✔	✔	✔	✔
GPU Support	✘	✔	✘	✔
Full-Graph Support	✔	✘	✔	✔
Message Quantization	✔	✔	✘	✔
Adaptive Configuration	✔	✘	✘	✔
Deep GNN Support	✘	✘	✘	✔

CPUs with huge CPU-GPU data transfer cost. Some works [64–66] improve the performance of full-graph training at scale, but suffer from extra computation burden due to the complexity of introduced operations. BNS-GCN [52] adopts random sampling on the boundary nodes and shows impressive acceleration, yet it risks downgrading the model performance by dropping node connections and its performance is highly dependent on the graph structure. Some works [56, 57] propose to use stale messages for high efficiency. [56] adopts asynchronous training in GNN scope to hide partial communication costs.

In recent years, there also emerge various works which apply the quantization technique on GNNs. Model quantization [67, 68] via simulation for memory reduction is a common direction, with the underlying computation still occurring in the 32-bit full precision. EXACT [69] aims to reduce memory demand at the cost of extra training time overhead, seriously deteriorating the training efficiency. Other works like [70] quantize GNN models for efficient inference. EC-Graph [71] also optimizes distributed GNN training by quantizing the communication but only for CPU clusters and empirically adjusts the quantization configuration. Degree-Quant [72] quantizes GNN models and parameters on small graphs, but the training efficiency on GPU clusters even downgrades. BiFeat [73] mainly targets mini-batch training and suffers from non-negligible accuracy loss. Table 2.1 summarizes the main differences between our work SYLVIE and some GNN quantization works. Compared with our work, all these methods have different targets or only consider small-scale datasets. More importantly, none of them considers the generality of deeper or special GNNs. Different from the aforementioned quantization works, we explore the opportunity of quantizing communication in GNNs together with multiple system techniques, and adjust them dynamically to reach an efficiency-accuracy balance so as to fit for versatile GNN architectures.

2.4.2 Graph Transformer Optimization

As a new kind of graph learning algorithms outperforming traditional GNNs, many graph transformer architectures have arisen in recent years. Among them, some works [6, 10, 11, 13, 36] utilize standard attention as foundation encoders to capture the all-pair interactions between nodes, leading to quadratic computation complexity. Some other works [36, 41, 74] adopt sampling or pooling methods which only select a subset of nodes to be trained at each iteration, without reducing the computation complexity. [25, 75, 76] use self-defined adapted attention with poor generality and scalability.

As for system optimizations for transformers, sparse attention [77–80] has been widely studied in NLP area for linear complexity. Borrowing this, [5, 21, 22] directly apply graph topology in the attention computation. Besides, multiple works for LLM [14–17] split the input sentence sequences and train distributedly for larger scalability. To overcome the computation bottleneck of attention, several efforts [21, 22, 25, 75–78] have been made to reduce the computation cost of attention module. Some works like [81] targeted for LLMs prune the attention module and leave a major backbone to reduce the computation cost. [77, 78] design special patterns to reduce the pair-wise computation, making the dense attention sparse. Among them, a few works [25, 36, 75, 76] modify the generation of input sequences by harnessing neighbor sampling [36] similar adopted in classic GNNs or use self-defined adapted attention modules. Nonetheless, these works either sacrifice model precision or limit the graph transformer to a single application, e.g., node classification, failing to generalize to versatile graph tasks. On the other hand, there are many existing efforts like sparse attention [22, 77, 78]. In language models, there are many block-based sparse attention adaptions, e.g., performer [77] and BigBird [78]. Performer [77] utilizes a technique called positive orthogonal random feature mapping to approximate the standard attention mechanism. This method allows Performer to significantly reduce the computational cost of attention module requirements. BigBird [78] replaces the standard transformer's self-attention mechanism with a hybrid sparse attention model, which includes global attention, sliding window attention, and random attention pattern together. We highlight the advantages of TorchGT over some typical works in Table 2.2.

However, these attention mechanisms work for natural language processing but fail to capture the inherent structure information in graphs, thus resulting in giant model degradation [22]. Exphormer [22] applies graph topology to attend nodes, which can maintain the most important graph-specific information but still suffers from model convergence loss. Besides, Exphormer only limits its implementation with a GNN-encoding-based graph transformer architecture, i.e., GraphGPS [21]. Inspired by Exphormer [22], we find that the graph's structural information is critical to designing

TABLE 2.2: Comparison of existing graph transformer methods and TorchGT.

Features	Graphormer [6]	GT [5]	NAGphormer [36]	EXPHORMER [22]	TorchGT [19]
Distributed Environment	✘	✘	✘	✘	✔
Large Graph Support	✘	✘	✔	✘	✔
Linear Computation Complexity	✘	✘	✘	✔	✔
Task-agnostic	✘	✘	✘	✔	✔
Scalability	✘	✘	✘	✘	✔

TABLE 2.3: Comparison of existing LM4Graph methods and UniTG.

Features	GIANT [60]	GLEM [82]	TAPE [59]	UniTG
Seamless Training Process	✘	✔	✘	✔
Deep GNN Support	✘	✘	✘	✔
Hardware Utilization	✘	✘	✘	✔
Graph-aware	✘	✘	✘	✔
Flexible Architecture	✘	✘	✔	✔

an efficient attention mechanism.

2.4.3 LM4Grah Optimization

LM4Graph is a branch of graph learning methods which incorporate LM model for analyzing. Current methodologies can be broadly categorized into two paradigms. The first adopts a cascaded architecture [29, 59, 60, 83–88], wherein the LM first encodes node embeddings offline, which are then cached and subsequently fed into a GNN for training. For instance, GLEM [82] introduces a variational expectation-maximization (EM) framework to jointly train LMs and GNNs, iteratively refining predictions. Similarly, GIANT [60] leverages self-supervised learning with XR-Transformers [89] to derive node embeddings, achieving state-of-the-art results on diverse graph benchmarks and underscoring the importance of high-quality node features in attributed graphs. Meanwhile, [90] proposes a distillation framework that transfers structural knowledge from GNNs to LMs. A key aspect of these joint-training approaches is the bidirectional interaction between LMs and GNNs, such as exchanging pseudo-labels [82] or hidden representations [90].

The second group of approaches [6, 82, 91] employs an alternating optimization strategy, iteratively updating the LM and GNN phases, with each phase leveraging the outputs of the preceding one to enhance performance. However, most existing methods decouple these two phases, adopting a rigid one-at-a-time update scheme that

requires cold-starting the system when switching phases. This design leads to significant inefficiencies and may introduce latent errors during training. With the effectiveness of LLMs across a spectrum of NLP tasks prevailing, [59] incorporates LLMs into TAG tasks to fully explore their power. [92–94] have sought to evaluate the capacity of LLMs in understanding graph-structured data and use them to enhance the graph processing capabilities. We summarize the characteristics of current LM4Graph methods in Table 2.3.

Chapter 3

Sylvie: Efficient Graph Neural Network Training System

This chapter presents the research[1] to improve the training throughput of GNNs on deeper and more intricate model architectures. While recent system advancements can improve the training throughput of GNNs, they fail to comprehensively consider diverse opportunities for acceleration.

3.1 Introduction

In recent years, GNNs have become very popular and exhibited state-of-the-art performance in learning structured data like graphs. Driven by such breakthroughs, GNNs have been applied to a variety of tasks such as community detection [20], node classification [95–98] and link prediction [24]. GNNs capture the underlying dependencies of the given graph via message passing operations [99]. Despite their impressive performance on graph-related tasks, training GNNs on large-scale graphs containing millions to billions of nodes is still a long-standing issue, as extensive memory resources are needed for loading and computing input graphs [4, 69, 100] and the memory demand easily exceeds the memory capacity. This hinders the practical development of more sophisticated graph datasets and GNN architectures.

Existing solutions to this problem can be categorized into two directions. First, some works [9, 47–50] utilize *sampling-based training*, which only selects a subset of nodes and edges to be trained at each iteration. Although this method can reduce the memory consumption, it requires careful consideration of the sampling strategy and may lead to the loss of important neighborhood information, suffering from model accuracy loss [3, 51, 52]. Table 3.1 shows the test accuracy comparison of GraphSAGE

[1] The work in this chapter has been published in [18].

Table 3.1: Test accuracy of training GraphSAGE on the Ogbn-products dataset with different sample sizes.

	Sampling-based			**Full-Graph**
Sample Size	5	10	15	
Accuracy (%)	73.55	74.87	76.84	**79.19**

model when trained in sampling way and full-graph. Apparently, the accuracy of sampling-based mode is always lower than that of full-graph training, especially as the sample size decreases. Second, distributed *full-graph training* [52–57] allows training over full graphs on multiple devices or servers to reduce the computing time and memory demand on each GPU. It preserves the complete input graph information so that model accuracy can be well preserved. Due to its promising features, our focus lies on developing full-graph training systems. First the whole graph is split into several subgraphs so that each can fit in a single GPU, and then trains subgraphs on each GPU in parallel.

While existing GNN training systems can achieve higher training throughput, nonetheless, they still meet some deficiencies in practice.

First, **they may compromise the model quality and fail to support universal GNN models**. As aforementioned, though sampling-based methods [3, 7–9, 47, 101] can reduce the memory footprint, they will forfeit crucial neighborhood information, ultimately resulting in obvious model accuracy degradation ($> 2\%$) compared with vanilla full-graph training as shown in Table 3.1. On the other hand, the fast development of GNN algorithms raises urgent demand for the compatibility of the underlying training systems. However, state-of-the-art full-graph training systems [52, 54, 56, 57] only consider specific model architecture (i.e., Graph Convolutional Network (GCN) [4]) with shallow layers (two or three) [33, 52, 56]. They fail to accommodate to more advanced GNNs with complex aggregators (e.g., LSTM and attention networks [24, 102]) or deeper GNNs such as DAGNN [34]. Consequently, this limitation of current works leads to model accuracy decline and restricts their applications in practice.

Second, **limited scalability and generality for distributed GNN training**. Distributed full-graph GNN training allows learning over the whole input graph directly, but current methods realize scalability by sacrificing convergence and the model quality [52, 56, 57]. Some work [54, 56, 66] perform badly when deployed in a large-scale datacenter, where communication overhead dominates. As such, current methods can only apply tasks to small-scale graphs (e.g., hundreds to thousands of nodes) or with a shallow structure (e.g., less than three layers) in GNN models. Moreover, there are emerging deeper and more powerful GNNs with more sophisticated aggregators (e.g.,

LSTM and attention networks) [102]. It is hard to generalize these approaches to those novel GNN models or datasets.

Third, **existing systems overlook the joint optimization opportunities tailored to GNNs**. Generally, GNN training is often hampered by substantial communication and memory bottlenecks, in contrast to DNNs whose computation tends to be the bottleneck (§3.2.1). Though compression has been well-studied in distributed DNN training [103, 104], it cannot be grafted to GNN case easily. The bottleneck of distributed DNN training stems from the weights and weight gradient transfer. However, for GNNs, the size of weight gradients are much smaller (data), with the layer-wise exchange of embeddings and embedding gradients being the main bottleneck. Regrettably, prevalent frameworks (e.g., DGL [105] and PyTorch Geometric [106]) do not provide effective distributed full-graph training support. To this end, some works [51, 54, 64, 66] have made efforts to improve the situation. However, they still bear significant communication overhead. Furthermore, existing systems [7, 52, 57, 62, 66] focus on system-level optimizations while ignoring exploitation of graph data information. Some works like [107] also utilize the input graph to enhance the optimization decisions, but they focus on different input information and only cope with single-GPU training on small-scale graphs. They typically target a single optimization aspect and fail to improve training efficiency while preserving model accuracy under the distributed setting.

Fourth, **substantial communication overhead**. GNN has brought breakthroughs in a wide range of graph-based tasks such as link prediction [24], node and graph classification [95]. Recently more sophisticated aggregators and the emerging giant real-world graphs have shown promising results on GNN training, but those improvements often come at a cost of significantly increased memory consumption. To cope with the problem, from the algorithmic perspective, sampling-based methods [3, 47] which adopts mini-batch training are raised to train the model with a reduced memory footprint. However, sampling-based methods often need centralized data storage such as the host memory, which also causes significant data transfer costs [8]. Besides, sometimes sampling introduces extra time overhead due to the complicated sampling strategies [108, 109]. From the system perspective, current GNN training frameworks such as DGL [105] and PyTorch Geometric [106] have scalability limitations and lack good support for distributed full-graph training. DistDGL [54], an advanced version of DGL, supports distributed training by partitioning the graph across multiple GPUs or nodes to scale out the training. However, the obligatory communication between workers impairs training efficiency badly. To solve this, some work [51, 64, 66] proposed various approaches to realize efficient distributed full-graph training, but still suffer from heavy communication overhead.

Finally, **failing to exploit GNN input information**. Nowadays there are emerging more powerful GNN models with various types of aggregators, increasing aggregation depth and so on. In addition, the diversity of input graphs further enriches the problem. These information greatly impact the effectiveness of system optimization choices. Some works like [107] also utilizes the input information to enhance the optimization decisions, but it focuses on different input information and only supports single-GPU training on small-scale graphs. Unfortunately, current works [7, 52, 57, 62, 66] on large-scale GNN training all follow a simple scheme and fail to craft a targeted optimization strategy that maximizes the whole system performance for each particular GNN task.

To bridge these gaps, we design Sylvie, a novel distributed full-graph GNN training system that supports various types of GNNs. It jointly optimizes training from three granularities, including *data*, *time* and *execution* aspects. The core design of Sylvie comes from the following three insights. First, *GNN input information guides the system optimizations*. Present GNN models are diverse in layer sequences, aggregation methods, and depths. Similarly, the input graphs vary in node properties and features. By profiling and analyzing these two, we find that valid optimization suggestions tailored to specific GNN tasks can be obtained from the input information. Second, *adaptive optimizations by monitoring the training process can preserve the quality of universal GNN models*. By integrating the online training characteristics, we are the *first* to enable the training acceleration of deeper GNNs while greatly preserving model quality, in contrast to current systems that only support limited GNN models with two or three layers. We find that extreme quantization (e.g., 1-bit, 2-bit) can be applied to communicated messages of GNNs, which is typically not applicable to DNN models. We also observe convergence can be further improved by adopting different optimization choices along the training process. Third, *the benefits of advanced execution mode (i.e., asynchronous pipeline) can be maximized by curtailed communication*. In the original case, as the model size increases and cluster size scales, the communication overhead in distributed learning dominates the training time. The communication is frequent and heavy while the bandwidth of network interfaces is limited, which significantly diminishes the benefits of pipelining. However, pipelining the reduced communication can manifest its advantages and greatly contribute to efficiency.

Integrating the above insights, we build a model-agnostic GNN training system Sylvie, consisting of several key components to facilitate training while improving model quality. In detail, it designs the optimization from two stages: **Offline Stage** is in charge of extracting and analyzing the input graph information to guide the downstream system optimizations. **Online Stage** is responsible for deploying the combined optimizations and monitoring the training status to dynamically adjust the optimization

configurations in a 3D-adaptive way. The coordination between different modules in SYLVIE greatly improves the training efficiency for shallow and deep models by jointly combining data-, time-level quantization and pipelining. Through extensive experiments, we show SYLVIE substantially outperforms SOTA baselines by up to 17.2× speedup across various models without hurting their accuracy.

In summary, we make the following contributions:

- Unlike existing systems that solely test on shallow GNNs, SYLVIE stands out as the *first* system designed to accelerate universal GNNs in practice. Our approach is supported by both theoretical proofs and extensive experiments conducted on versatile models and datasets, demonstrating that SYLVIE can effectively expedite training while preserving model accuracy.
- We are the *first* to explore the GNN input-level properties (§3.4) and exploit their potential performance benefits to adjust the system-level optimizations tailored to distributed full-graph training. Besides, we pioneer in identifying the unique opportunities of quantizing communicated messages in GNNs.
- We synthesize a set of system optimizations to improve the GNN training efficiency in a *3D-adaptive* manner, including a novel *data-adaptive* and *time-adaptive* quantization algorithm (§3.5.1) and an *execution-adaptive* scheme (§3.5.2).

3.2 Background and Motivation

3.2.1 Communication Bottleneck

Unlike classical distributed DNN training where training samples are independent of each other, it is non-trivial to apply data parallelism on GNNs due to the node dependency between subgraphs, leading to obligatory data communication overhead. The communication overhead is non-trivial since the amount of boundary messages can be excessive, as shown in Table 3.2. In addition, such cost becomes more intensive as the number of partitions and layer size grow larger. For node 4 on GPU-1, communication of nodes 7 and 1 should be done before the computation of layer 1 begins. The same process repeats for the rest layers in a synchronous way. In this case, excessive communication will take up the training time and block the subsequent computation.

To show the massive communication cost more intuitively, we profile the epoch time along with its breakdown trained on two models and three representative datasets, as shown in Figure 3.1. We can see for all cases, the communication time nearly dominates the entire training process (up to 89.23%), while the computation and the transfer of weight gradients (all-reduce) only occupy a very small portion. This

Table 3.2: Data volume of communicated messages (embeddings & embedding gradients) and weight gradients of three models on the Ogbn-products dataset over 4 GPUs. 4-GCN-256 means a 4-layer GCN with the hidden size of 256.

Model	Embeddings	Embedding Gradients	Total Messages	Weight Gradients
4-GraphSAGE-128	1.56 GB	1.55 GB	3.11 GB	0.40 MB
4-GCN-256	3.10 GB	3.10 GB	6.20 GB	0.65 MB
3-GAT-256	2.07 GB	2.07 GB	4.14 GB	0.41 MB

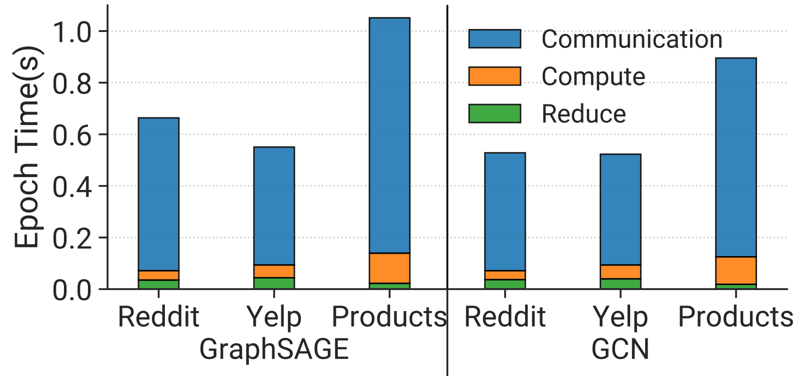


Figure 3.1: Training time per epoch in vanilla distributed training with DGL on a single server (8 GPUs).

is different from distributed DNN training where the transfer of weight gradients are more costly. The scalability and efficiency of distributed GNN training thus are seriously restrained due to this excessive communication overhead.

Prior works propose new frameworks to accelerate distributed GNN training, e.g., AliGraph [55] and NeuGraph [62]. However, these methods all store the partitions in CPUs, which inevitably incur frequent CPU-GPU swapping and largely impair the benefits of distributed training. DistDGL [54] provides the scaling results but only on sampling-based methods. LLCG [63] totally drops dependent information between partitions and adds a global correction server to compensate for the error with redundant overhead. Moreover, those works only support mini-batch training on graphs rather than full-graph training. Different from the above sampling-based works, ROC [51] accelerates distributed full-graph training, but it also stores the partitions in CPUs with huge CPU-GPU data transfer cost. Some works [64–66] improve the performance of full-graph training at scale, but suffer from extra computation burden due to the complexity of introduced operations. BNS-GCN [52] adopts random sampling on the boundary nodes and shows impressive acceleration, yet it risks downgrading the model performance by dropping node connections and its performance is highly dependent on the graph structure.

Most recently, some works [56, 57] propose to use stale messages for high efficiency. [56] adopts asynchronous training in GNN scope to hide partial communication costs. However, the benefits of simply pipelining computation and communication are limited. Moreover, asynchronous training could introduce stale messages, which may affect the convergence of model. Since when the cluster size scales and layer size increases, communication overhead dominates the training time and the efficacy will corrupt badly, as shown by results in Table 3.7. Differently, we find that the benefits of pipeline can be maximized to further improve the training throughput by overlapping the reduced communication cost with computation. CAGNET [66] raises different dimension partitioning methods to boost training by slicing the node features to sub-vectors, at the extra communication and synchronization costs.

3.2.2 Quantization for GNNs

As a common technique, model quantization has already been applied to accelerate inference [110, 111], or compress activations to reduce memory consumption during training [69, 112]. Different from these works, we aim to speed up the distributed GNN training. Gradient compression is also extensively studied to reduce the communication cost in distributed DNN training [104, 113–115]. However, all those works target large models where the bottleneck stems from the transfer of **weights** and **weight gradients**. However, the size of weight gradients is much smaller for GNNs, with the layer-wise exchange of **embeddings** and **embedding gradients** being the main bottleneck. To better illustrate it, we train three representative GNN models on the Ogbn-products dataset and record the transferred volume of messages and weight gradients in Table 3.2. We can clearly observe that the size of weight gradients in the GNN case is far smaller than that of transferred embeddings and embedding gradients. For a four-layer GCN with the hidden size of 256, the weight gradients only take up to 0.65MB but with total 6.2GB communication volume (3.1GB embeddings and 3.1GB embedding gradients). Hence, compression methods in distributed DNN training cannot be simply grafted to our scenario since the communication of layer-wise messages is far more costly than that of the weight gradients. Therefore, we have to design a new efficient compression method dedicated to GNNs.

Though there are many works on DNNs, quantization on GNNs is still in its infancy. In recent years, there also emerge various works which apply the quantization technique on GNNs. Model quantization [67, 68] via simulation for memory reduction is a common direction, with the underlying computation still in the 32-bit full precision. EXACT [69] aims to reduce the memory demand at the cost of extra training time overhead, seriously deteriorating the training efficiency. Other works like [70] quantize GNN models for efficient inference. EC-Graph [71] also optimizes distributed GNN

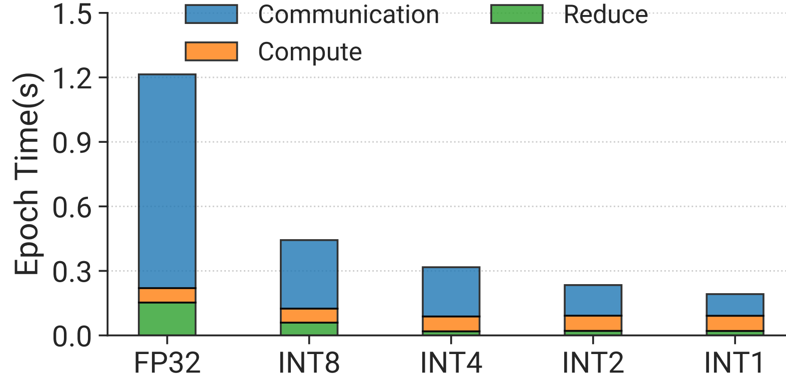


FIGURE 3.2: Training time per epoch and its breakdown when using different quantization bit-widths to train GraphSAGE on Yelp over 8 GPUs.

training by quantizing the communication but only for CPU clusters and empirically adjusts the quantization configuration. Degree-Quant [72] quantizes GNN models and parameters on small graphs and the training efficiency on GPU clusters even downgrades. BiFeat [73] mainly targets mini-batch training and suffers from non-negligible accuracy loss. Table 2.1 summarizes the main differences between SYLVIE and some GNN quantization works. Compared with our work, all these methods have different targets or only consider small-scale datasets. More importantly, none of them considers the generality of deeper or special GNNs. Different from the aforementioned quantization works, we explore the opportunity of quantizing communication in GNNs together with multiple system techniques, and adjust them dynamically to reach an efficiency-accuracy balance so as to fit for versatile GNN architectures.

Benefits of Quantization. Quantization of the interacted messages greatly reduces communication time. To show this, we take training GraphSAGE as an example, of which the results are shown in Figure 3.2. Clearly, the communication overhead decreases rapidly with the decrease in bit-widths. In particular, 1-bit quantization cuts down almost 89.8% communication overhead and 84.2% training time per epoch compared to the full-precision case.

Challenge of Quantization. Quantization of the interacted messages can greatly reduce the communication time. As shown in Figure 3.2, the communication overhead decreases rapidly with the decrease in bit-widths. In particular, 1-bit quantization cuts down almost 89.8% of communication overhead and 84.2% of training time per epoch compared to the full-precision case. However, it also deteriorates the accuracy. Particularly, lower bit-widths come with more accuracy reduction. To further demonstrate this, we showcase the experiment results over various models and datasets in Table 3.3. All experiments are done under a fixed number of epochs which is sufficient for all models to converge. The number of epochs is set to 2000 for GraphSAGE and GCN, 800 for JKNet. It is apparent that the smaller the bit-widths, the greater

TABLE 3.3: The test accuracy (%) of three models trained with different quantization bit-widths. 4-GCN denotes a GCN with 4 layers and the other models are similar.

Dataset	Model	FP32	INT8	INT4	INT2	INT1
Amazon	4-GraphSAGE	81.29	81.09	79.14	79.12	79.09
Amazon	4-GCN	53.7	53.59	53.53	53.34	53.16
Reddit	8-JKNet	92.75	92.66	92.73	91.99	90.91

reduction in accuracy. The essential reason is that the dequantization cannot accurately recover the original messages and has a variance term $\mathrm{Var}(\tilde{\boldsymbol{h}}^{(l)}) = \frac{D \cdot scale^2}{6}$ (D is the dimension of hidden layers), even though the expected dequantized message is unbiased $\mathbb{E}\left[\tilde{\boldsymbol{h}}^{(l)}\right] = \mathbb{E}[\mathrm{Deq}(\mathrm{Q}(\boldsymbol{h}^{(l)}))] = \boldsymbol{h}^{(l)}$. Based on this analysis, to maximize the benefits brought by quantization without sacrificing the model quality, it is necessary to dynamically adapt the quantization level.

We do not quantize activations or weights like previous works [110, 111] because (1) computation only occupies a very small portion while communication of embeddings & feature gradients dominates the training overhead (Figure 3.1); (2) unique sparse computations in GNNs, e.g., SpMM in cuSPARSE, lack support for very low precision computation, unlike dense operations (e.g. GEMM) in DNNs which support fast low precision computation.

3.2.3 Pipeline of Distributed GNN Training

While quantization can significantly reduce the communication overhead, it cannot completely eliminate the communication latency. Pipelining the layer-wise communication and computation [56] shows the potential to fully hide the communication time. Different from synchronous training, the model directly begins each layer's computation with the stale messages obtained from the previous epoch, with the communication proceeding between partitions concurrently. The currently overlapped communication is for the use of the next epoch, ensuring the data integrity when the computation starts. The pipeline technique allows for overlapping computation and communication, thus obtaining training speedup.

Challenge of GNN Pipeline Training. The efficiency benefits of simply pipelining computation and communication are limited in GNN training. When the cluster size scales and layer size increases, the communication overhead dominates the training time and the efficacy will corrupt badly, as shown by results in §3.6.1. A way to improve the pipelining efficiency is that each layer is computed by using the information from several rounds ago. Existing works [57] utilize historical messages via cache to

improve the pipelining efficiency, but come with increased training error when the number of stale epochs is large. Considering these limitations, we propose to jointly exploit the quantization and pipeline strategy, which inherits both benefits including bounded staleness control and minimized communication latency. We find that the benefits of pipeline can be maximized to further improve the training throughput by overlapping the reduced communication cost with computation.

Different from works in traditional DNNs that quantize all the activations [69, 112] or models [67, 72, 116], we propose quantizing only the exchanged messages to reduce the communication cost in distributed GNN training via the stochastic integer quantization strategy [112]. Take the subgraph on GPU-1 in Figure 3.4 as an example, at the l-th layer forward pass of the model, the boundary nodes' (e.g., nodes 4, 5 and 6) embeddings are quantized to b-bit integers with low precision.

3.3 System Overview

To achieve efficient distributed GNN training while maintaining model accuracy, we design Sylvie as depicted in Figure 3.3. It realizes the dynamic optimization via two key stages: **offline stage** for graph property profiling and **online stage** for improving the training efficiency and model performance. Each stage contains newly-designed module(s) for different purposes. Specifically, the offline stage contains one key module:

- *Graph Extractor*: exploits the input-level graph information for potential performance benefits and quantization suggestions in guiding the system-level optimizations.

The online stage contains three key modules:

- *Quant Orchestrator*: dynamically orchestrates the quantization of messages from both data- and time-adaptive perspectives.
- *Pipeline Adaptor*: adjusts the training between the synchronous and asynchronous modes in a staleness-bounded way.
- *Coordinator*: deploys the 3D-joint optimization decisions and keeps monitoring the training feedback from the training process, where DGL [105] and PyTorch [117] serve as the backend.

System Workflow. The system workflow of Sylvie is depicted by black arrows in Figure 3.3. In detail, when a user submits a large-scale GNN training task, *Graph Extractor* first investigates the input graph properties and gives an in-depth analysis of their importance in guiding the system optimizations. Specifically, it utilizes the node importance and graph structure to assist the quantization decision-making.

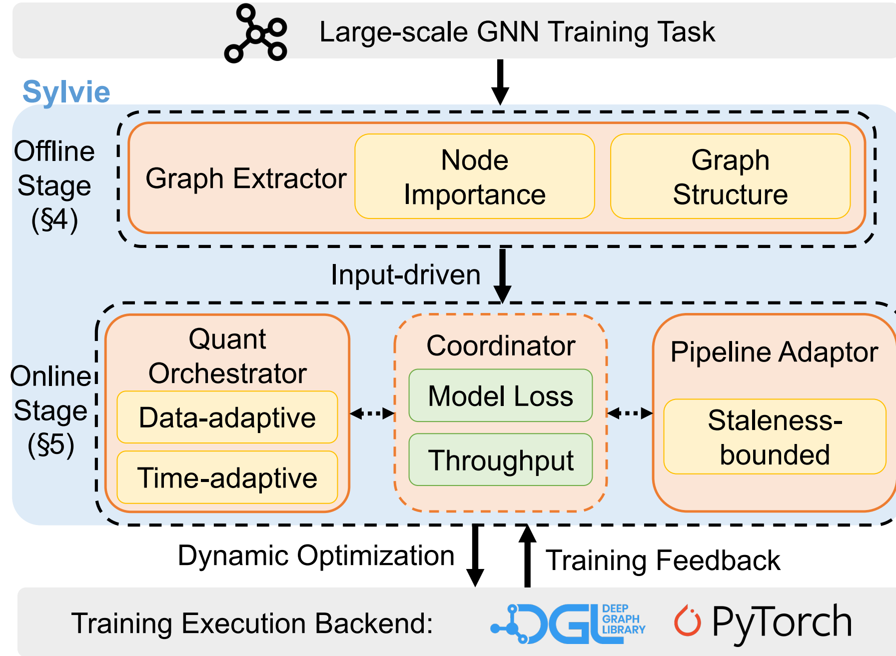


FIGURE 3.3: Overview of SYLVIE architecture and workflow.

After the pre-training analysis, the GNN task starts training, and meanwhile *Coordinator* receives and evaluates the training feedback such as the model loss and training throughput from the execution backend. Then *Quant Orchestrator* dynamically adjusts the quantization strategies by incorporating the input-driven knowledge and training feedback in a node- and time-adaptive way. *Pipeline Adaptor* also modulates the training mode for convergence improvement based on the evaluations. *Coordinator* then coordinates and deploys the optimizations jointly to prune the training. After the profiling analysis, the system starts training GNN. Meanwhile, *Coordinator* receives and evaluates the training feedback such as the model loss and training throughput from the execution backend. Then *Quant Orchestrator* dynamically adjusts the quantization strategies by incorporating the extracted graph properties and training feedback in a data- and time-adaptive way. *Pipeline Adaptor* also modulates the training mode for convergence improvement based on the evaluations. *Coordinator* then coordinates and deploys the optimizations jointly. We elaborate on the details in the following §3.4 and §3.5.

GNN Training with Sylvie. Here we illustrate the distributed GNN training process on the full graph with SYLVIE in Figure 3.4. The graph is first partitioned into several subgraphs and allocated to different GPUs or servers. In each partition, the inner node set (orange circles) as well as the boundary node set are constructed for the preparation of later message exchange. During both the forward and backward passes of each layer, *Quant Orchestrator* first adaptively quantizes messages sent to other partitions into low-bit integers by analyzing from both the time and node dimension (❶). Then those quantized data are broadcast to the corresponding

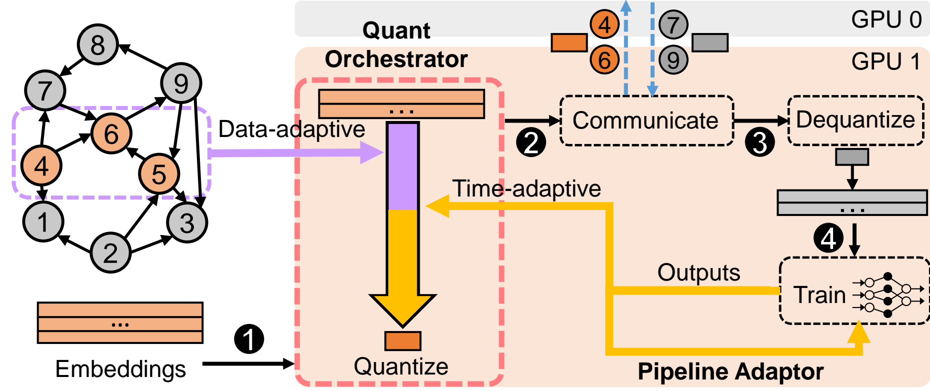


FIGURE 3.4: The training process with SYLVIE. Orange circles and rectangles represent nodes allocated to GPU-1 and their corresponding messages. The others in gray represent nodes/messages on other GPUs.

partitions through network communications (❷). Upon arrival at other partitions, these quantized messages are dequantized back to full-precision values for subsequent computation (❸). Then *Coordinator* deploys the joint optimizations on GNN model training and continuously monitors the feedback to improve training (❹). SYLVIE successfully balances the trade-off between training efficiency and model quality in a *3D*-adaptive manner including the *data* dimension, *time* dimension, and *execution* dimension.

3.4 Offline Stage of Sylvie

Though applying the lowest bit-width quantization can substantially improve the training efficiency, the model quality could also be impaired remarkably (as illustrated in Table 3.3). In this section, we show the input graph information can guide the system optimization based on our key observation that nodes with different in-degrees will favor different optimization decisions. Specifically, we first show why static low-precision arithmetic would fail by analyzing the sources of errors and how the node degree impacts the performance. Then we describe how we utilize this key insight in designing SYLVIE. We introduce *Graph Extractor* in the offline stage to extract and exploit graph and node properties to support the subsequent automatic quantization process.

Quantization of Sylvie. Different from prior works in traditional DNNs that quantize all the activations [69, 112] or models [67, 72, 116], we propose quantizing only the exchanged messages to reduce the communication cost in distributed GNN training via the stochastic integer quantization strategy [112]. Take the subgraph on GPU-1 in Figure 3.4 as an example, at the l-th layer forward pass of the model, the boundary nodes' (e.g., nodes 4, 5 and 6) embeddings are quantized to b-bit integers with

low precision. Specifically, at the forward pass of each l-th GNN layer, each GPU quantizes the embedding of each boundary node $\boldsymbol{h}^{(l)} \in \boldsymbol{H}^{(l)}$ to b-bit integers:

$$\hat{\boldsymbol{h}}_b^{(l)} = \left\lfloor \frac{\boldsymbol{h}^{(l)} - \min(\boldsymbol{h}^{(l)})}{scale} \right\rceil \tag{3.1}$$

where $\hat{\boldsymbol{h}}_b^{(l)}$ is a node embedding quantized to b-bit at the l-th layer, $scale = \frac{(\max(\boldsymbol{h}^{(l)}) - \min(\boldsymbol{h}^{(l)}))}{2^b - 1}$ is the scaling factor, $\min(\boldsymbol{h}^{(l)})$ is the zero-point, and $\lfloor \cdot \rceil$ is the stochastic rounding operation [118]. Before conducting layer computation, each GPU receives the quantized embeddings $\hat{\boldsymbol{h}}_b^{(l)}$ from the other GPUs and dequantizes them back to 32-bit full-precision floating-point values $\tilde{\boldsymbol{h}}^{(l)}$:

$$\tilde{\boldsymbol{h}}^{(l)} = scale \cdot \hat{\boldsymbol{h}}_b^{(l)} + \min(\boldsymbol{h}^{(l)}) \tag{3.2}$$

We showcase the training results when adopting the above quantization strategy on communicated messages in Figure 3.2. GraphSAGE is trained with different bit-widths b on Yelp dataset on a single server with eight GPUs. We can clearly see the communication overhead is substantially reduced with lower bit-widths, thus leading to a speedup on the whole epoch time. The vanilla method adopts full-precision numerics for communication, which occupies nearly the whole epoch time (0.99s out of 1.21s). Adopting 1-bit cuts down almost 89.8% communication overhead and 84.2% training time per epoch compared to the vanilla case. However, lower bit-widths also come with more accuracy loss. The stochastic integer quantization is a lossy compression method which inevitably introduces numerical precision loss.

From the above-unbiased nature of quantization, the subsequently calculated weights and gradients are also unbiased. But the variance term reveals that quantization still brings some extra noise to the messages. To investigate how the noise affects the model performance, we train three popular GNN models GraphSAGE, GCN and JKNet on the Reddit dataset with different bit-widths quantized communication. Specially, GraphSAGE and GCN are only set to have four layers while JKNet is deeper and has eight layers. The results are shown in Table 3.3. For all models, it is obvious that the loss in accuracy becomes severer when smaller bit-width quantization is applied on communicated messages. And the error aggregates more when the model becomes deeper for JKNet. Especially, adopting the static INT1 or INT2 quantization will cause a substantial accuracy drop, which shows more quantization error is introduced in training when using very low precision. This demonstrates the necessity of dynamic adaptations of the quantization level without sacrificing the model quality.

Sources of Errors. Many real-world graphs follow the power-law distribution [119] of node degrees. Such distribution leads to some nodes having a substantially larger number of neighbors than others (e.g., large node degree). The aggregation process which

fuses nodes' messages of their neighbors is the main source of substantial numerical errors. The errors become more significant as the node in-degree increases. Here we recapitulate the relation between quantization error and node in-degree following [72]. Taking the GCN layer as an example, supposing we have incoming embeddings X_i, after the aggregation phase, nodes with in-degree d have value: $Y_d = \sum_{i=1}^{d} \frac{1}{\sqrt{d_0 d}} W X_i$. it is trivial to get $\mathbb{E}(Y_d) = \mathcal{O}(\sqrt{d})$. When the model converges without over-fitting $\sum_{i \neq j} \operatorname{Cov}(X_i, X_j) \ll \sum_i \operatorname{Var}(X_i)$, the variance of the aggregated value is $\mathcal{O}(\sqrt{d})$. TThis observation demonstrates that in most GCN-based graph neural networks, both the mean and variance of aggregated outputs exhibit positive correlation with node in-degree. For other GNN models, similar conclusions can also be summarized from Figure 3 in [72]. Further, the quantization error in aggregation also introduces errors in subsequent weights. Through the update rule in GCN, we can get the weight gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{v=1}^{|V|} \sum_{u \in \mathcal{N}(v)} \frac{1}{\sqrt{d_v d_u}} \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_v^{(l+1)}} \circ f'(\boldsymbol{y}_v) \right) \boldsymbol{h}_u^{(l)\top} \tag{3.3}$$

where $\boldsymbol{y}_v$ is the updated embeddings. It is obvious that the larger aggregation error in $\boldsymbol{y}_v$ and $\boldsymbol{h}_u^{(l)\top}$, the larger error in the weight gradients, resulting in model quality degradation. The dequantization process has a variance term $\operatorname{Var}(\tilde{\boldsymbol{h}}^{(l)}) = \frac{D \cdot scale^2}{6}$ (D is the dimension of hidden layers), even though the expected dequantized message is unbiased $\mathbb{E}\left[\tilde{\boldsymbol{h}}^{(l)}\right] = \mathbb{E}[\operatorname{Deq}(\operatorname{Q}(\boldsymbol{h}^{(l)}))] = \boldsymbol{h}^{(l)}$. Therefore, to address the introduced aggregation error, intuitively we can apply node-aware quantization to improve weight update accuracy.

3.4.1 Graph Extractor

To deploy optimizations aiming at specific GNN settings, *Graph Extractor* learns and analyzes the graph structure and node properties for dynamically adjusting the quantization level of each node. This means that even within a single round, each boundary node can be assigned a different quantization bit-width b. It first collects the structure information of input graphs and analyzes the importance of each node, then allocates higher bit-width quantization for more important nodes. In this way, SYLVIE can encourage more accurate embeddings and gradients by protecting important nodes from excessive quantization.

Data-adaptive Quantization. To encourage more accurate embeddings and weight updates, SYLVIE protects nodes with higher in-degree values as we find high in-degree nodes contribute most towards errors in weight updates. For undirected graphs, we protect nodes with high degrees. Specifically, before training, we pre-process the input graph and construct an importance factor $p(0 \sim 1)$ for each boundary node to

denote its probability of introducing quantization errors to embeddings and gradients. A higher importance factor means a higher probability of causing errors. The importance factor is higher if the node's in-degree is large and nodes with the same in-degree also have the same importance. Boundary nodes with the maximum in-degree are assigned to $p = 1$ and the important factors of other nodes are calculated by interpolating between 0 and 1 based on their in-degree ranking in the boundary nodes pool. We re-order the tensor of in-degree values and match them with evenly distributed percentiles. After this, we generate an importance-aware node mask to map nodes to their corresponding quantization bit-width. This node mask is later combined with the time-adaptive part (illustrated in §3.5.1) to jointly decide the assigned bit-width for boundary nodes. Nodes with the same mask level are quantized in the same bit-width for communication of both embeddings and embedding gradients. In this way, Sylvie lowers communication overhead while encouraging more accurate messages to flow back to weights via high in-degree nodes.

3.5 Online Stage of Sylvie

After exploiting the input-level information, Sylvie further monitors the training status on the fly and makes optimizations dynamically tailored to GNN training. Sylvie incorporates an online stage consisting of a lightweight *Quant Orchestrator* for automatic quantization decisions and *Pipeline Adaptor* for adjusting the training mode to greatly accelerate the training while balancing the trade-off between training efficiency and model convergence. It also has a *Coordinator* for analyzing the training feedback and dynamically deploying the optimizations to execution.

3.5.1 Quant Orchestrator

Sylvie explores the opportunity of quantization to reduce the substantial communication in GNNs for training acceleration. It integrates a novel *Quant Orchestrator* to balance the efficiency-accuracy trade-off for distributed GNN training on GPUs. It is computationally lightweight and effective in boosting training and empirically keeping the model quality. In detail, it jointly orchestrates the quantization from two dimensions to minimize the communication, namely data and time dimensions. The first dimension lies in graph's node level as discussed in §3.4.1. The second dimension lies in the GNN training time, where we identify that different training epochs can use different quantization bit-widths to reach the efficiency-convergence balance.

Convergence versus Bit-width. From Equation 3.1 and the variance term $\mathrm{Var}(\tilde{\boldsymbol{h}}^{(l)})$, we can see that changing the bit-width b leads to a trade-off between the total communication volume and the variance value. With smaller b, the communication cost

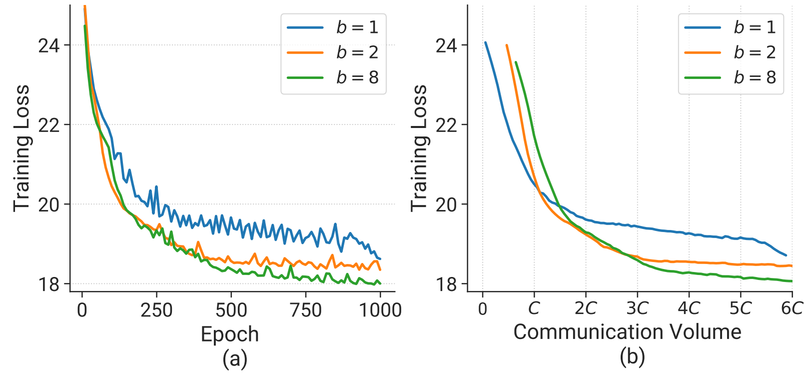


FIGURE 3.5: Training loss with respect to different quantization bit-widths on GAT (Yelp dataset).

decreases while the variance increases. Building on this, we empirically analyze the effect of b on the model convergence over time and use it to design a strategy to accustom b during the training course. The motivation behind the adoption of time-adaptive quantization during training to minimize the communication can be understood from Figure 3.5. We can see from Figure 3.5(a) that a smaller b,i.e., coarser quantization, results in worse convergence of training loss versus training time. We also plot the training loss with respect to the communication volume in Figure 3.5(b), where C is the unit communication volume we use to plot loss values and equals 5GB here. It reveals that a smaller b enables to perform more epochs for the same communication volume and achieves higher convergence speed in early training stages, which is also pointed out in [120] on a similar problem. Based on this observation, intuitively we can start from the smallest bit-width b along the time dimension and dynamically adjust it according to the training status to balance the training efficiency and convergence. Next, we will introduce the designed metrics to formalize the optimization problem.

Time-adaptive Quantization. Intuitively, time-adaptive quantization changes the quantization bit-width b_t along epoch t during training. *Quant Orchestrator* chooses b_t to minimize the communication overhead without sacrificing the model accuracy as much as possible. Some works [120, 121] use similar observations but only monotonically increase the quantization level. Differently, *Quant Orchestrator* monitors both the training loss (to measure the model convergence) and throughput (to measure the training efficiency) to adjust b_t in a nonmonotonic way, which boosts the training as much as possible while not sacrificing the model convergence significantly. At the end of epoch t, *Coordinator* takes the training outputs, and the global loss L_t of N partitions is estimated using the local losses $L_t = \frac{\sum_{n=1}^{N} L_t^n}{N}$. To better estimate the convergence, we track a running average loss $F_t = \lambda F_{t-1} + (1-\lambda)L_t$. To integrate the consideration of training throughput, a Loss Descent Rate (LDR) tailored to GNN training is measured as $LDR_t = \frac{F_t - F_{t-1}}{et_t}$, where et_t is the t-th epoch training time.

According to the aforementioned analysis, at the beginning b_0 is initialized as b_{min} from the bit-width set $\{1, 2, 4, 8\}$. When $LDR_t \geq LDR_{t-\delta}$ for some $\delta \in \mathbb{N}$ which specifies the range of epochs for LDR comparisons, SYLVIE heuristically determines the current bit-width suffices to reduce the loss. Then *Quant Orchestrator* interacts with *Coordinator* to decrease b to gain higher speed. On the condition of this well-balanced training, *Quant Orchestrator* adapts $b_{t+1} = \frac{b_t}{2}$ for higher throughputs. On the other hand, $LDR_t < LDR_{t-\delta}$ in δ epochs denotes the training is about to converge or too many errors are introduced by quantization. The partly trained model requires higher precision to get further improved. In this case, *Quant Orchestrator* increases the bit-width $b_{t+1} = 2b_t$ to reach lower errors and enable more stable and accurate training. The above time-adaptive quantization process is summarized as the following equation:

$$b_{t+1} = \begin{cases} b_{\min} & t = 0 \\ 2b_t & LDR_t < LDR_{t-\delta} \cap t > \delta \cap b_t < b_{\max} \\ b_t/2 & LDR_t \geq LDR_{t-\delta} \cap t > \delta \cap b_t > b_{\min} \\ b_t & \text{else} \end{cases} \tag{3.4}$$

The time-adaptive quantization on communication messages ensures models sensitive to noise quickly reach a sufficiently high bit-width and those not sensitive to noise get accelerated as much as possible. It empirically achieves a good trade-off between the training convergence and efficiency.

Joint Orchestration. As shown in Figure 3.4, *Quant Orchestrator* combines the data-adaptive (§3.4.1) and time-adaptive quantization to meticulously facilitate training. As illustrated previously, the data-adaptive part constructs an importance-aware node mask in the offline stage. At each epoch t during training, the time-adaptive part first determines a base bit-width b_t. Then the data-adaptive part uses the node mask to further adjust the node-wise bit-widths $b_t^v, v \in V_{\text{boundary}}$ on the basis of b_t. Figure 3.6 gives an example of the detailed process of how the time-adaptive part, data-adaptive part and *Quant Orchestrator* adjust the bit-widths. In Figure 3.6(a), for partition-1, the data-adaptive quantization assigns small bit-widths (e.g., $b = 1$) to less important nodes (1 and 5) and large bit-widths ($b = 8$) to more important nodes. On the other hand, time-adaptive quantization in Figure 3.6(b) alters the bit-widths for all nodes across training epochs, e.g., it increases from 1 to 2 at epoch $t+1$. In Figure 3.6(c), *Quant Orchestrator* applies the data-adaptive part on the basis of time-varying bit-widths. For instance, at t-epoch the time-adaptive part determines a preliminary $b_t = 1$, then the candidate node-wise bit-widths are $\{1, 2, 4, 8\}$. However, the base bit-width is 8 at epoch $t + 5$, so all nodes will be assigned with $b_t^v = 8$ regardless of their importance.

Node ID	1	2	3	4	5	6 ···
Partition	Assigned Bit-width					
1	**1**	**4**	**2**	**8**	**1**	**2**

(a) Data-adaptive Quantization

Epoch	t	t+1	t+2	t+3	t+4	t+5 ···
Partition	Assigned Bit-width					
1	**1**	**2**	**1**	**2**	**4**	**8**

(b) Time-adaptive Quantization

Epoch	t	t+1	t+2	t+3	t+4	t+5 ···
Node ID	Assigned Bit-width					
1	**1**	**2**	**1**	**2**	**4**	**8**
2	**4**	**8**	**4**	**8**	**8**	**8**
3	**2**	**4**	**2**	**4**	**4**	**8**
4	**8**	**8**	**4**	**8**	**8**	**8**
5 ···	**1**	**2**	**1**	**2**	**4**	**8**

(c) Quant Orchestrator

FIGURE 3.6: The process of how *Quant Orchestrator* jointly orchestrates the quantization from time- and data-dimension.

3.5.2 Pipeline Adaptor

In the former parts, *Quant Orchestrator* enhances the efficient training of GNNs by reducing the communication volume from two dimensions. However, there exist some large-scale distributed GNN training jobs, where communication still occupies a large portion of the training time. Additionally, the asynchronous training [114, 115, 122] is usually adopted in distributed DNN training to enhance the algorithm efficiency. However, some frameworks like [115] are based on a centralized compute topology with workers running asynchronously to hide partial communication of **weights** and **weight gradients** to the parameter server, suffering from completely stale weight gradients. Pipe-SGD [114] proposes a decentralized learning framework pipelining the local training iterations to hide the communication of **weight gradients**. Nonetheless, all these works target on large models, where the main communication overhead comes from the communication of **weights/weight gradients** other than the **embeddings/embedding gradients** in distributed GNN training (as introduced in §3.2.2). Moreover, different from the staleness of **all** weights/weight gradients in asynchronous distributed DNN training, the staleness in our case incurs only in **partial** embeddings/embedding gradients.

In the online stage, inspired by [56], SYLVIE designs *Pipeline Adaptor* which leverages the pipeline of layer-wise communication and computation across two adjacent epochs

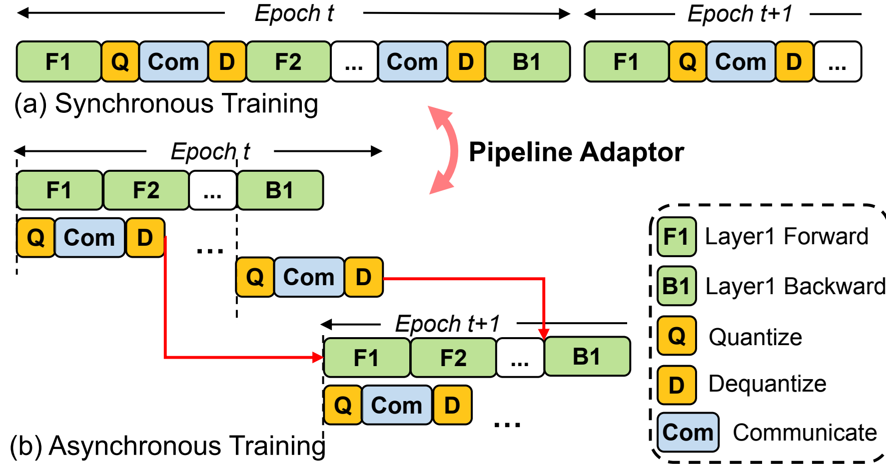


FIGURE 3.7: Illustration of how *Pipeline Adaptor* adjusts execution mode between synchronous and asynchronous training.

to further hide all the latency (quantization/dequantization operations and reduced communication duration). Furthermore, because asynchronization inevitably leads to stale messages, *Pipeline Adaptor* automatically adapts training between the synchronous and asynchronous settings to bound the number of delay epochs for staleness control as shown in Figure 3.7. The asynchronous pipeline is perfectly suitable for our case due to one unique feature: the quantization and dequantization operations perform simple linear mappings to message vectors, which are low-overhead and thus can be easily parallelized. To better illustrate how *Pipeline Adaptor* works, we first introduce the vanilla synchronous training in Figure 3.7(a). After each layer's computation, the intermediate activations of boundary nodes are quantized and transferred during both forward and backward passes between all workers using all-to-all communication [52]. The subsequent computation cannot begin until the worker receives and dequantizes the messages. Thus each worker is blocked from computation and cannot continuously utilize the GPU.

In Figure 3.7(b), to realize the inter-epoch pipeline, each layer's computation directly begins with the latest updated messages in this worker. In parallel, messages are quantized and communicated concurrently. To realize the asynchronous training, we wrap the GPU kernels of inner nodes' computation and pipelined operations (quantization, dequantization and communication) with independent CUDA streams. Note that the communicated boundary messages at epoch t will be used for computation at epoch $t+1$, leading to a compound usage of the latest inner nodes' messages and stale boundary nodes' messages. To mitigate the effects on the convergence of the partial staleness, *Pipeline Adaptor* performs compulsory synchronization of the latest messages for staleness control, reaching a good trade-off between the training throughput and convergence rate. To achieve this, *Pipeline Adaptor* also monitors *LDR* introduced in §3.5.1 to evaluate the training status. In detail, it determines convergence is

downgraded by staleness when $LDR_t < LDR_{t-\delta}$ and informs *Coordinator* to perform synchronous training at epoch $t+1$. Otherwise, the training stays in the asynchronous mode.

3.5.3 Coordinator

In the online stage, *Coordinator* retrieves and analyzes the training outputs. According to the feedback, it then interacts with *Quant Orchestrator* and *Pipeline Adaptor* to coordinate the optimizations on training jointly. Past works [69, 112] prove the convergence of GNNs with quantization as long as the quantization is unbiased and has bounded variance, which has been claimed in §3.2.2. In addition, SYLVIE only applies quantization to partial messages (messages of boundary nodes), which also limits the introduced variance. Similar methods can be found in existing works [72, 123, 124] which adopt the subset quantization. In addition, [56] demonstrates the convergence of distributed GNN training under the asynchronous setting and the convergence rate is even better than sampling-based methods. These convergence results can extend to SYLVIE and we refer to the detailed analysis from them.

To realize communicating messages quantized with different bit-widths between workers, each worker first prepares multiple buffers for sending and receiving messages, whose sizes are determined by the assigned bit-widths of *Quant Orchestrator*. We group messages based on their assigned bit-widths, perform static quantization to each group and then concatenate them into a compressed single tensor for transferring. After communicating with the help of buffers, the compressed tensors are decoded back to full-precision messages based on a node-matching table generated by *Quant Orchestrator*. Those recovered messages then participate in the subsequent computations.

3.6 Evaluation

We implement SYLVIE atop DGL 0.9 [105] and PyTorch 1.10 [117]. The communication process is implemented via `torch.distributed` in the ring all2all pattern [52]. For graph partitioning, we use the widely-adopted METIS [58] partition algorithm whose objective is set to minimize the communication volume.

Datasets and Models. We evaluate SYLVIE on five real-world large-scale graph benchmarks: Reddit [3], Yelp [47], Ogbn-products [1], Amazon [2], and Ogbn-papers100M [1]. The detailed information is shown in Table 3.4. Reddit is a post-to-post graph.

Table 3.4: Detailed information of datasets used in evaluation.

Datasets	# Nodes	# Edges	Features Dim.	# Classes
Reddit	232,965	114,615,892	602	41
Yelp	716,847	6,977,410	300	100
Ogbn-products	2,449,029	61,859,140	100	47
Amazon	1,598,960	132,169,734	200	107
Ogbn-papers100M	111,059,956	1,615,685,872	128	172

Yelp categorizes the types of business based on users' reviews and relationships. Ogbn-products classifies Amazon products according to customers' reviews. Amazon predicts product categories using their properties and relations between them. We choose versatile GNN models commonly adopted in GNN applications for evaluation, including two shallow models GraphSAGE [3] and vanilla GCN [4], deep models GCNII [125], DGANN [34], SGC [32] and JKNet [31], and special GNN model GAT [24] (the number of heads is set to 1). Not that JKNet is only applicable to Reddit dataset and DAGNN is unsuitable to be deployed on Yelp dataset. Regarding the models, we follow the hyperparameter configurations reported in the original papers as closely as possible. The detailed model hyperparameters used for evaluation are presented in Table 3.5. For JKNet, the number of layers is 8 and hidden size is 128. The training epoch equals to 800 and the dropout rate is 0.5. The adopted learning rate is 0.01 and other hyperparameter configurations are also listed in Table 3.5. The optimizer is Adam [126] for all datasets and models and we use the default hyperparameters for Adam optimizer. All methods terminate after a fixed number of epochs which is sufficient for all models to converge.

Baselines. For the baselines, we compare Sylvie with four SOTA-distributed full-graph training methods: (1) DGL [105]: the vanilla distributed GNN training on top of the latest DGL 0.9; (2) SAR [64]; (3) PipeGCN [56]; (4) BNS-GCN [52]: the p value is set to 0.1 as suggested by the paper. Baselines are orthogonal to each other in distributed GNN system designs so that we can make a fair comparison. Note that all the baselines do not implement deeper GNNs originally, so we modified deeper GNNs on them ourselves and only show their results on their respective supported GNNs. We do not compare with ROC [51] and CAGNET [66] here since their performance are far inferior to DGL, which now is the default choice for most users. In addition, CAGNET is deployed on OLCF Summit which we do not have access on.

Testbeds. Our experiments are performed on two different GPU servers. ❶Severs each with 8 RTX 3090 GPUs (24GB), intra-server connection (CPU-GPU and GPU-GPU) based on PCIe 4.0 lanes and inter-server connection via 1Gbps Ethernet. ❷Servers each with 8 A100 GPUs (80GB) with NVLink and 200Gbps InfiniBand.

TABLE 3.5: Model architecture and detailed hyperparameters. ***Arch.***: Number of layers × Number of hidden neurons in each layer. ***HP.***: (Epoch, Dropout)

Model	**Config**	**Dataset**			
		Reddit	**Yelp**	**Ogbn-products**	**Amazon**
GraphSAGE	*Arch.* *HP.*	4 ×256 (2000, 0.5)	4×512 (2000, 0.1)	3×128 (500, 0.3)	4×128 (2000, 0.1)
GCN	*Arch.* *HP.*	4×256 (2000, 0.5)	4×512 (2000, 0.1)	3×128 (500, 0.3)	4×128 (2000, 0.1)
GCNII	*Arch.* *HP.*	8×256 (1000, 0.5)	8×512 (1000, 0.5)	8×128 (500, 0.5)	6×128 (2000, 0.5)
DAGNN	*Arch.* *HP.*	8×256 (1000, 0.8)	-	8×128 (500, 0.8)	6×256 (1000, 0.5)
SGC	*Arch.* *HP.*	8×256 (1000, 0.1)	8×512 (1000, 0)	8×128 (500, 0)	6×256 (500, 0)
GAT	*Arch.* *HP.*	2×256 (200, 0.5)	2×256 (1000, 0.1)	3×128 (200, 0.3)	3×128 (1000, 0.1)

3.6.1 End-to-end Experiments

We compare the end-to-end performance of SYLVIE with baselines on both RTX 3090 and A100 servers.

Training Speedup and Accuracy Maintenance. Table 3.6 and Figure 3.8 describe throughput and test accuracy comparisons between SYLVIE and SOTA baselines on versatile GNN models over two 3090 servers. Here throughput is defined as the number of epochs run per second, and we normalize the throughput of each method on base of DGL. In each training task, we treat the first 10 epochs as the warmup stage and only record statistics afterward. We can clearly see that SYLVIE substantially outperforms other methods by a large margin on each dataset and model. Specifically, SYLVIE achieves a marvelous throughput improvement of 8.67∼16.03× over DGL and far exceeds SAR and PipeGCN. Among the baselines, SAR shows the lowest throughput since it does not cope with the communication overhead, and its computation burden even increases due to the rematerialization. SYLVIE also delivers 1.03∼1.86× larger throughput than BNS-GCN. We note that PipeGCN does not show significant performance since in the multi-server training, the communication cost is immensely larger than computation and could hardly be hidden.

To further unfold the effectiveness of SYLVIE in distributed setting, we also conduct evaluations on A100 servers with NVLink and 200Gbps InfiniBand, as shown in Table 3.7. SYLVIE still shows impressive acceleration and outperforms baselines on such frontier equipment. For the largest dataset Ogbn-papers100M, we partition it to 32 parts and deploy the training on 4 servers (each 8 GPUs). We can see even at such

TABLE 3.6: Detailed comparison of training throughput and test accuracy between SYLVIE and other baselines when training on two 3090 servers, where the best performance is highlighted in bold. Dash line '-' means the method does not converge. SYLVIE always outperforms others in throughput on all the models and datasets while still achieving high accuracy.

			Reddit		Yelp		Ogbn-products		Amazon	
	Model	Method	Thr.	Test Acc.(%)	Thr.	F1-micro(%)	Thr.	Test Acc.(%)	Thr.	Test Acc.(%)
Shallow	GraphSAGE	DGL	1.00×	97.10±0.01	1.00×	65.07±0.19	1.00×	79.19±0.15	1.00×	**81.29**±0.02
		SAR	0.42×	96.02±0.12	0.37×	60.51±0.09	0.64×	74.42±0.07	0.43×	78.85±0.07
		PipeGCN	1.15×	97.02±0.11	1.15×	65.14±0.08	1.19×	**79.29**±0.05	1.05×	81.27±0.08
		BNS-GCN	9.02×	**97.14**±0.01	8.11×	**65.22**±0.23	8.38×	79.11±0.11	9.08×	80.90±0.05
		SYLVIE	**14.64×**	96.87±0.03	**11.27×**	64.92±0.38	**15.74×**	78.85±0.26	**13.70×**	81.24±0.11
	GCN	DGL	1.00×	94.84±0.58	1.00×	47.50±0.07	1.00×	73.70±0.20	1.00×	**56.59**±0.11
		SAR	0.42×	**95.34**±0.17	0.38×	47.00±0.12	0.65×	70.13±0.10	0.43×	53.08±0.07
		PipeGCN	1.15×	94.69±0.56	1.16×	47.16±0.01	1.20×	**74.04**±0.23	1.01×	56.56±0.34
		BNS-GCN	9.18×	95.00±0.33	8.40×	47.27±0.37	8.64×	73.54±0.42	9.34×	56.47±0.60
		SYLVIE	**15.15×**	95.31±0.01	**13.13×**	**47.62**±0.30	**16.03×**	73.78±0.19	**14.61×**	56.07±0.21
Deep	GCNII	DGL	1.00×	**89.53**±0.20	1.00×	61.55±0.08	1.00×	**58.34**±0.16	1.00×	42.15±0.21
		PipeGCN	1.14×	84.08±0.32	1.13×	60.18±0.21	1.20×	56.78±0.11	1.03×	41.47±0.18
		BNS-GCN	-	-	-	-	-	-	-	-
		SYLVIE	**17.18×**	89.16±0.11	**12.48×**	**62.43**±0.07	**10.60×**	58.15±0.07	**10.42×**	**43.25**±0.11
	DAGNN	DGL	1.00×	**91.94**±0.20			1.00×	63.22±0.14	1.00×	54.01±0.14
		PipeGCN	-	-	-	-	1.18×	60.32±0.22	1.03×	52.83±0.31
		BNS-GCN	-	-			-	-	-	-
		SYLVIE	**7.88×**	91.89±0.13			**10.06×**	**63.41**±0.12	**12.47×**	**54.91**±0.18
	SGC	DGL	1.00×	80.64±0.19	1.00×	50.30±0.05	1.00×	54.76±0.20	1.00×	**41.12**±0.05
		PipeGCN	1.02×	80.03±0.37	1.12×	49.31±0.12	1.07×	54.08±0.29	1.05×	39.12±0.17
		BNS-GCN	-	-	-	-	-	-	-	-
		SYLVIE	**7.56×**	**80.68**±0.10	**13.46×**	**50.32**±0.07	**12.12×**	**55.02**±0.11	**13.22×**	41.11±0.14
Special	GAT	DGL	1.00×	**93.97**±0.60	1.00×	**44.39**±0.16	1.00×	78.14±0.12	1.00×	**42.84**±0.96
		SAR	0.25×	91.47±0.08	0.21×	44.30±0.11	0.27×	76.40±0.06	0.21×	42.48±0.07
		PipeGCN	1.14×	93.85±0.64	1.15×	43.75±0.23	1.19×	77.03±0.11	1.04×	42.37±0.07
		BNS-GCN	7.86×	89.08±0.63	8.11×	43.66±0.24	8.08×	74.07±0.92	8.43×	40.67±0.79
		SYLVIE	**12.26×**	93.40±0.62	**13.48×**	44.15±0.63	**13.21×**	**78.38**±0.18	**8.67×**	42.08±0.25

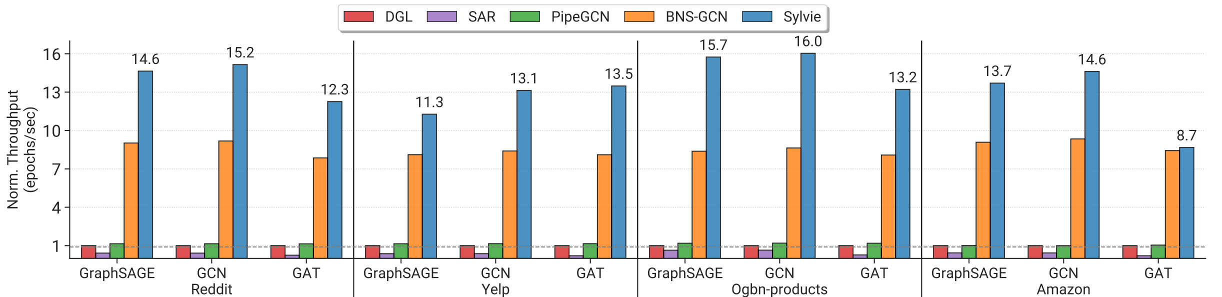


FIGURE 3.8: Training throughput of different methods (normalized to that of DGL, shown in the dashed line) when training three representative models on four datasets on two 3090 servers. SYLVIE outperforms DGL by up to 16.0×.

a large-scale setting where communication overhead dominates, SYLVIE still provides the largest speedup and substantially reduces the communication time by 95%.

Generality on Versatile GNNs. Unlike other baselines, SYLVIE consistently performs well in efficiency and model accuracy on deeper and special structured GNNs. In Table 3.6, SYLVIE always achieves far better training throughput than other methods on all types of GNNs. Especially, SYLVIE successfully converges and maintains model accuracy on deeper and special GNNs, and even reaches higher accuracy in some cases, e.g., enables DAGNN to reach 63.41% on Ogbn-products while achieving

TABLE 3.7: Training epoch time comparison between SYLVIE and other methods on GraphSAGE on A100 servers with NVLink.

Dataset	Server Setting	Method	Epoch Time (s)	Comm. (s)
Ogbn-products	2 Servers 16 GPUs	DGL	0.99 (1.00×)	0.87
		PipeGCN	0.73 (1.36×)	0.57
		BNS-GCN	0.39 (2.54×)	0.17
		SYLVIE	**0.23 (4.30×)**	0.11
Ogbn-papers100M	4 Servers 32 GPUs	DGL	17.00 (1.00×)	14.00
		PipeGCN	12.40 (1.37×)	9.70
		BNS-GCN	2.10 (8.10×)	1.47
		SYLVIE	**1.30 (13.08×)**	0.69

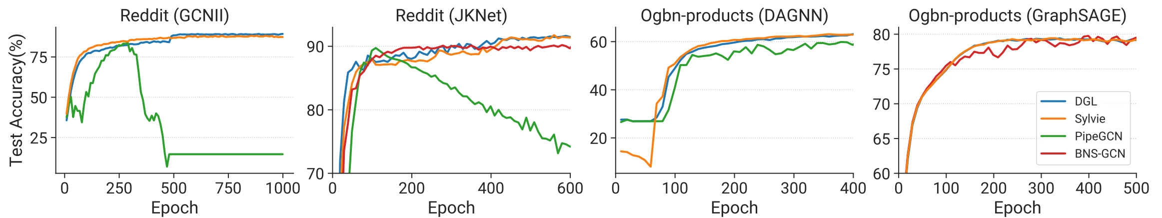


FIGURE 3.9: The convergence curve comparisons of SYLVIE and baselines on different models and datasets over single-server.

the largest throughput 10.06×. On the contrary, current systems fail to accommodate to deeper and special GNNs. For instance, BNS-GCN cannot converge on deeper GNNs at all due to the excessive node dependency loss along layers. Additionally, it incurs a significant accuracy loss of up to 4.9% on GAT, showing its limited generality to other models. PipeGCN also suffers from serious accuracy drop up to 5.45% since the staleness errors accumulate essentially when the model is deep. Via the adaptive optimizations by monitoring training status, SYLVIE is robust to the noise introduced by compressed activations, indicating SYLVIE enables to train deeper and more complex GNNs on large graphs with minimal loss in performance.

Maintaining Model Convergence. We examine the convergence curves of SYLVIE on various models in Figure 3.9. We can see the curves of SYLVIE are almost identical to that of the original DGL version and converge to high accuracy, verifying SYLVIE preserves model quality well. However, other methods either converge to low accuracy (BNS-GCN) or lead to slower convergence and even occur over-fitting (PipeGCN on GCNII and JKNet respectively). The over-fitting is mainly due to PipeGCN's smoothing method, which increases stability on the training set. It constrains the model from exploring a more general minimum point on the test set, thus leading to overfitting on deeper models. These results are also consistent with the methods' theoretical analysis. SYLVIE greatly maintains the convergence thanks to the 3D-adaptive adjustments according to the monitored training status.

TABLE 3.8: Throughput when training on a single 3090 server.

Model	Method	Dataset		
		Yelp	**Ogbn-products**	**Amazon**
GraphSAGE (N=8)	DGL	1.00×(1.82 ep./s)	1.00×(0.95 ep./s)	1.00×(0.38 ep./s)
	SAR	0.99×	1.27×	1.12×
	PipeGCN	1.08×	1.05×	0.97×
	BNS-GCN	3.10×	3.07×	6.93×
	SYLVIE	**4.02**×	**4.40**×	**7.78**×
GCN (N=8)	DGL	1.00×(1.91 ep./s)	1.00×(1.12 ep./s)	1.00×(0.42 ep./s)
	SAR	1.04×	1.32×	1.23×
	PipeGCN	1.07×	1.06×	0.94×
	BNS-GCN	2.30×	2.46×	4.78×
	SYLVIE	**4.36**×	**3.44**×	**5.04**×

Performance on Single Server. We also test the performance of SYLVIE on a single 3090 server in Table 3.8. SYLVIE still outperforms other methods in training throughput, with a maximum of 7.78× speedup when training GraphSAGE. Table 3.8 shows partial results of the throughput on different methods due to the page limit. The speedup is relatively less significant compared to the multi-server setting, which involves more partitions with larger communication overhead.

3.6.2 Ablation Studies

To verify the effectiveness of our 3D-adaptive scheme and explore the impact of each system module explicitly, we compare SYLVIE with different static settings. All ablation studies are conducted on A100 servers.

Quantization Ablation Study. The evaluations consider different static values of b, from no quantization to 1-bit quantization as shown in Table 3.9. We fix the execution mode to always-synchronous training to make fair comparisons since the adaptive pipeline adjustment is unpredictable in each training. We can see with the decrease of b value, training epoch time also decreases, but with greater accuracy loss. This is because applying low-bit quantization introduces significant variance and degrades accuracy. However, *Quant Orchestrator* enables SYLVIE to gain high throughput (2.3× compared with 32-bit) and maintain robust accuracy (65.0% vs 64.4% of 1-bit). This verifies simply performing static quantization cannot maximize its benefits or keep model quality.

Pipeline Ablation Study. Here we compare SYLVIE with an always-synchronous and always-asynchronous version. Similarly, we fix b values to make fair comparisons and provide the results in Table 3.10. We observe that SYLVIE has a larger training speed than always-synchronous version and higher accuracy than always-asynchronous

TABLE 3.9: Training GraphSAGE on Yelp with different b values and fixed execution on single A100 server.

b	32	8	4	2	1	SYLVIE (adaptive quant)
Epoch Time (s)	0.90	0.52	0.37	0.28	0.22	0.39
Accuracy (%)	65.3	65.3	65.1	64.5	64.4	65.0

TABLE 3.10: Training GraphSAGE on Yelp with different execution modes and fixed b values on single A100 server.

Method	Fix b=32		Fix b=1	
	Epoch Time (s)	Acc. (%)	Epoch Time (s)	Acc. (%)
Always-sync.	0.90	65.3	0.21	64.4
Always-async.	0.75	64.6	0.12	64.2
SYLVIE (adaptive pipeline)	0.81	64.9	0.17	64.6

version with comparable speed, which validates *Pipeline Adaptor* successfully strikes efficiency-accuracy trade-off.

Trained on the same model and dataset, the epoch time of 3D-adaptive SYLVIE is 0.27s and accuracy is 65.0%. Together with both ablation studies, we can see SYLVIE combines the best of all worlds from efficiency and accuracy. By dynamically adjusting the system optimizations guided by training status, the 3D-adaptive scheme greatly boosts training while bounding the gradient variance to a limited level, thus reaching a better efficiency-accuracy balance.

3.6.3 More Evaluation

Communication Volume and Time. To demonstrate the training speedup is due to the reduced communication, we record the actual communication volume per epoch and training time breakdown in Table 3.11. We observe that SYLVIE cuts down the communication volume dramatically. For example, there are originally 5632.6 MB of communication per epoch for the Amazon dataset. After deploying SYLVIE, there are only 254.7 MB communicated messages, reducing almost 22× communication volume. Accordingly, the communication time is vastly shortened (from 11.47s to 0.81s).

Sensitivity Analysis. The introduced hyper-parameter δ and λ determine the performance and overhead of SYLVIE. Here we perform sensitivity experiments on δ. As shown in Figure 3.10, the faster convergence and higher accuracy are obtained when smaller δ value is adopted, but coming with possibly lower throughput (here for GraphSAGE, 4.98 epochs/s with $\delta = 5$ vs. 5.93 epochs/s with $\delta = 10$). Seriously

TABLE 3.11: Epoch communication volume and time breakdown of training GraphSAGE over two servers.

	Method	Comm. Volume(MB)		Per-epoch Time (s)	
		Main Data	Scales	Total	Comm.
Reddit	DGL	2791.7	0	7.28	6.62
	SYLVIE	126.9	15.6	0.5	0.44
Amazon	DGL	5632.6	0	13.33	11.47
	SYLVIE	254.7	30.4	0.97	0.81

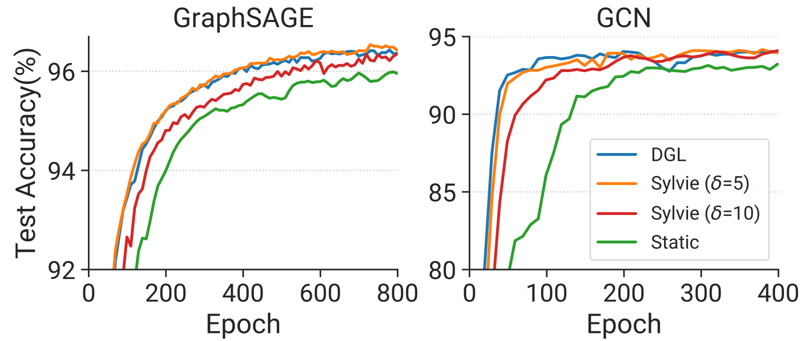


FIGURE 3.10: Sensitivity experiments of comparing range δ on Reddit (N=8, single server).

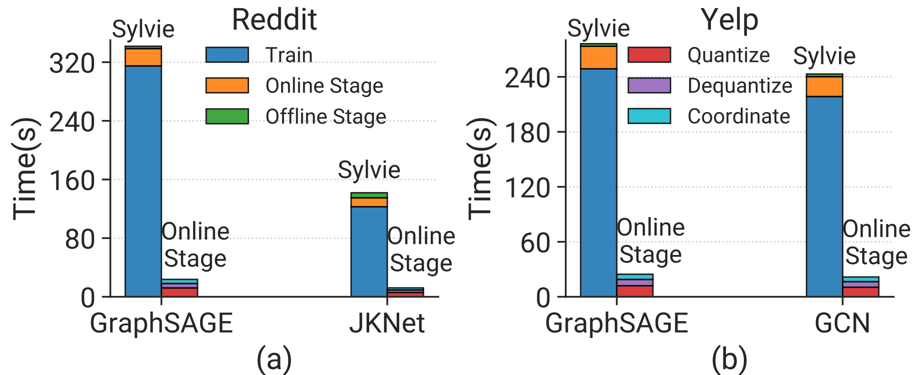


FIGURE 3.11: Wall-clock time of DGL and SYLVIE with overhead.

chasing the lowest quantization variance ($\delta = 1$) or just caring about the highest training throughput (very large δ) is not the best choice to fully utilize the benefits of quantization and pipelining. Choosing different values always exists a trade-off between efficiency and accuracy. Currently, we suggest $\lambda = 0.9$ for better convergence, and users can select $\delta = 5$ for higher model quality or larger $\delta = 20$ for faster speed. How to choose the values wisely can be leaved for further investigation.

System Overhead Analysis. To understand how much extra overhead brought by SYLVIE, we record the time breakdown from two levels: wall-clock time level in Figure 3.11 and more fine-grained epoch time portions in Figure 3.12.

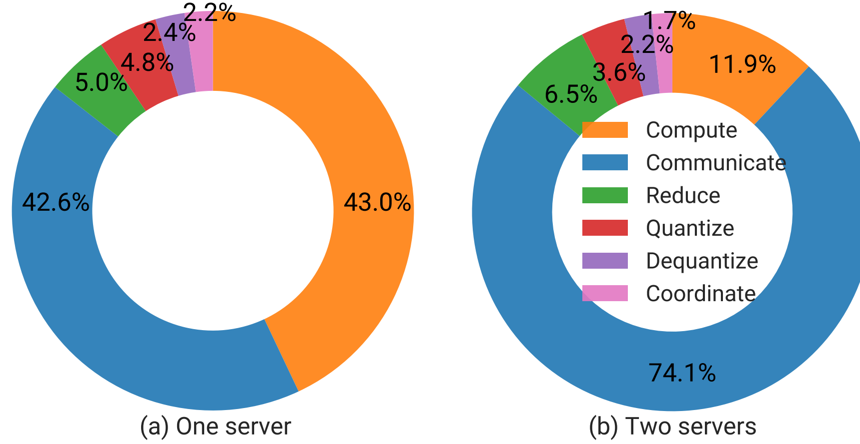


FIGURE 3.12: Ratios of different components in epoch time when training GraphSAGE with SYLVIE on Reddit over single server and two servers. Both quantization and coordination take up negligible overhead.

The wall-clock time is split into three coarse-grained parts (offline stage time, online stage time and actual training time) in the left bar for each model, and the online stage is further split into three parts (quantization, dequantization and coordination time) in the right bar. We can observe both the online and offline overhead are negligible compared with the training time reduction. The wall-clock time of SYLVIE on GraphSAGE-Reddit in Figure 3.11 is 314.9s, 23.7s and 3.1s for training, online stage and offline stage respectively. The overhead proportion (online and offline stage) in total training time is only 7.8%. Similar conclusions can be obtained from Figure 3.12, where we record the per-epoch time on a single server and two servers. Both cases demonstrate the time consumed by quantization and coordination occupies the smallest portions, indicating the negligible overhead brought by SYLVIE. Coordination accounts for the least time, even lower than the quantization-related operations.

3.6.4 Scalability Analysis on More Servers

To further evaluate SYLVIE's capability, we scale up the training over multiple servers on both server types. Table 3.12's results on A100 server show SYLVIE still obtains considerable speedup on high-speed network servers and shows not bad scalability. The speedup rate increases even on more servers, e.g., 2.8∼4.3∼4.9×. Figure 3.13 presents the normalized training throughput of SYLVIE over 3090 servers. We also observe that SYLVIE maintains great performance and even achieves a higher throughput acceleration ratio when the number of servers increases. On both settings, SYLVIE offers the best training speedup compared with other methods, while SAR and PipeGCN show very limited performance in large-scale training. In a nutshell, SYLVIE can deliver desired performance for larger-scale training scenarios.

TABLE 3.12: Epoch time of GraphSAGE on Ogbn-products under different A100 server settings.

Server Setting	Method	Epoch Time (s)	Comm. Time (s)	Comp. Time (s)
1 Server, 8 GPUs	DGL	0.83	0.71	0.09
	SYLVIE	**0.30 (2.8×)**	0.13	0.09
2 Servers, 16 GPUs	DGL	0.99	0.87	0.04
	SYLVIE	**0.23 (4.3×)**	0.11	0.04
3 Servers, 24 GPUs	DGL	1.23	1.13	0.03
	SYLVIE	**0.25 (4.9×)**	0.12	0.03

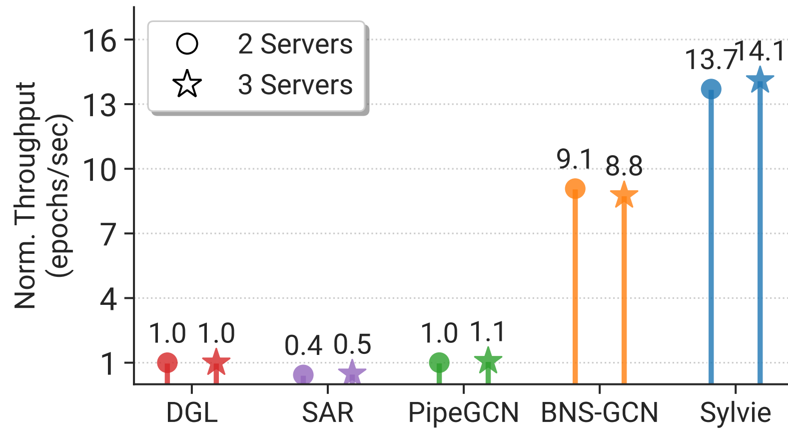


FIGURE 3.13: Normalized training throughput on multiple 3090 servers for GraphSAGE on Amazon.

3.7 Summary

This work proposes SYLVIE, an efficient distributed GNN training framework that enormously boosts training efficiency in a 3D-adaptive way, while maintaining the model quality. Unlike existing methods which fail to accommodate to universal GNN models, SYLVIE outperforms well on all model structures. Extensive experiments show that SYLVIE can substantially boost the training throughput by up to 17.2×.

Chapter 4

TorchGT: Large-scale Graph Transformer Training System

This chapter presents the research[1] to design an accuracy-maintained and compute-efficient system from both algorithm and system perspectives to support large-scale graph transformer training. Existing graph transformer works neither facilitate efficiency by well-designed parallelism from the system perspective nor propose scalable algorithms for universal graph learning tasks.

4.1 Introduction

Graph-structured data has long been prevalent and indispensable in many real-life applications such as social network construction and molecule analysis. Thus, there emerges a specific family of graph learning methods, namely graph neural networks (GNNs) [3, 4, 24]. GNNs have gained giant breakthroughs and exhibit impressive performance in many tasks such as node classification [18, 95–97] and link prediction [24], mainly due to their message passing mechanism [99], which models the inherent properties of graph structures. However, this module in classic GNNs also leads to commonly acknowledged over-smoothing [127], over-squashing [128, 129] and limited expressivity [130] issues.

To address these deficiencies, a latest approach called *graph transformer* shows more promising power in capturing the inter-dependencies among nodes. Graph transformer is built upon the classical Transformer [131] which allows nodes to attend to all other nodes, and integrates multiple graph structure encodings to include important graph properties. Due to the great modeling capability, graph transformer has garnered surging interest in recent years and a large number of models have been proposed

[1] The work in this chapter has been published in [19].

TABLE 4.1: Graph transformers outperform classical GNNs on graph-level and node-level (Flickr) tasks.

Model		ZINC Test MAE↓	PCQM4M-LSC Validate MAE↓	Flickr Test Acc.(%)↑
Traditional GNNs	GAT	0.384	-	54.29
	GCN	0.367	0.169	61.49
Graph Transformers	GT	0.226	0.141	**68.59**
	Graphormer	**0.122**	**0.123**	66.16

[5, 6, 25, 36]. Existing graph transformers mainly operate by treating graph nodes as input tokens and constructing an input sequence consisting of all the graph nodes. Besides, graph structure encoders are designed as a graph adaptation of the original Transformer architecture. By integrating structural information, graph transformers exhibit competitive performance and outperform traditional message-passing GNNs (e.g., GCN [4] and GAT [24]) on both node classification [25, 36, 39, 43, 75, 76] and graph classification [5, 6, 132] tasks, as shown by Table 4.1. We can obviously see graph transformers obtain the highest scores than GNNs on all tasks.

Real-world graphs can easily involve millions of nodes [1, 2], making the sequence length enormously large. For example, in the current graph transformers' operation way, processing the citation graph dataset ogbn-papers100M from Open Graph Benchmark [1] (including more than 100 million nodes) requires high dimensional inputs with prohibited sequences. Moreover, as illustrated by the profiled results in §4.2.1, the profiled results of training graph transformer models with different sequence lengths also unfold the accuracy benefits of training in long sequence. Therefore, training graph transformers with long sequence is crucial for model quality and the development of versatile graph transformer application scenarios. However, we find most existing graph transformer research works [5, 6, 21, 22] are only limited to small graphs due to a lack of compatible systems tailored for the graph transformer model training with long sequences. More specifically, there are three deficiencies in current works:

First, graph transformers with standard attention have poor scalability to long sequences, due to the computation and memory complexity of $O(N^2)$, quadratic on the number of nodes (N) in a graph [5, 6, 10–13]. Taking fully-connected attention as graph foundation encoders captures the implicit all-pair influence beyond neighboring nodes, but also limits existing graph transformers only on small-graph applications [5, 6, 10–13]. Figure 4.1 demonstrates that even with a state-of-the-art attention library, i.e., FlashAttention [133], the computation of the dense attention mechanism is still a bottleneck during the graph transformer training.

Second, current algorithms either compromise model quality or are only applicable to a single graph learning task. To reduce computation pressure, some graph transformers shorten the input sequences by harnessing neighbor sampling [36, 41] similar adopted in classic GNNs [3, 47, 49]. Others like [21] attempt to overcome the quadratic complexity by replacing full attention with approximate attention methods. However, using sampling methods or simply adopting sparse patterns like [77, 78] loses critical connectivity information and thus sacrifices model precision. On the other hand, some works [25, 75, 76] use self-defined adapted attention modules to reduce memory consumption, but are limited to a specific task, e.g., node classification. They are neither general to versatile graph learning tasks nor portable to be scaled in large-scale training.

Third, no existing works exploit systematic optimizations to realize efficient and scalable training. Several graph transformers [21, 22] apply graph structure to relieve the computation burden. However, this sparse pattern is highly irregular in memory access due to the skewed property of graphs, which is challenging for optimizing the system throughput. Due to the skewed property of graphs, the graph structure-pattern attention is immensely sparse in computation and highly irregular in memory access, which can be very challenging for optimizing the system throughput. Moreover, with large datasets, the memory consumption of model activations grows rapidly, necessitating a scalable system design and memory optimization. But existing works [5, 6, 21, 22, 36] only focus on the implementation of graph transformers in a single GPU, thus limited to very small graphs. Although there has been a breakthrough for large language models (LLMs) by partitioning along the input sequence dimension and training long sequences across devices [14–17], those sequence parallelism ways cannot be directly transplanted on graph transformers due to the extra graph encodings and neglection of structure properties.

To bridge these gaps, we design TorchGT, the ***first*** distributed training system that scales *graph transformer model* to large graphs with billions of edges. Our system abides four design goals: ***scalable***, ***efficient***, ***convergence-maintained***, and ***task-agnostic***. Existing graph transformer works neither facilitate efficiency by well-designed parallelism from the system perspective nor propose scalable algorithms for universal graph learning tasks, thus making it challenging to meet those goals. This hinders the practical development of advanced graph transformer models on real-world graphs. The core design of TorchGT derives from the following three key insights. First, *the learning of graph transformers highly benefits from graph structures.* Specifically, the structure of many real-world graphs is highly sparse [119, 134, 135], which reflects the inherent vertex-vertex interactions. This sparsity could be a guide for how graph transformers attend to nodes to reduce computation costs while maintaining correct connections. In addition, considering the structural property in

the system design also contributes to optimal hardware throughput. Second, *the order of input graph tokens is alterable.* Unlike inputs in famous LLMs like GPT [136] whose token order is crucial for model understanding and generation process[136, 137], graph transformers focus more on connections between nodes. Thus, we can modify the input arrangement to exploit graph properties (e.g., local clusters) for more specialized optimizations. Third, *the block-sparse format is a good match for irregular graph clusters.* Block-sparse formats store data contiguously in memory, reducing storage overheads and memory access. But directly exploiting it on dense attention matrices will drop connectivity and result in substantial accuracy loss [138, 139]. However, by integrating it into our specialized clustered pattern, we find the computation can be further accelerated while maintaining model accuracy. We revisit the graph transformer training from both the algorithm and system perspectives.

As such, our key idea is to design an accuracy-maintained and compute-efficient system from both algorithm and system perspectives to support large-scale graph transformer training. Specifically, TorchGT consists of three key innovations. **Dual-interleaved Attention** is a local-global interleaved attention that integrates graph structural topology into the attention module and selectively combines the global information into the attention with the graph structure search, which efficiently speeds up the attention computation while maintaining the models' qualities. **Cluster-aware Graph Parallelism** splits the input graph tokens according to the cluster nature of graphs, thus boosting the attention computation throughput and facilitating system scalability. It also allows us to take advantage of the cluster feature in more fine-grained kernel optimizations. Inspired by the block-sparse format, **Elastic Computation Reformation** dynamically transfers the clustered attention pattern into a specialized cluster-sparse format to reduce the irregular memory access latency. It includes an *Auto Tuner* to automatically control the transfer to maintain the model convergence. Through extensive experiments, we show TorchGT successfully achieves scalable and efficient graph transformer training on large graphs. It also boosts training by up to 62.7× across various graph learning tasks while maintaining accuracy.

In summary, we make the following contributions:

- ★ TorchGT is the *first* graph transformer system that facilitates efficient, scalable, and accurate training on large-scale graphs as well as universal graph learning tasks.
- ★ TorchGT is the *first* to identify the major challenges that hinder existing graph transformers from scaling to large graphs and explore the graph-specific optimization opportunities which are neglected previously.
- ★ We propose three key techniques to meet all design goals from algorithm and system co-design perspectives.

★ Experiments show TorchGT achieves up to 62.7× speedup and near-linear scalability, supporting graph sequence lengths of up to millions.

4.2 Challenges and Motivation

4.2.1 Long Sequence for Graph Transformers

4.2.2 Issues and Opportunities

Most existing graph transformer works [5, 6, 21, 22, 41, 43] are only limited on small graphs due to a lack of compatible systems tailored for the graph transformer model training with long sequences. They have three main issues when applied to long sequence training.

I1: Attention Computation Bottleneck. Graph transformers with standard (dense) attention treat the graph as fully-connected with the MHA mechanism calculating attention for all node pairs. Thus, it requires the computation complexity of the attention module to be quadratic on the number of nodes (N^2) in a graph, which limits the models' scalability to extremely long sequences. Currently, there is a breakthrough in standard attention optimization, i.e., FlashAttention [133]. FlashAttention accelerates the attention module by fusing the IO-bound GPU kernels like `Softmax` and `Dropout` within the attention computation. On the Tensor Core Unit (TCU) of A100 GPU, FlashAttention can reach almost 200 peak TFLOPs under the attention computation with FP16/BF16 precision. However, even with FlashAttention to train graph transformers with long sequences, e.g., sequence length of 512K, we still identify that the attention module dominates the overall training time.

To show this, we conduct an experiment to record the iteration time breakdown when using FlashAttention, as illustrated in Figure 4.1. Current FlashAttention does not support the modified attention module like those augmented with bias encodings [5, 6, 25, 41], so we disable the bias in this experiment to only examine the computation efficiency. We separate the computation time of FlashAttention from the comprehensive training iteration. We can obviously observe that no matter on longer or shorter sequences, attention computation still dominates over 80% of training time, indicating a severe attention bottleneck. However, both the standard attention and FlashAttention fail to leverage one important characteristic of the graph, namely its topology structure, which we find profoundly impacts the effectiveness of system optimizations.

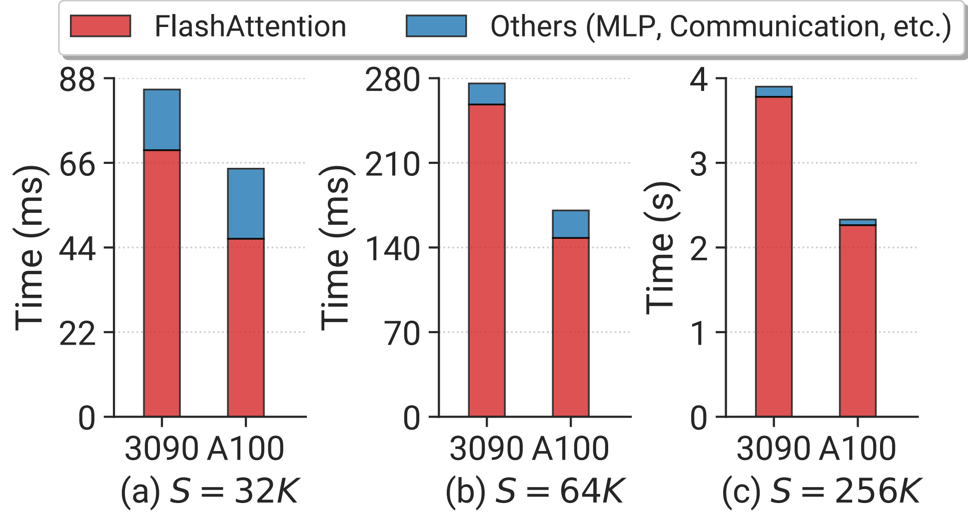


FIGURE 4.1: Training iteration time breakdown when training Graphormer on ogbn-products in different sequence lengths on two types of GPUs: RTX 3090 and A100.

I2: Degraded Model Convergence and Limited Tasks. Many efforts [21, 25, 77, 81] have been made to overcome the computation bottleneck of the attention module. Among them, [81] prunes the attention module and leaves a major backbone to reduce the computation cost for LLMs. Some works [77–80] propose sparse patterns on attention to scale linearly, but most of them are designed for natural language processing (NLP). They cannot be simply grafted to graph transformers since they fail to consider the inherent graph structure information when approximating attention, thus resulting in subpar model performance. Several graph transformers [36, 41, 74] harness neighbor sampling or graph pooling that only selects a subset of nodes to be trained at each iteration, without reducing the computation complexity. Nonetheless, all the above methods sacrifice model precision by dropping the connectivity information.

In the graph domain, efficient attention is not well studied. Few graph transformers like [22] apply the graph structure to attend nodes and maintain graph-specific information. However, they limit the implementation to the GNN-encoding-based model architecture, e.g., GraphGPS [21], and highly rely on the message-passing scheme for excellent model performance. Other methods [25, 75, 76] use self-defined adapted attention modules to achieve linear complexity. However, all those works are constrained to a single application task, failing to generalize to versatile graph tasks. Additionally, with GNN structure encodings or self-defined attention, the model can hardly be scaled to multiple workers. Therefore, motivated by these observations, in TorchGT we design a topology-induced attention pattern that greatly improves training efficiency while maintaining the models' qualities.

TABLE 4.2: Backward (BW) time of topology-pattern & dense counterpart when training Graphormer on ogbn-products.

Seq. Length	S=64K	S=128K	S=256K	S=512K
Topology-pattern BW. Time/ms	116.99	234.28	499.289	963.91
Dense BW. Time/ms	1.53	3.78	10.02	29.01

I3: Lack of Specific System Optimizations. As far as we know, currently there is no existing framework to optimize graph transformer training from the system level. FFN operations in MHA are dense in computation and regular in memory access. However, utilizing graphs on the attention module is sparse in computation and requires irregular and fine-grained memory access due to the skewed nature of graph structures, which inevitably becomes the performance barrier.

Existing solutions [5, 21, 22] directly apply graph topology in the attention computation while ignoring the pattern differences between graph transformer and standard Transformer-based models. To better illustrate, we experimentally examine the impact of irregular memory access by the topology-pattern attention in Table 4.2. The dense backward time is computed using the dense counterpart of the topology pattern. The topology-induced memory access latency is tremendous, reaching up to 33.2× slowdown than dense computation. To increase models' scalability, recent works [14–17] split the input sequences and distribute the computation across devices. However, this parallelism neglects various graph encoding modules and fails to distinguish input tokens in graph domain from tokens in NLP. Those differences invoke specialized and dedicated system designs for graph transformers towards more efficient memory optimizations and more aggressive parallelism.

4.3 System Design

To achieve efficient distributed graph transformer training with long sequence while maintaining model accuracy, we propose TorchGT, an algorithm-system co-optimized system tailored for graph transformer training on large-scale graphs. It follows four design principles:

- *Scalable.* TorchGT can scale graph transformer training to extremely large graphs (solving **I3**).
- *Efficient.* TorchGT reduces over 90% computation required by standard attention, overcoming the attention computation bottleneck (solving **I1**).
- *Convergence-maintained.* TorchGT maintains comparable model convergence to the graph transformer with standard attention, by explicitly balancing the efficiency

and quality trade-offs in two key designs: (i) *Dual-interleaved Attention* (§ 4.3.2) reduces computation by restricting attention to the topology-induced pattern, but may limit long-range interactions; thus we periodically interleave fully-connected attention to recover global information when needed. (ii) *Elastic Computation Reformation* (§ 4.3.4) improves throughput by transferring irregular sparse clusters into compact sub-blocks, but this transfer may perturb the original structure; therefore an *Auto Tuner* adjusts the transfer threshold to avoid noticeable convergence degradation (solving **I2**).

- *Task-agnostic.* TorchGT generalizes to various graph transformer models and graph learning tasks (graph-level and node-level) (solving **I2**).

Below we introduce its architecture and the detailed design of each component.

4.3.1 System Overview

Motivated by all the observations in §4.2.2, our key idea is to co-design an accuracy-maintained and compute-efficient attention module with a graph-parallelism-enabled system framework to support long sequence training. As shown in Figure 4.2, TorchGT intelligently optimizes training across three levels from the top to bottom hierarchy: *algorithm*, *runtime* and *kernel*. We propose a topology-induced and accurate attention algorithm in the algorithm level. We present a novel cluster-aware graph parallelism to scale the training in the runtime level. In the kernel level, we design a memory-optimized computation pattern specialized for clustered attention. Specifically, TorchGT consists of three key modules:

- *Dual-interleaved Attention*: In the algorithm level, it integrates locally graph-induced topology into the attention computation pattern and periodically overlays it with the fully-connected information, which efficiently reduces the computation burden while maintaining the model's quality as much as possible. It is tailored for versatile graph transformer models with local-global interleaved attention.
- *Cluster-aware Graph Parallelism*: From the distributed runtime perspective, we design a cluster-aware graph parallelism tailored for graph transformers. It splits the graph tokens in sequences according to the clustering nature of graphs, thus computing attention with locality and facilitating the system scalability.
- *Elastic Computation Reformation*: It reformats the graph-induced pattern obtained at the runtime level into our customized and fine-grained cluster-sparse pattern at the underlying execution kernel level. It further improves the attention computation throughput by greatly alleviating irregular memory access. To balance the efficiency-quality trade-off, we build an *Auto Tuner* to make an elastic transfer of cluster sparsity.

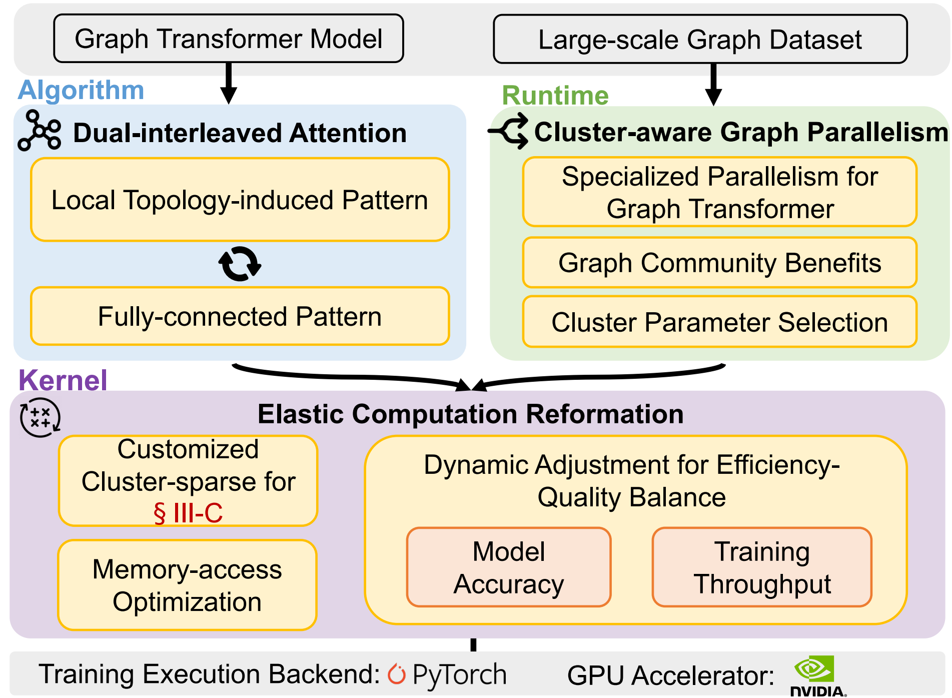


Figure 4.2: Overview of TorchGT architecture and workflow.

4.3.2 Dual-interleaved Attention

Motivated by **I1** in §4.2.2, TorchGT explores the opportunity of integrating graph structure to reduce the substantial computation cost. For optimizations aiming at graph transformers, we design an interleaved attention to realize a local-global interleaved attention and ensure model convergence. Specifically, it exploits the graph topology to improve the computational and memory efficiency by restricting the attention pattern, i.e., reducing the pairs of nodes that can interact with each other. Meanwhile, the fully-connected information is periodically interleaved into the graph-induced pattern. This attention is computationally lightweight and effective in keeping the model quality.

Local Topology-induced Pattern. In NLP tasks, the tokens in a sequence represent words, while in graph transformers the tokens are nodes of the input graph. Besides, most graph transformers like [6, 25] adopt the standard attention, which can be viewed as a fully-connected graph since all tokens attend to every other token to perform inner products, leading to quadratic complexity. Motivated by the sparse attention methods [77, 78], we find the local topology-induced pattern that makes use of the underlying structure of the input graph is desirable to guide the pair-wise node interactions. Graphs innately own two desiderata for attention mechanism: (1) *small pair-wise node interactions (large sparsity)*, and (2) *data locality*. In addition, most sparse patterns in NLP are only approximations [78] to their dense counterparts under specific contexts, while in our scenario the graph structure is real and valid, without

the need of approximations. Exploiting the graph structure information can not only significantly cut down heavy attention computation, but preserve the properties of the original graph dataset. Thus, we compute attention by attending each node to its immediate neighbors in the graph, reducing the interacted node pairs.

We formulate the local topology-induced pattern as below. To train on a graph $G = (V, E)$, we generate an input sequence $\boldsymbol{S} \in \mathbb{R}^{S\times d}$ comprised of graph tokens corresponding to a node set $\tilde{V} \in V$. $\tilde{V}$ can be equal to either the whole nodes V, e.g., in graph-level tasks, or a subset of V, e.g., in node-level tasks if the node number is too large. For each node set $\tilde{V}$, we construct a local attention graph $\tilde{G} = (\tilde{V}, \tilde{E})$, where the edge set $\tilde{E}$ is also a subset of the original edge set E. If there exists a global token in the model that attends to all nodes in the graph and is attended to by all nodes, we augment $\tilde{E}$ with the global token's edges. The general attention coefficient $\tilde{\boldsymbol{A}}_{ij}$ of graph transformers without graph encodings is computed in: $\tilde{\boldsymbol{A}}_{ij} = \text{softmax}\left(\frac{(h_i \boldsymbol{W}_{\tilde{\boldsymbol{Q}}})\cdot(h_j \boldsymbol{W}_{\tilde{\boldsymbol{K}}})^\top}{\sqrt{d_{\tilde{\boldsymbol{K}}}}}\right)$. The updated node attribute h_i' for each node i is computed as the weighted sum of the features of its neighboring nodes from $\tilde{V}$: $h_i' = \sum_{j\in\mathcal{N}(i)} \tilde{\boldsymbol{A}}_{ij}(h_j \boldsymbol{W}_{\tilde{\boldsymbol{V}}})$, where $\mathcal{N}(i)$ denotes the set of neighboring nodes of node i. Each neighbor's feature vector h_j is weighted by the attention coefficient $\tilde{\boldsymbol{A}}_{ij}$, and these weighted features are summed to update the attribute of node i. By using our local topology-induced pattern $\tilde{G}$, the attention only computes coefficients of connected node pairs.

Interleave Fully-connected Pattern. Implementing the attention computation via the graph structure can greatly reduce the computation cost. However, it sometimes slows the model accuracy and convergence, which can be shown by experiment results in Figure 4.9. The local graph-induced attention slightly degrades the model convergence, which is mainly because the topology-induced pattern restricts the attention mechanism from extracting the high-order neighboring information. Intuitively, larger sparsity induces more absence in the attention computation, and increases the model error. Building on this, we empirically interleave a fully-connected attention on the local graph-induced attention.

To fill the performance gap between sparse attention and its dense counterpart, we need to figure out when to interleave the dense pattern. Motivated by the sparse attention theories in [140], we conclude three critical conditions under which we use the topology-induced pattern on attention:

- **C1:** Every node in the sequence $\boldsymbol{S}$ always attends to itself.
- **C2:** There exists a Hamiltonian path that directly connects all nodes $\tilde{V}$ in the sequence.
- **C3:** All nodes in the sequence should be able to attend to other nodes, either directly or indirectly after L graph transformer attention layers.

The Hamiltonian path [141] or traceable path is a path in a graph that visits each node exactly once. For each graph $\tilde{G}$ corresponding to the input sequence, the *Dual-interleaved Attention* module searches it across the above three conditions. We use a heuristic approach Dirac's theorem [142] to do quick checks so the overhead is negligible in epoch time. If it satisfies these conditions, the sparse attention can universally approximate any sequence-to-sequence function at this moment. We perform attention computation with the topology-induced sparse pattern. Otherwise, TorchGT heuristically determines the current sparse pattern may introduce more errors than its dense counterpart. To ensure model quality, we utilize the fully-connected attention mechanism in this case to ensure model quality.

Computation & Memory Complexity. The topology of many real-world graphs can be immensely sparse. For instance, the ogbn-arxiv graph has 169K nodes and 1.2M edges, resulting in a sparsity of 4.1×10^{-5} (the proportion of nonzero elements in the whole adjacency matrix). This sparsity nature indicates that a very limited number of message propagation is enough to aggregate the inherent graph structural information. As a result, the local topology-induced attention significantly reduces the computation and memory-access complexity from $O(N^2)$ to $O(\tilde{E})$. Though we interleave several fully-connected attention occasionally, the overall computation efficiency is still improved significantly. Furthermore, the memory complexity can be reduced to $O(N)$ by tiling technique [133].

Model Convergence. Both the graph-centric attention [22] and classical graph neural networks [3, 24] propagate messages through edges in the nodes' neighborhood. Graph-centric attention [22] and classical GNNs [3, 24] prove that sparse attention can maintain the model convergence comparable to its dense counterpart. [39, 140] propose sparse attention can obtain similar universality as dense attention under some assumptions. Borrowing it to TorchGT, our *Dual-interleaved Attention* can provide universal approximation properties that every continuous function f can be approximated to any desired accuracy using a suitable sparse pattern under the three conditions, thus obtaining convergence similar to dense counterparts. These convergence analysis can be extended to our case and we refer the detailed analysis from them. Additionally, from results in §4.4.4, we can also conclude the model convergence maintains and sometimes is even better than the dense pattern.

4.3.3 Cluster-aware Graph Parallelism

To better fit the topology-induced attention pattern and increase the system scalability, we introduce a graph transformer-specialized parallel training style: *Cluster-aware Graph Parallelism*, which exploits the graph cluster characteristics to guide the distributed training.

Utilization of Graph Cluster. Graph cluster (community) [143, 144] is one essential characteristic of real-world graphs, referring to a subset of nodes within a graph that exhibit a higher degree of connectivity with each other compared to nodes in other parts of the graph. Although the graph structure-based attention in §4.3.2 greatly reduces the computation, this sparse and highly-skewed nature of graphs triggers substantial irregular memory access since edge connections are distributed in an uneven pattern, bringing extra overhead to training. Consequently, employing the graph cluster structure on GPUs is promising for graph transformer training improvement. There exist some approaches in traditional graph learning [144, 145] to utilize graph cluster, but they are aimed for CPU processing with limited parallelization. [107] also exploits graph cluster but focuses on redundant data loading in GNN computing.

Therefore, to explore the performance benefits of graph cluster on graph attention computing, we incorporate a lightweight node reordering to cluster nodes and improve the spatial locality during attention computation, without changing the connectivity correctness. The key insight is that the proximity of node IDs is more likely to be scheduled to the adjacency of computing units on GPUs where they get processed. In detail, we leverage METIS [58], a community-based graph reordering technique for great cluster locality and ease of integration with parallelism. Specifically, it uses multilevel recursive bipartitioning to divide and coarsen the graph while preserving the essential structure. Then, it applies partitioning algorithms to find an optimal node ordering that reduces the cost of accessing neighboring nodes. We optimize the implementation of METIS for a lower cost: we capture the cluster information of graphs and map such locality from the upper level to the underlying GPU kernels, which also enables us to leverage the L1 & L2 cache for refined cluster capturing (later discussed in §4.3.4).

Specialized Graph Parallelism. To increase the scalability of graph transformers, intuitively TorchGT employs parallelism technologies to dispense the computation across devices. There have been extensive studies in sequence parallelism technologies [14–16] for LLMs to support efficient long sequence training. However, current parallelism methods for language models trigger two challenges when applied to graph transformers: (1) failing to leverage graph properties; (2) not supporting various graph encodings. In traditional language models, the input sequence encodes the context of a specific sentence. As such, training the language model requires tokens in the input sequence concatenated in a pre-defined order. In contrast, we observe that for graph learning tasks, *there is no need for graph transformers to predict sequences (graph-level task) or tokens (node-level task) within a position-fixed context*, since they only rely on the graph topology to construct the structural encodings. A motivating example of parallelizing graphs with graph cluster is the graph-level task, where only

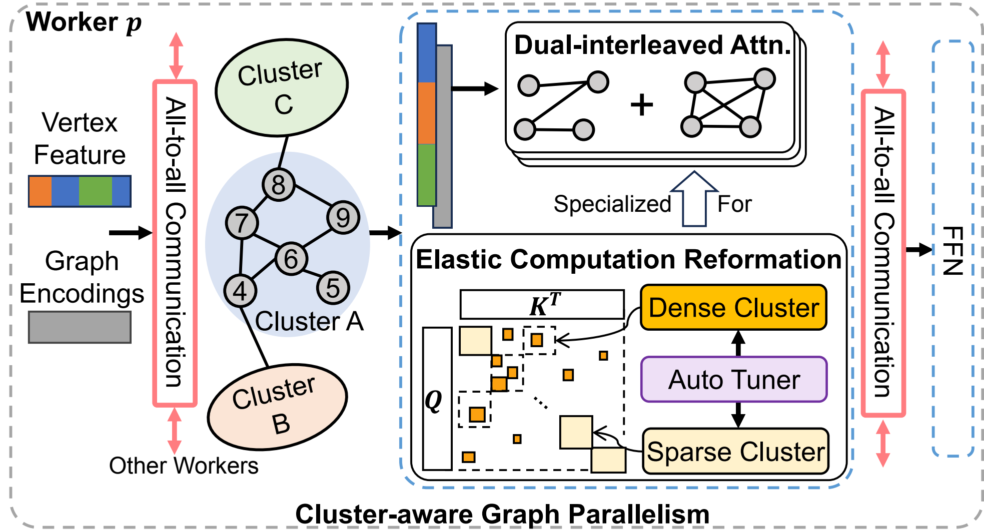


FIGURE 4.3: Detailed training process on one worker with TorchGT, which includes three key components.

the global token is critical for inferring the graph type and other node tokens can be arranged in any order. For the node-level task, node tokens can be reorganized with their corresponding structural encodings. Therefore, the graph tokens are not required to be in a pre-defined order.

Based on this insight, we are the ***first*** to design a *Cluster-aware Graph Parallelism* ***specialized for graph transformers***, as shown in Figure 4.3. Specifically, the raw input sequences $\boldsymbol{S}$ and graph encodings $\boldsymbol{B}$ are randomly partitioned across P devices. Each local sub-sequence $\boldsymbol{S}_{sub}$ and sub-encodings are projected to local matrices: $\boldsymbol{Q}_{sub}, \boldsymbol{K}_{sub}, \boldsymbol{V}_{sub}, \boldsymbol{B}_{sub} \in \mathbb{R}^{\frac{S}{P} \times d}$, assuming they have the same dimensionality. Then in each graph transformer layer, all subspaces are combined together into complete matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, and $\boldsymbol{B}$ via the highly efficient all-to-all collective communication operation. All-to-all operation owns an advantage over other communication operations (e.g., all-gather and reduce-scatter) in terms of much smaller communication volume and overall better scalability, which is also proved in [15]. All-to-all gathers matrices in sequence dimension and splits in the head, resulting in $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, and $\boldsymbol{B} \in \mathbb{R}^{S \times \frac{d}{P}}$. Now that since matrices are complete in the sequence dimension, TorchGT reorganizes the layout according to the clustering nature of graphs discussed before. Then the *Dual-interleaved Attention* conducts attention computation in the clustered layout, exemplified as blue, orange and green rectangles in Figure 4.3. After attention computation, another all-to-all transforms the output tensor back to subspace $\boldsymbol{S}'_{sub}$ for subsequent operators such as FFN and layer normalization in the graph transformer layer.

Communication Complexity. Thanks to all-to-all, *Cluster-aware Graph Parallelism* has low communication volume and scales exactly well with more servers. Given

hidden size d, sequence length S, and parallelism degree P, TorchGT performs all-to-all for the $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, and $\boldsymbol{B}$ with an a total message size $3Sd$ ($\boldsymbol{B}$ is in sparse format so communication is negligible) before the attention computation, and another all-to-all for attention output with size Sd. Therefore, TorchGT performs two all-to-alls with communication volume per GPU of $4Sd/P$ and communication complexity of $O(S/P)$, while other operations like all-gather have communication complexity of $O(S)$. Thus, TorchGT has better scalability with increasing parallelism degree on extremely long sequences.

In summary, compared with sequence parallelism methods for LLMs, our *Cluster-aware Graph Parallelism* favors the graph transformer architecture in several aspects. First, all-to-all gathers in sequence dimension, leading to exactly integrated graph topology, which the topology-induced sparse pattern in §4.3.2 can be perfectly applied to. Second, the graph encodings $\boldsymbol{B}$ share the same sparse layout as attention mapping so the parallelism of graph transformers only brings a trivial memory footprint and communication overhead, thus facilitating model scalability and ensuring memory efficiency.

4.3.4 Elastic Computation Reformation

Cluster Sparsity. The topology-induced attention pattern can significantly reduce the computation cost, but also leading to substantial irregular memory access due to the highly skewed nature of graphs. Although the graph cluster alleviates this problem, it also has some limitations. Figure 4.4(b) gives an example with the cluster dimensionality of 8 and sequence length of 64K. We observe that only the diagonal clusters in the clustered adjacency matrix appear in dense patterns most and show lower sparsity, which can benefit from locality since nodes in each cluster are close to each other. On the other hand, other clusters appear highly sparse patterns and more irregular shapes (denoted as *sparse cluster*). Accessing the embeddings of computation like aggregation in this pattern requires a large number of atomic operations. Consequently, those remaining irregular clusters still engender heavy overhead. To exemplify, training Graphormer on ogbn-arxiv (S=64K) in Figure 4.4(b) takes 375 ms per epoch, while its dense counterpart only takes 81ms.

To further reduce the memory access latency, we propose a memory-efficient *Elastic Computation Reformation* which introduces the cluster sparsity. Motivated by the block-sparse pattern in [78, 138, 139], we reformat the clustered layout in Figure 4.4(b) to a fine-grained cluster-level fashion in Figure 4.4(c). Specifically, as shown in Figure 4.3, for each sparse cluster, TorchGT transfers the scattered edges inside it to multiple substructures of compact and adjacent edge connections, which is denoted as *sub-blocks*. The transferred *dense cluster* can have multiple randomly scattered

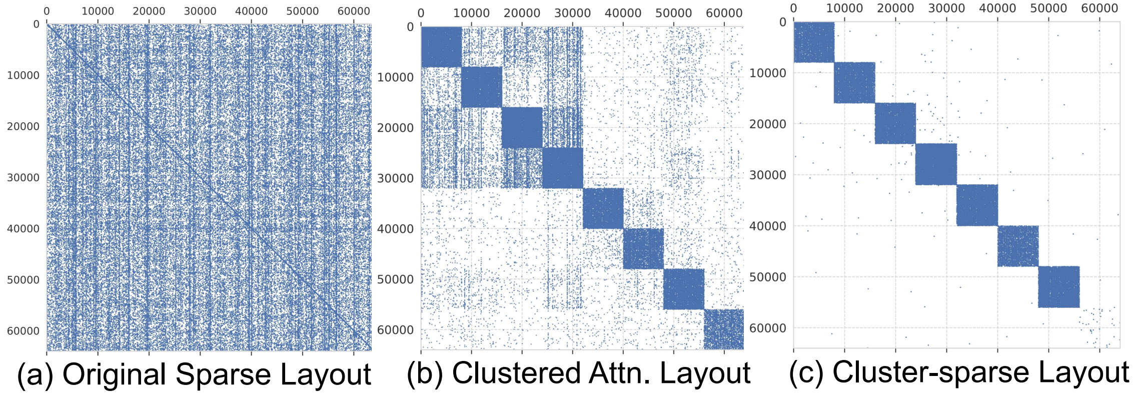


FIGURE 4.4: Three attention layouts after *Dual-interleaved Attention*, *Cluster-aware Graph Parallelism* and *Elastic Computation Reformation* respectively. (c) is obtained by compacting elements to adjacent neighbors inside clusters.

sub-blocks, the number of which is decided by the number of real edges in the cluster and the dimension of sub-block d_b. Note that there will be totally k^2 clusters for the whole layout with the cluster dimensionality of k. Figure 4.4(c) depicts the cluster-sparse layout with k=8, in which most sparse clusters are transferred to dense ones with tight sub-blocks. This cluster sparsity offsets the downside of irregular memory access incurred by the topology-induced pattern. Then we can generate a masking matrix $\boldsymbol{M}$ of pattern Figure 4.4(c), and compute attention scores as follows:

$$\tilde{\boldsymbol{A}}_{ij} = \text{softmax}\left(\frac{(h_i \boldsymbol{W}_{\tilde{\boldsymbol{Q}}}) \cdot (h_j \boldsymbol{W}_{\tilde{\boldsymbol{K}}})^\top}{\sqrt{d_{\tilde{\boldsymbol{K}}}}}\right) \odot \boldsymbol{M} \tag{4.1}$$

where $\odot$ is defined by $(\boldsymbol{A} \odot \boldsymbol{M})_{ij} = \boldsymbol{A}_{ij}$ if $\boldsymbol{M}_{ij} = 1$, otherwise $= -\infty$ if $\boldsymbol{M}_{ij} = 0$.

Transfer Strategy. Although we can reduce the irregular memory access by transferring clusters to dense ones, applying static cluster-sparse transferring will result in model quality degradation since the cluster sparsity changes the original graph structure by modifying edges. Existing methods of block-sparse [139, 146] usually adopt static strategies as well. Simply chasing the extreme throughput by transferring all clusters or deploying static attention layout will not fully utilize the benefits of cluster property. Some performance-related information (e.g., convergence) for model training is only available at runtime. Without considering the runtime information, the system will suffer from an inferior model accuracy or inefficient memory access. This urges the need for elastic designs to cope with a wide spectrum of circumstances.

Thus, TorchGT designs the following two strategies:

- *Indolent Transferring.* TorchGT only transfers clusters that are extremely sparse and irregular. Although such an inactive way may miss some optimization opportunities, it can refrain from model quality decline and be more portable. Concretely,

TorchGT only transfers sparse clusters whose sparsity value β_C is smaller than that of the current whole graph β_G. Note that the sparsity value refers to the proportion of **nonzero** elements in the whole adjacency matrix.

- *Elastic Transferring.* We dynamically adjust the amount of transferred dense clusters along the training. First, we set a threshold value β_{thre} for controlling cluster-sparse transfer. If the sparsity of a cluster β_C is smaller than the threshold β_{thre}, TorchGT transfers it to a dense cluster. To decide the value of β_{thre} in each training epoch, we design an *Auto Tuner* in the next part for modeling β_{thre}.

Hyperparameter Modeling. The hyperparameters can be tuned to accommodate various graph patterns. We design an *Auto Tuner* to dynamically select the hyperparameters k, d_b and β_{thre}, and formulate the analytical model. Here, we treat the cluster dimensionality k, the size of sub-block d_b and the cluster-sparse transfer threshold β_{thre} as hyperparameters to balance computation efficiency and model quality.

(1) Cluster dimensionality k, sub-block dimension d_b. We can leverage GPU L1 and L2 caches to improve the memory access locality of sub-block computation. In this way, smaller sub-blocks can enjoy the data locality benefit from the L1 cache while larger clusters can enjoy the locality from the larger L2 cache. Specifically, we determine k as: $k = \lfloor \sqrt{\frac{Q_{\mathrm{L2}}}{id}} \rfloor, i \in \mathbb{N}$, where Q_{L2} is the L2 cache size and d is model hidden dimension. To determine the sub-block dimension d_b, we profile the computation throughput and some hardware statistics of the indexing kernel w.r.t. different d_b values. Figure 4.5(a) shows the workload balance in GPU computing unit downgrades (the lower warp occupancy, the worse balance) as d_b increases, while both L1 & L2 cache hit rates increase. Thus, there exists a trade-off between these two metrics in deciding d_b. Moreover, Figure 4.5(b) also demonstrates the values of obtaining the optimal computation throughput lie in the middle range. Both cases suggest the middle value is the ideal choice. For example, for RTX 3090 GPU and model hidden dimension of 64, we fit k=8 and d_b=16. These two values can also be selected by users.

(2) Transfer threshold β_{thre}. Motivated by [18], to estimate the convergence, *Auto Tuner* tracks a running average loss $F_t = 0.9F_{t-1} + 0.1\mathcal{L}_t$, where $\mathcal{L}_t$ is the loss at epoch t. Considering the training throughput, a Loss Descent Rate (LDR) is defined as $LDR_t = \frac{F_t - F_{t-1}}{et_t}$, where et_t is the t-th epoch training time. At the beginning $\beta_{thre,0}$ is initialized as β_G from the set $\{0, \beta_G, 1.5\beta_G, 5\beta_G, 7\beta_G, 10\beta_G, 1\}$, which is developed by profiling different datasets. When $LDR_t \geq LDR_{t-\delta}$ for some $\delta \in \mathbb{N}$ (here we use $\delta = 10$) which specifies the range of epochs for LDR comparisons, TorchGT heuristically determines the current β_{thre} suffices to reduce the loss. Then *Auto Tuner* increases β_{thre} to the next value in the set to gain higher speed. On the other hand, $LDR_t < LDR_{t-\delta}$ in δ epochs denotes the training is about to converge or too many

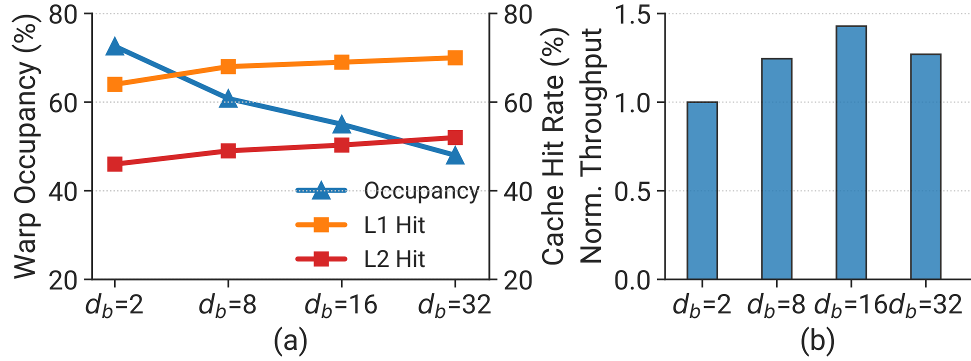


FIGURE 4.5: (a) Profiled hardware statistics of GPU when computing in different sub-block size d_b. The ideal d_b considers both load balance and cache hit rate. (b) Computation throughput of the indexing kernel in different d_b normalized on that of d_b=2.

errors are introduced. In this case, *Auto Tuner* reduces β_{thre} to the previous value from the set to enable more stable and accurate training.

4.4 Evaluation

We implement TorchGT atop PyTorch 2.1 [117]. We study the performance of TorchGT on versatile datasets and graph learning tasks in the following aspects: (1) Efficiency, (2) Convergence, (3) Scalability, and (4) micro-benchmarks and ablation studies to examine the impact of each technique and hyperparameter.

- **Efficiency** We compare the training throughput of TorchGT with baselines (§ 4.4.1).
- **Convergence** We study the model quality obtained by TorchGT and baselines and examine the convergence rate (§ 4.4.2).
- **Scalability** We study how well TorchGT can scale to long sequences and scale out in distributed clusters with multiple GPUs (§ 4.4.3).
- **Microbenchmarks and Ablation Studies** We conduct microbenchmarks on TorchGT to examine the impact of each design and the settings of hyperparameters in TorchGT (§ 4.4.4).

Datasets and Models. We evaluate TorchGT on versatile real-world graph datasets with multiple scales. The detailed information is shown in Table 4.3, including both node-level and graph-level tasks. The MalNet dataset is constructed from all categories of the full datasets. The Arxiv, Products, and Paper100M are available in Open Graph Benchmark [1]. The Malnet (MA) is generated from the Malnet dataset [26]. For each dataset, we choose a specific graph learning task for evaluation. We use three classical graph transformer models commonly adopted for evaluation, including

TABLE 4.3: Detailed information of datasets in evaluation.

Node-level				
Datasets	**# Nodes**	**# Edges**	**# Feats**	**Task**
Amazon [2]	1,598,960	132,169,734	200	107-class Classif.
ogbn-arxiv [1]	169,343	1,166,243	128	40-class Classif.
ogbn-products [1]	2,449,029	61,859,140	100	47-class Classif.
ogbn-papers100M [1]	111,059,956	1,615,685,872	128	Binary Classif.
Graph-level				
Datasets	**# Graphs**	**Avg. # Nodes**	**Avg. # Edges**	**Task**
ZINC [147]	12,000	23.2	24.9	Regression
ogbg-molpcba [1]	437,929	26.0	28.1	128-task Classif.
MalNet [26]	10,833	15,378	35,167	5-class Classif.

TABLE 4.4: Detailed information of graph transformer models.

Models	**# Layers**	**Hidden Dim.**	**# Head**
Graphormer$_{slim}$ (GPH$_{slim}$)	4	64	8
Graphormer$_{large}$ (GPH$_{large}$)	12	768	32
GT	4	128	8

Graphormer$_{Slim}$ (GPH$_{Slim}$) [6], Graphormer$_{Large}$ (GPH$_{Large}$) [6], and GT [5]. Note that TorchGT can also be applied to other graph transformer models. As shown in Table 4.4, we follow the hyperparameter configurations reported in their original papers as closely as possible.

Baselines. All models cannot be directly trained on selected large graphs. Due to the lack of existing graph transformer systems, we meticulously replicate each model with simple graph parallelism following its original implementation as the vanilla version, denoted as GP-RAW. On the basis of this, we have also developed other variants incorporating FlashAttention [133] denoted as GP-FLASH, and topology-induced sparse attention denoted as GP-SPARSE. Additionally, we have implemented graph parallelism to scale all baseline models to multi-GPUs.

Testbed. Our experiments are performed on two GPU servers. ❶3 GPU servers each with 8 NVIDIA RTX 3090 GPUs (24GB). Intra-server connections (CPU-GPU, GPU-GPU) are based on PCIe 4.0x16 lanes and inter-server connections are via 1Gbps Ethernet. ❷2 servers each with 8 A100 GPUs (80GB) with NVLink and 200Gbps InfiniBand.

TABLE 4.5: Detailed comparison of training speed and test accuracy of methods when training on one 3090 GPU server. OOM means the method runs out of memory. TorchGT always outperforms GP-FLASH in throughput and accuracy on all the models and datasets. GP-RAW with full attention runs out of memory in all cases.

Model	Method	MalNet		ogbn-papers100m		ogbn-products		ogbn-arxiv		Amazon	
		t_{epoch}(s)	Test Acc.(%)	t_{epoch}(s)	Test Acc.(%)	t_{epoch}(s)	Test Acc.(%)	t_{epoch}(s)	Test Acc.(%)	t_{epoch}/s	Test Acc.(%)
GPH$_{Slim}$	GP-Raw	OOM	-	OOM	-	OOM	-	OOM	-	OOM	-
	GP-Flash	2158.37	90.87	1201.13	90.11	27.69	66.39	0.44	48.25	17.31	63.51
	TorchGT	195.54(**11.0**×)	92.71	19.15(**62.7**×)	96.82	0.54(**50.8**×)	66.75	0.11(**3.9**×)	53.81	1.00(**17.5**×)	73.10
GPH$_{Large}$	GP-Raw	OOM		OOM	OOM	OOM	-	OOM	-	OOM	-
	GP-Flash			2512.88	96.93	56.51	44.48	3.46	22.11	36.83	73.34
	TorchGT			654.72(**3.8**×)	98.60	16.10(**3.5**×)	63.06	1.16(**3.0**×)	42.38	11.07(**3.3**×)	73.75
GT	GP-Raw	OOM	-	OOM	-	OOM	-	OOM	-	OOM	-
	GP-Flash	1426.24	74.54	1235.02	88.86	28.80	66.20	0.50	53.98	8.88	69.07
	TorchGT	242.58(**5.9**×)	90.13	26.33(**46.9**×)	89.60	0.79(**36.3**×)	82.11	0.09(**5.3**×)	56.72	0.76(**11.7**×)	72.98

TABLE 4.6: Training time per epoch of trianing GPH$_{Slim}$ on one A100 server. TorchGT can still improve training efficiency compared with GP-FLASH.

Model	Method	MalNet	ogbn-papers100m	ogbn-products	Amazon
		t_{epoch}(s)	t_{epoch}(s)	t_{epoch}(s)	t_{epoch}(s)
GPH$_{Slim}$	GP-Flash	668.23	492.79	5.34	3.43
	TorchGT	160.61(**4.2**×)	244.07(**2.1**×)	2.86(**1.9**×)	1.69(**2.0**×)

4.4.1 End-to-end Training Throughput

We compare the end-to-end training time per epoch and test accuracy of TorchGT with all baselines on one server, as shown in Table 4.5. The sequence length is 256K for GPH$_{Slim}$ and GT, and 32K for GPH$_{Large}$. When training on ogbn-arxiv, we set the sequence length to 64K for GPH$_{Slim}$ and GT. The speedup in the bracket is the relative throughput of each method on the basis of GP-FLASH. In each training task, we treat the first 10 epochs as the warmup stage and only record statistics afterward.

TorchGT substantially outperforms GP-FLASH by 3.3∼62.7×. This is mainly because TorchGT significantly reduces the computation complexity of the attention module. Additionally, GP-RAW runs out of memory (OOM) on all datasets under the current sequence lengths due to its $O(N^2)$ memory complexity of the attention module. For instance, GP-RAW requires over 200GB memory to store the attention score, i.e., $QK^\top$, of only one attention head for the ogbn-products dataset. We also conduct evaluations of GPH$_{Large}$ on one A100 server as shown in Table 4.6. TorchGT still shows impressive acceleration and outperforms GP-FLASH up to 4.2× on such frontier equipment. In summary, TorchGT realizes efficient training of graph transformers with a marvelous improvement.

The speedup difference is mainly related to the input graph topology and model structure under long sequences. If the input graph is very sparse (up to 99% sparsity), *Dual-interleaved Attention* first boosts attention by a large margin. If an obvious clustering pattern exists in the graph, then attention can be further accelerated by

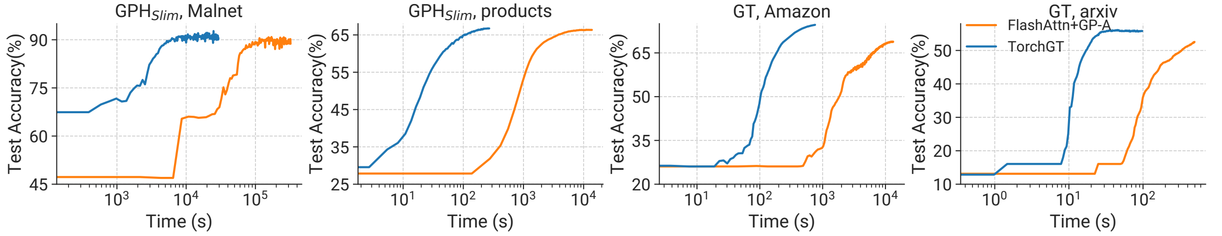


FIGURE 4.6: The convergence curve comparisons of TorchGT and GP-FLASH on different models and datasets.

2∼3× with the other two modules. Graph transformer model varies in layer number, hidden size and FFNs, thus resulting in different computation time per portion. Since we mainly improve attention computation, the more proportion it accounts in total model computation, the higher speedup on epoch time. For instance, in Table 4.5 GPH$_{Slim}$ achieves notable speedup on papers100M owing to the above. Ogbn-arxiv shows a smaller speedup on all models since it has poorer sparsity and cluster property. Therefore, TorchGT can gain speedup in most cases, and bring greater improvement when the input graph can be clustered and attention dominates model computation. In Table 4.5, TorchGT also performs better in model accuracy and successfully maintains model quality.

4.4.2 Model Convergence

We examine the model accuracy and convergence curves of TorchGT on various models in Table 4.5 and Figure 4.6. Table 4.5 summarizes the test accuracy achieved by all systems on three models. On large-scale datasets, TorchGT gives higher model accuracy while GP-RAW runs out of memory. GP-FLASH harms the model accuracy in some datasets, e.g., Malnet and Amazon since FlashAttention only supports FP16/BF16 precision [133] in computing which may downgrade model convergence compared to FP32 precision. In contrast, TorchGT supports FP32 precision without compromising model accuracy. To better validate this, we compare the training throughput and test accuracy of GP-FLASH and TorchGT-BF16 in Table 4.7. On BF16, TorchGT obtains similar accuracy with FlashAttention, indicating the accuracy drop of FlashAttention is mainly caused by reduced precision. TorchGT even achieves higher speedup with BF16, but we choose to use FP32 since it gives higher accuracy with notable speedup. From Figure 4.6, we also see that GP-FLASH converges to low accuracy and lead to far slower convergence than our system, verifying it preserves model quality well.

In this set of experiments, we compare the model convergence of TorchGT with GP-RAW and GP-FLASH. Table 4.5 summarizes the test accuracy achieved by all systems on three models. On small datasets, we can observe that TorchGT maintains

TABLE 4.7: Training throughput and test accuracy of methods. The accuracy of GP-FLASH decreases because of BF16.

		GP-Flash	**TorchGT-BF16**	**TorchGT-FP32**
ogbn-arxiv	t_{epoch}(s)	0.44	**0.08**	0.11
	Test Acc.(%)	48.25	48.29	**53.81**
Amazon	t_{epoch}(s)	17.31	**0.60**	1.00
	Test Acc.(%)	63.51	63.58	**73.10**

the model convergence compared to the original graph transformer models with standard attention (GP-RAW). On large-scale datasets, TorchGT achieves high accuracy while GP-RAW runs out of memory. GP-FLASH harms the model accuracy in some datasets, e.g., Malnet, ogbn-products, and Amazon. This is because the FlashAttention requires FP16/BF16 precision [133] which loses convergences compared to FP32 precision. In contrast, TorchGT supports FP32 precision without compromising model convergence. In conclusion, TorchGT achieves the goal G3.

4.4.3 System Scalability

In this set of experiments, we evaluate the scalability of TorchGT to long sequences. We take the GPH_{Slim} model and ogbn-products dataset for evaluation.

Training Throughput on Multiple Servers. First, we evaluate the training throughput of TorchGT on multiple A100 servers to validate it also scales out well with more servers. As shown in Figure 4.7, we conduct two sets of scalability evaluations on up to 8 servers(each with 8 A100 GPUs) with extremely long sequences and record the sequence training time on the ogbn-products dataset. In Figure 4.7(a), we fix the sequence length to 1024K and increase the server number. We can see TorchGT still obtains notable speedup when scaling to more servers. Especially, when the GPU count is doubled, the training throughput correspondingly increases by almost 1.7×, indicating a certain degree of scalability. In Figure 4.7(b), we fix the computational load per GPU when increasing the sequence length from 256K to 512K. Note that when doubling the sequence length, we need 4× GPUs than before to keep the same computational load per GPU(attention calculation is proportional to S^2/P). In this case, TorchGT achieves approximately the same throughput on each GPU as before, also verifying good scalability.

Sequence Length w.r.t. Number of GPUs. We examine the maximum sequence length of GPH_{Slim} that can be trained on 1∼8 GPUs with TorchGT in Figure 4.8(a). Note that GP-RAW employs standard full attention. We can see the maximum sequence length of TorchGT can reach up to 1.3M on 8 GPUs. It also enables the

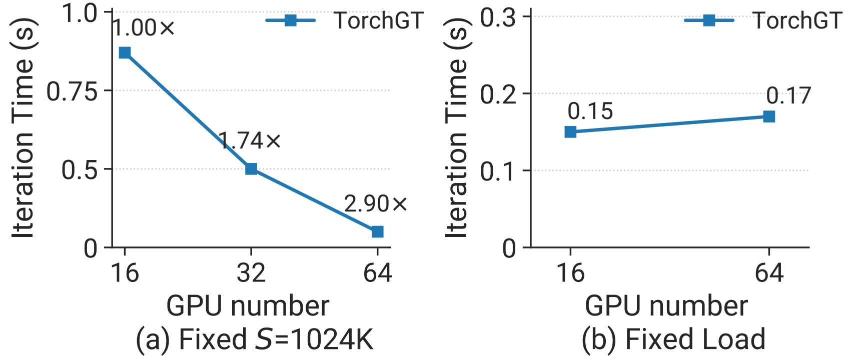


FIGURE 4.7: Scalability results of TorchGT in training GPH$_{Slim}$ on ogbn-products on many A100 servers. (a) With fixed sequence length, throughput reduces almost linearly. (b) With fixed computational load per GPU, throughput remains well.

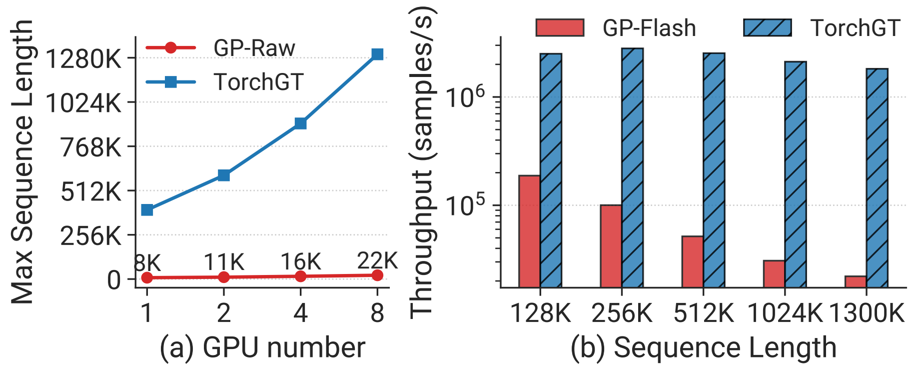


FIGURE 4.8: Scalability experiments of training GPH$_{Slim}$ on ogbgn-products. (a) The supported maximum sequence length w.r.t. GPU number. (b) Training throughput w.r.t. sequence length. In both cases TorchGT shows greater scalability than others.

sequence length of 400K with only 1 GPU, substantially 50× larger than that of GP-Raw. Moreover, the sequence length of TorchGT almost scales linearly w.r.t. the number of GPUs, while the maximum sequence length GP-Raw can support nearly remains unchanged with the growth of GPU numbers. With 8 GPUs, TorchGT supports 1.3M in length while GP-Raw only supports 22K in length.

Throughput w.r.t. Sequence Length. We further compare the training throughput of TorchGT and GP-Flash under sequence lengths varying from 128K to 1300K in Figure 4.8(b). We fix the number of GPUs to 8 and report the throughput as samples per second. Figure 4.8(b) shows that the training throughput of GP-Flash sharply decreases from 1.9×10^5 samples/s to 2.2×10^4 samples/s when the sequence length increases. The speed degradation of GP-Flash mainly comes from the computation bottleneck of FlashAttention with $O(N^2)$ complexity. In contrast, TorchGT maintains the training throughput at around 2.5×10^6 samples/s by significantly reducing the attention computation costs (in §4.3.2).

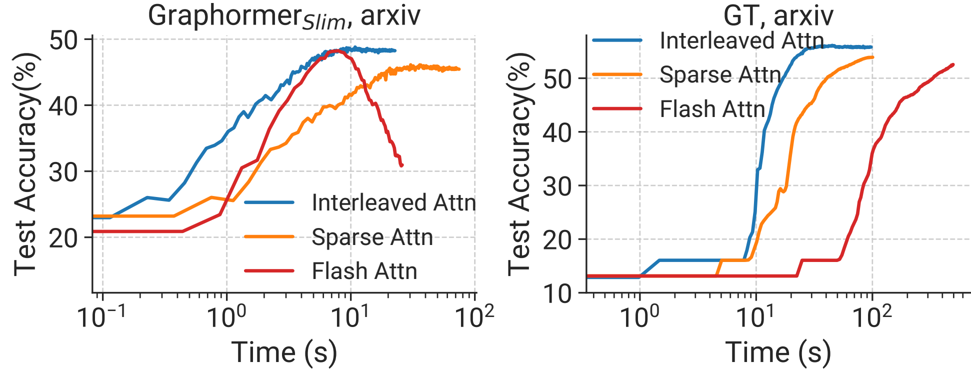


FIGURE 4.9: Convergence comparisons of different attentions: our interleaved attention, FlashAttention and sparse attention.

4.4.4 Micro-benchmarks

We explore the effects of each component in TorchGT via ablation studies and perform sensitivity analysis of the introduced hyperparameters on one 3090 GPU server.

Impact of *Dual-interleaved Attention.* After employing the topology-induced attention and interleaving mode introduced in §4.3.2, we investigate their effect on model quality. Specifically, we train models on large and small graph datasets to evaluate the convergence.

(1) On large-scale graphs. We measure the convergence curves of GPH$_{Slim}$ and GT on ogbn-arxiv, and compare the convergence of interleaved attention with that of FlashAttention and the sparse variant in Figure 4.9. The model with interleaved attention shows faster convergence than the other two and finally converges to higher accuracy, verifying that the interleaved attention improves computation efficiency while displaying great convergence.

(2) On small graphs. Since the raw graph transformer models fail to be trained on large graphs, we further evaluate the convergence of interleaved attention on small graphs in Figure 4.10. Sparse attention shows the worst convergence rate while full attention has the best. The model with interleaved attention converges to nearly the same as the model with full attention and obviously outperforms the sparse variant in both convergence speed and final test score. On ogbg-molpcba dataset, it is almost identical to that of the full attention.

Impact of *Elastic Computation Reformation.* We particularly examine the impact of the *Elastic Computation Reformation* module in TorchGT, FlashAttention, and sparse attention w.r.t. the sequence length and the model hidden dimension on one GPU. Note that we implement the sparse variant with the pure topology-induced attention pattern.

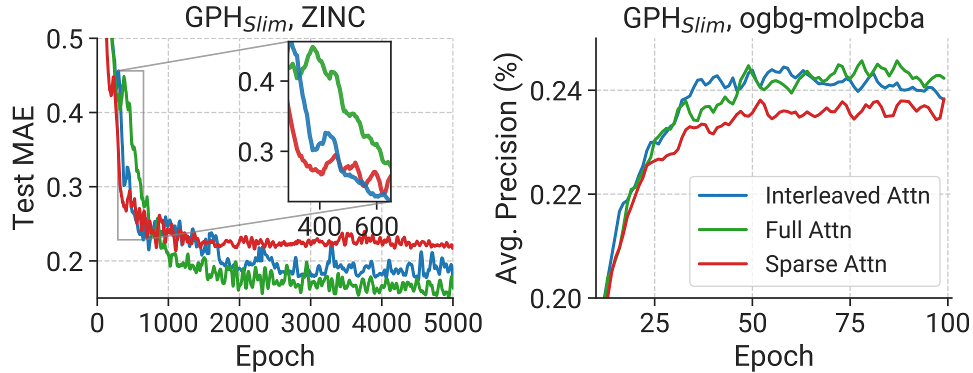


FIGURE 4.10: Convergence curves of different attentions: our interleaved attention, full and sparse attention on small graphs.

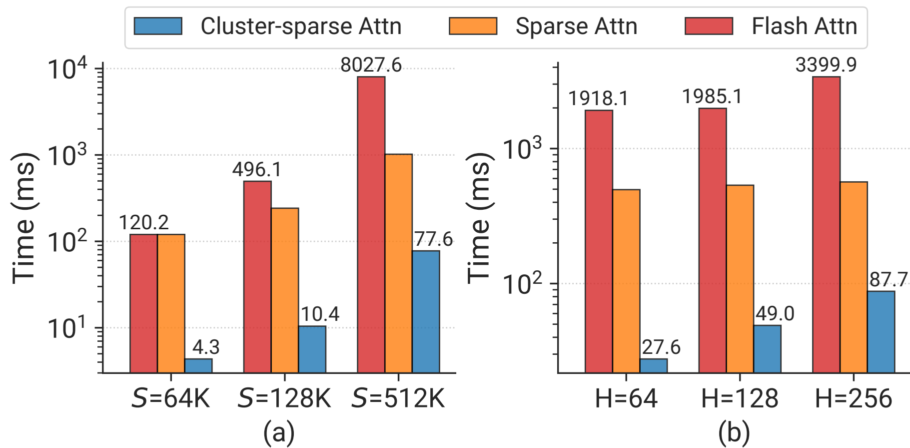


FIGURE 4.11: Computation time of attention modules when training Graphormer on ogbn-products with different (a) sequence lengths and (b) model hidden dimensions when S=256K.

(1) Attention computation time w.r.t. sequence length. In Figure 4.11(a), we first evaluate the speed of the attention module when varying the sequence length from 64K to 512K. We use the model hyperparameter setting in GPH_{Slim} and record the computation time of the attention module in each method. Clearly, as the sequence length increases, the computation time of FlashAttention grows quadratically, resulting in heavy training slowdown. The sparse attention improves the computation speed a bit, but shares a similar computation speed as FlashAttention when the sequence length is small. In contrast, TorchGT essentially improves the computation efficiency by up to 103.4× compared to FlashAttention. It is even faster than the sparse attention largely, validating its effectiveness in reducing irregular memory access with our cluster-sparse pattern.

(2) Attention computation time w.r.t. hidden dimension. We fix the sequence length to 256K and change the hidden dimension from 64 to 256. The computation time of the attention module is recorded in Figure 4.11(b). When the model size increases,

TABLE 4.8: Training time per epoch and test accuracy on ogbn-arxiv dataset regarding different transfer threshold β_{thre}.

	β_{thre}	β_G	$1.5\beta_G$	$5\beta_G$	$7\beta_G$	$10\beta_G$	TorchGT
GPH$_{Slim}$	t_{epoch}**(s)**	0.368	0.257	0.088	0.087	0.077	0.114
	Test Acc.(%)	53.34	54.19	53.82	50.84	48.31	53.81
GT	t_{epoch}**(s)**	0.098	0.090	0.089	0.084	0.071	0.093
	Test Acc.(%)	56.70	56.84	56.51	53.65	45.95	56.72

TorchGT still largely outperforms FlashAttention and sparse attention in all cases, owing to the specialized cluster sparsity in *Elastic Computation Reformation* module. We can conclude that FlashAttention shows poorer adaptation on long sequences, compared to its higher tolerance on larger model sizes. This indicates the better scalability of TorchGT for long sequences and large model sizes.

(3) Sensitivity analysis of transfer threshold. The introduced hyperparameter β_{thre} determines the model performance and efficiency of TorchGT. As shown in Table 4.8, we adopt different values of β_{thre} and record the training time per epoch and test accuracy. Note that a larger β_{thre} means more clusters are transferred to dense ones with sub-blocks. Higher accuracy can be obtained when smaller β_{thre} is adopted, but coming with possibly lower training speed (e.g., for GraphSAGE, 0.368s with $\beta_{thre} = \beta_G$ vs. 0.077s with $\beta_{thre} = 10\beta_G$). Seriously chasing the lowest error ($\beta_{thre} = 0$) or just caring about the highest training throughput ($\beta_{thre} = 1$) is not the best choice to fully utilize the benefits of the cluster-sparse pattern. Choosing different values always creates a trade-off between efficiency and accuracy and further analysis on the value choice can be done in the future. Currently, we suggest $\beta_{thre} = 5\beta_G$ for better balance.

4.4.5 Pre-processing Cost

We record the pre-processing cost versus model convergence time on both tasks to understand how much extra time is brought by TorchGT. The proportion is 5.2s (5.4%) versus 91.2s (94.6%) for ogbn-arxiv, and 239.7s (2.0%) versus 11732.4s (98.0%) for MalNet. The overhead only occupies less than 5.4% of the total training time on all epochs, which is acceptable compared with the huge model convergence time.

4.5 Summary

This work introduces TorchGT, a scalable and efficient graph transformer training system for versatile long-sequence graph learning tasks. The key idea of TorchGT is

to co-design an accuracy-maintained & computation-efficient attention module with a graph-parallelism framework to support large-scale graph training. Experiments show TorchGT can boost training by up to 62.7× with near-linear scalability.

Chapter 5

UniTG: Unified System for Textual Graph Learning in One Step

5.1 Introduction

Graph-structured data has long been prevalent in many real-world scenarios. Among them, a lot of graphs possess textual information, and are often referred to as text-attributed graphs (TAGs) [148]. In TAGs, each node (i.e., graph vertex) is associated with text descriptions such as documents or sentences, and edges represent relationships between these descriptions. For instance, in the ogbn-arxiv dataset [1], a citation network is structured as a TAG, with each node representing a paper and its title and abstract serving as its text attributes. Compared with numerical node attributes, graph topology embedded with textual attributes provides more profound information, significantly enhancing graph representation learning in many tasks such as text classification [60, 82, 149], node classification [59], social networks and recommendation systems [83]. Thus, the representation learning of TAGs has become a significant research area.

Graph Neural Networks (GNNs) exhibit impressive performance on many graph learning tasks [18, 24, 96, 97] and have become a standard methodology in this field [3, 150]. However, in most works, GNNs learn graphs only with numerical attributes as inputs, or handle text attributes by transforming them into shallow or hand-crafted features, such as skip-gram [151] or bag-of-words [152] features. These shallow embeddings are convenient for usage, but are limited in the text semantics they can capture, and fail to incorporate graph properties. To overcome this deficiency, recently, language models (LMs) are raised as a promising method for encoding textual attributes. A new branch of LM-based approaches called *LM4Graph* are emerging, whose architecture is shown in Figure 2.3(a). It consists of two phases. The first phase leverages pre-trained LMs, such as BERT [45], to generate deeper embeddings that specifically

Methods		pubmed Test Acc.(%)↑	ogbn-arxiv Test Acc.(%)↑	ogbn-products Test Acc.(%)↑
GNN	GCN	76.06	71.55	70.01
LM4Graph	GIANT	84.19	73.39	73.89
	TAPE	94.31	75.23	80.37

TABLE 5.1: LM4Graph methods outperform classical GNNs on TAG-realted tasks by a large margin.

capture the semantic richness of text attributes. In the second phase, GNNs learn structure relationships between nodes on the generated attributes. Due to the great modeling capability on both texts and graph structures, LM4Graph has garnered surging interest in recent years and a large number of approaches have been proposed [59, 60, 82–85]. By integrating textual semantics, LM4Graph methods exhibit competitive performance and outperform traditional message-passing GNNs (e.g., GCN [4]) on many TAG-related tasks, as shown in Figure 2.3(b). However, we argue that state-of-the-art LM-based methods are still inefficient and laborious in practice, as they suffer from several fundamental issues:

- **I1: Inefficient learning time and hardware utilization**. Current LM4Graph paradigms [59, 60, 82–85, 88] usually adopt the decoupled training pattern which separately conducts the LM-based embedding generation and GNN training. For example, in each step, GLEM [82] first fine-tunes an LM to obtain the encoded features, and then feeds them into a GNN model to obtain final predictions. As shown in Figure 5.3(a)-(c), one needs to go through two distinct training processes sequentially, leading to substantial cumulative learning time. Moreover, in this one-at-a-time paradigm, the imbalanced workloads across two phases can lead to suboptimal hardware utilization. For instance, the LM phase may require 8 GPUs, most of which will be idle during the GNN phase, as it only needs 2 GPUs.
- **I2: Laborious and inflexible designs**. Current LM4Graph designs heavily rely on human labor and expertise knowledge to determine the proper organizations between LM and GNN phases [82, 83, 91, 148] and suitable feature encoding strategies [29, 59, 85, 88], resulting in high integration cost on new TAG tasks. On the other hand, many works [82] require frequent checkpoint-reloading and program cold-start for switching phases, which not only introduce unnecessary overhead but also potentially incur uncertain bugs. These burdensome hand-crafted features are neither flexible for practical deployment nor portable for model generality.
- **I3: Poor scalability and generality**. Due to versatile self-defined learning procedures, current approaches [59, 60, 82, 84, 85, 88] meet the scalability difficulty when learning efficiency is required. For instance, some works [59, 60, 85] adopt data parallelism for LM phase while keeping GNN phase original on a single GPU, making

it arduous to scale both at the same time. They also design specific text encoding mechanisms exclusive to themselves. This greatly limits the GNN models and TAG tasks one can apply to very small scale [82, 83, 85], conflicting with the development of versatile graph representation application scenarios. Those speific designs hinders the generality of graph learning to other AI domains, including vision, language, and recommendation, to build unified AI systems.

- **I4: Agnostic to graph properties**. To accelerate learning, some solutions [59, 60, 82, 153] simply graft existing parallelism strategies, such as data parallelism to the LM phase, without considering graph properties. This graph-agnostic way remains the computation burden, missing optimization opportunities. Others like [85] reduce the computation cost of text encoding via efficient fine-tuning in natural language processing (NLP) era, e.g., LORA [154]. However, simply adopting them fails to consider the inherent graph structure and results in subpar model performance.
- **I5: Lack of system optimizations**. GNNs rely on massive sparse operations with highly irregular memory accesses, which challenges the optimization of the system throughput. Moreover, with large GNN models, the memory consumption of model grows rapidly, necessitating a scalable system design and memory optimization. Although there have been plentiful system breakthroughs for deep neural networks in scalable training [15, 155–158], those optimizations cannot be directly grafted on LM4Graph approaches due to the inconsistent graph learning pipelines. Those pipelines hardly align with popular distributed frameworks.

To bridge these gaps, we design UniTG, the ***first*** unified and efficient system for textual graph learning in one step with joint optimizations. Our system abides by four design goals: ***efficient, seamless, accurate*** and ***portable***. Existing LM4Graph works neither facilitate efficiency by well-designed parallelism from the system perspective nor support flexible extensions to general models and datasets, thus making it challenging to meet those goals. Specifically, the core design of UniTG derives from the following three key insights. *First, bubble intervals in GNN pipelines can be leveraged for other training jobs for better efficiency.* Pipelined GNN training occupies the majority of resources, which incurs the starvation of other jobs and resource waste. However, it enables us to take advantage of the recurring spare resources for higher hardware utilization and holistic learning efficiency. *Second, the fine-tuning of LMs on graph data benefits from graph properties.* The LM learns the text embeddings of all graph nodes, while each node actually participates differently in GNN computation. Some nodes are more isolated with few connections with others, thus contributing less to GNN message passing [18, 72]. Thus, embeddings of isolated nodes can be less "deep" for higher tuning efficiency. *Third, the intra-layer GNN computation can*

be further boosted by data affinity. The sparse computations in the GNN aggregation phase are one bottleneck for efficiency. There naturally exist many localities in graphs, which can be exploited to relieve the heavy memory access overhead.

As such, our key objective is to design a compute-efficient and accuracy-maintained system from both system and algorithm perspectives to support portable textual graph learning. In detail, UniTG consists of three key components. **Affinity-aware Flow Parallelism** partitions the message flows of GNN model across workers and updates each subset model according to the affinity nature of graphs, thus enhancing system scalability and reducing the irregular memory access latency. **Dual-modality Collaborative Learning** fuses the learning of both text- and graph-modality by regularly synchronizing the encoded text attributes from the LM to the GNN phase, which maintains the model quality to the greatest extent. It also speeds up the fine-tuning of LM with node centrality, a critical graph property. To fully exploit the hardware resources, **Streamlined Pipeline Schedule** interleaves the fine-tuning of the LM into the idle time intervals on each worker known as bubbles, incurred by the pipeline-enabled GNN phase. It automatically manages the execution settings to streamline the original decoupled process, save energy consumption, and provide portable generality to other LM-based learning paradigms. Besides, it enables us to take advantage of the recurring spare resources for better hardware utilization and holistic learning efficiency.

Through extensive experiments, we show UniTG successfully realizes both time- and resource-efficient LM-based graph representation learning. It substantially outperforms prominent baselines by up to 17.3$\times$ on total training time while maintaining model accuracy. Besides, our ablation studies demonstrate that interleaving bubbles with LM tuning can further unleash hardware resources without sacrificing the throughput of the pipelined GNN phase. In summary, we make the following contributions:

- ★ UniTG is the *first* unified system for *LM4Graph* learning in one step. Inspired by the unique features of LM4Graph, UniTG exploits the bubble resources to improve hardware utilization and holistic system efficiency.
- ★ We summarize the current challenges and identify graph-specific opportunities for optimizing both LM and GNN phases, including guiding fine-tuning with node centrality and computing nodes based on graph affinity.
- ★ We demonstrate the excellent performance of our novel LM-GNN training paradigm across large GNN models with accuracy maintained.

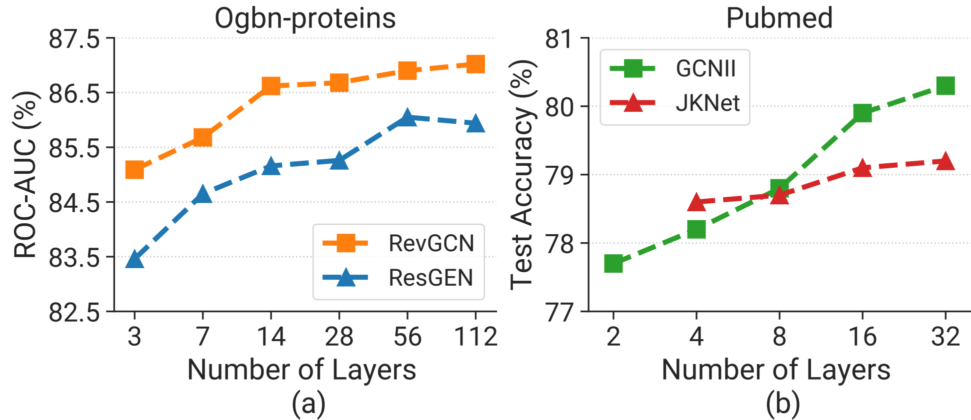


FIGURE 5.1: The test score of GNNs when trained with different numbers of layers. Deep layers show better performance.

5.2 Motivation and Opportunities

5.2.1 Large Models for LM4Graph

Large LM. Existing LM-based graph learning methods all adopt particularly small pre-trained LMs (around 100∼300 million model parameters) for encoding text attributes. Unlike long context texts in the NLP domain, the graph texts usually have limited tokens for each node. Thus, a fine-tuned small LM already provides enough and distinguishable feature space [60, 82, 85] in the graph scenario. On the other hand, the learning effect of different LMs [59, 85] turns out to be comparable. This validates that LM-based graph learning is insensitive to variations in LM selections. Therefore, in this paper we just consider small LM models which suffice to offer good performance on text encodings.

Large GNN. Most GNN models [3, 4] achieve their best performance with 2 or 3 layers, but such shallow architectures limit their ability to extract structural information from high-order neighbors. Also, real-world graphs can be intricate and easily involve millions of nodes [1, 2], on which shallow GNNs show subpar performance [159, 160]. Recent studies [9, 31, 102, 125, 161, 162] demonstrate that with sophisticated architecture designs, deep GNN models are able to achieve state-of-the-art (SOTA) performance over shallow ones on multiple applications [18, 51, 163, 164] such as point cloud segmentation [165, 166] and node classification [102, 167].

For better illustration, Figure 5.1 shows the impact of model layers on the test score of several representative GNNs on two datasets. All models show superior performance on deeper layers, and the power of large depth even becomes more evident on a larger graph dataset ogbn-proteins: ResGEN with 112 layers improves the test score

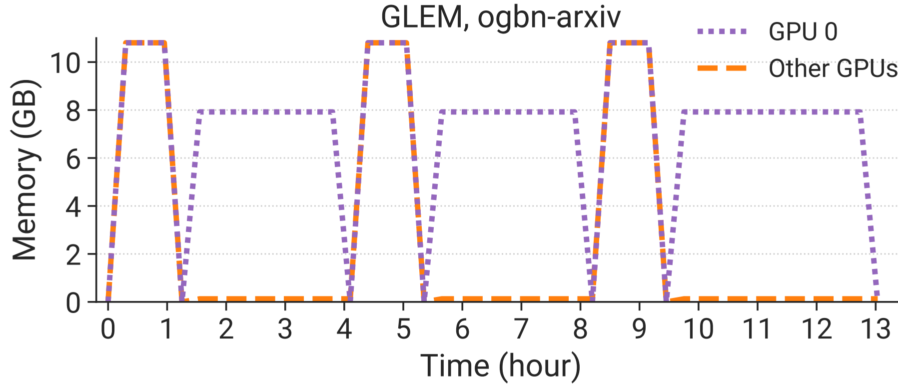


FIGURE 5.2: The profiled time and memory consumption of a LM4Graph method on different GPUs.

by up to 2.5% compared to the shallow models. On the Pubmed dataset, the 32-layer GCNII also outperforms the model with 2 layers by 2.6% of accuracy. These results prove the necessity for deep GNNs of LM4Graphs. Deep GNNs show superior representation power on graphs because of their increased receptive field. However, it is noteworthy that existing LM4Graph methods [59, 60, 82, 84, 153] have some inherent constraints in deploying deep GNNs. Training deep GNNs on current LM-based paradigms requires heavy memory demand and prohibitive training time, which seriously deteriorates the overall learning efficiency. Besides, simply grafting current methods on deep GNNs results in a substantial waste of resources. Our UniTG strives to include larger models by joint algorithm-system co-design.

5.2.2 Opportunities for Efficient Graph Learning

Time-sharing of Two Phases. Current LM4Graph methods mainly focus on the sequential or alternative architecture. In either case, they employ a decoupled training pattern, where the LM-based embedding generation and downstream GNN training are individually conducted one-at-a-time. For instance, a recent work GLEM [82] organizes text encoding and graph aggregation into an iterative workflow, where at each step it fixes one component while updating the other one. Despite its effectiveness in improving the graph learning precision, the learning makespan increases severalfold and GPUs are underutilized due to the iterative mode. To show this, we conduct an experiment to record the time and memory consumption of GLEM in Figure 5.2. We find only GPU 0 (purple line) continuously holds LM and GNN phases in memory and has at least 8 GB of memory usage. Except for GPU 0, other GPUs are only active when the LM phase recurs, remaining idle during the GNN phase. This underutilization is exacerbated by the repetitive training rounds in the figure, leaving other GPUs at low utilization.

Model	Epoch Time(s)		
	RevGCN	RevGAT	ResGNN
Original Training	2.59	0.89	0.26
Affinity-aware Training	2.39	0.83	0.23

TABLE 5.2: Epoch time of training GNN in affinity-aware way and the original counterpart on ogbn-products.

Nonetheless, the unique features of LM4Graph methods also bring opportunities for more efficient learning. (1) *Minor LM portion.* Unlike single model training, in our scenario, the LM phase usually occupies smaller proportion of the total learning time, making it more tolerant to partial throughput slowdown. Therefore, we can ignite the LM phase earlier and shorten the makespan by leveraging the long-term bubble resources of pipelined training coexisting in the cluster. This extends the computing resources for the LM phase in a time-sharing execution way, without hurting the training throughput of the original job. (2) *Decreasing resource demand.* We design a Lazy Tuner as described in §5.3.3, which allows the LM to exit early at the middle layers for certain inputs. Therefore, the early-exit LM usually requires more resources at the beginning and gradually uses a smaller amount of computation [168, 169]. Near the end of an epoch, only a few layers are exploited. Based on this, there will be less kernel competition for GPU resources and we can exactly utilize the bubble GPU resources to run more LM encodings.

Graph Property Awareness. Although LMs have been novelly applied to conduct text encoding for TAGs, current methods [29, 59, 60, 82, 85] typically regard them as general LM workloads in NLP without graph-specific designs. The LM equally handles the textual information of all nodes and treats them with no difference, going through the same deep encoding process. But nodes actually contribute differently to GNN aggregation where they combine messages from their neighbors [18, 29, 72]. The GNN phase also fails to consider graph topology on plentiful sparse operations like the sparse-matrix dense-matrix multiplication (SpMM). The neglect of graph properties may miss pivotal optimization opportunities. Graphs innately own two desiderata for our graph learning scenario: (1) *data affinity*, and (2) *node centrality*. Taking property (1) for illustration, Table 5.2 shows the impact of training a GNN on graphs with more affinity. We can observe that affinity-aware training leads to a certain degree of acceleration (up to 1.13×). This implies nodes with more affinity in a graph can be clustered to improve the data locality in computation, enhancing the computational performance of sparse operators [29, 107, 170]. Therefore, inspired by the benefits of unique graph properties, UniTG implements an affinity-aware computation in the GNN phase to boost the aggregation execution, and a centrality-guided tuner for LM fine-tuning to adaptively exit from the pre-trained LM in advance based on different

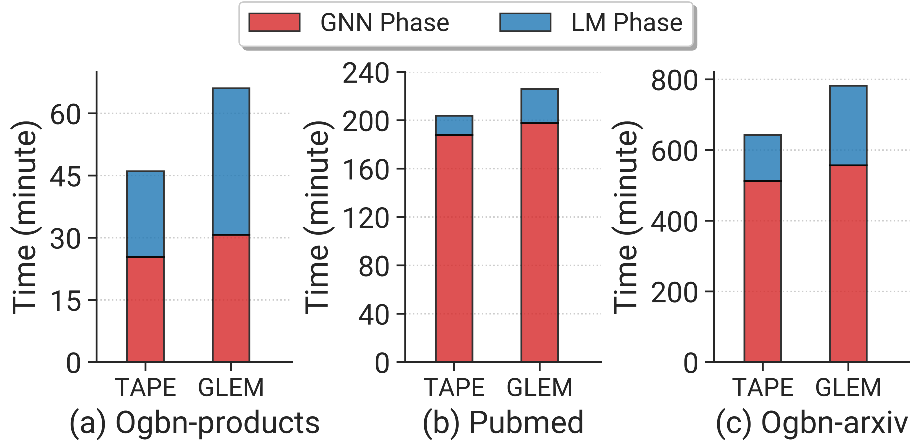


FIGURE 5.3: Time breakdown of each phase when training different methods on 3 datasets on GPU-A (details in §5.4).

input attributes. Graph-specific optimizations can be applied to both phases at the same time owing to models in LM4Graph sharing the same graph data and learning target.

Interactive Training Fusion. As mentioned before, existing LM4Graph methods all adopt the decoupled training pattern, which renders the learning inefficient because of tedious and repetitive processes. To understand the time bottleneck of current methods, we conduct experiments to record the training time summation of the LM and GNN phases respectively. Figure 5.3 shows the detailed training time breakdown of two representative LM4Graph methods, TAPE [59] and GLEM [82], on three datasets, with LM and GNN backbones being DeBERTa-base [46] and RevGAT [164]. We can see the training time even prolongs to 13 hours on the ogbn-arxiv [1] dataset in Figure 5.3(c). In lengthy training cases (Figure 5.3(b, c)), the GNN phase occupies over 70% for both methods. The long training time of either method originates from the independent pattern, frequent checkpoint-reload, and redundant cold-start mechanism of phase switching. However, we find it is feasible to realize continuous and coherent training in a cold-start-free way. Specifically, instead of waiting for the completion of LM tuning, we extract the encoded embeddings periodically from the partially tuned LM and feed them into the concurrent GNN task. According to our evaluation results on runtime accuracy (later shown in Figure 5.12), the intake of encoded embeddings for GNNs can be brought forward along with LM tuning without hurting the model convergence, thus enabling GNN training to be interleaved with LM fine-tuning as well.

5.3 System Design

5.3.1 UniTG Overview

Motivated by all observations in §5.2.2, we propose UniTG, an algorithm-system co-optimized system tailored for portable LM-based graph learning coherently in one step.

Principles & Goals. For simple and efficient system adoptions, UniTG follows three design principles: (a) *Efficient and accuracy-maintained.* The system greatly improves the graph learning efficiency from both time and hardware perspectives for LM and GNN phases as a whole, supporting more powerful GNN models. Meanwhile, it maintains comparable final accuracy and training convergence to the baselines (solving **I1,3,5**). (b) *Seamless and portable.* Instead of manually launching two separate phases and determining the feature encoding manner, our system accomplishes the graph learning in just one step, enabling two phases to update simultaneously and mutually. Moreover, the whole workflow is automated and easy to use, which requires minimal human effort for program initiation and checkpoint checking (solving **I2**). (c) *Modular and specialized.* Each component in UniTG works independently and thus can be applied to existing algorithms non-invasively. UniTG is tailored for LM4Graph methods, considering unique graph properties. Thus, it supports large GNN models which enhance accuracy, and more TAG tasks can be easily integrated (solving **I4**). Our primary objective is to minimize the makespan of LM4Graph learning. Apart from this, UniTG also improves resource utilization, reduces energy cost and maintains model quality.

System Architecture. Figure 5.4 shows the architecture and workflow of UniTG. It intelligently optimizes training across three levels from the top to bottom hierarchy: runtime, algorithm and execution. In the runtime level, we propose a novel parallelism considering graph affinity to scale the GNN phase. Then we simplify the LM tuning and fuse the text- and graph-modality learning interactively at the algorithm level. During the execution, we design a time-sharing plan to fully utilize resources. Specifically, UniTG consists of three innovative components:

- *Affinity-aware Flow Parallelism*: From the distributed runtime perspective, we design a Flow Parallelism (FP) tailored for GNNs to split the model across workers at the inter-layer level. On the other hand, with graph affinity, we can boost the intra-layer computation by alleviating irregular memory access.
- *Dual-modality Collaborative Learning*: At the algorithm level, we integrate the training of two modalities, text and graph concurrently as a whole with *Modality Collaborator*. It periodically fetches the encoded text embeddings from the LM to

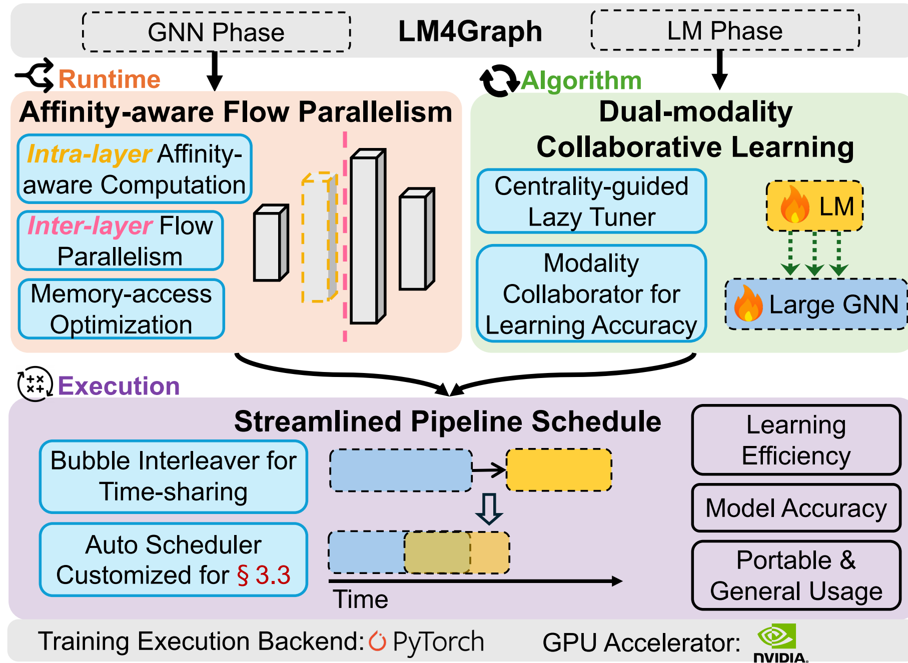


FIGURE 5.4: Overview of UniTG architecture and workflow.

mutually enhance the GNN phase while maintaining final accuracy. In addition, a *Lazy Tuner* is proposed to safely accelerate the LM fine-tuning, enjoying the advantages of node centrality.

- *Streamlined Pipeline Schedule*: It consists of a *Bubble Interleaver* which inserts the LM tuning into the bubble intervals of pipelined GNN training so as to realize the time-sharing computation. We also build an *Auto Scheduler* customized for collaborative learning to adjust the pipelining settings and support extensions to different LM4Graph approaches.

5.3.2 Affinity-aware Flow Parallelism

Targeted on the GNN phase, to better fit large GNN models and increase the system scalability, we introduce a novel parallel training style specialized for LM4Graph methods, which exploits the graph affinity characteristics to boost the distributed GNN training.

Graph Affinity. Graph affinity [143, 144, 170] is an important characteristic of real-world graphs, referring to a subset of nodes in the graph that exhibit a higher degree of connectivity (more edges) while keeping few connections with nodes in other parts of the graph. SpMM serves as a foundational yet computationally intensive kernel operation that underlies many GNN implementations, which requires memory-intensive computation. This is because the sparse and highly skewed nature of graphs incurs

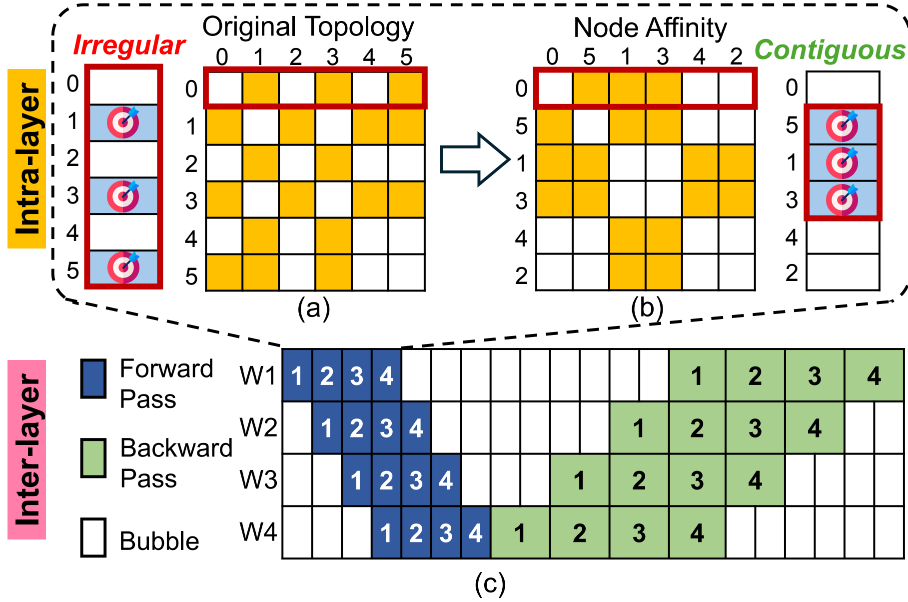


FIGURE 5.5: Design of Affinity-aware Flow Parallelism. (a) Original graph topology, (b) affinity-aware topology and (c) Flow Parallelism for a GNN model with 4 pipeline stages and 4 graph microbatches.

uneven edge connections. To reduce storage overhead, the matrix representation arranges the non-zero elements (nnz) in a row-major sequence. However, their column indices are widely dispersed [171], leading to non-contiguous memory accesses when referencing the associated dense matrix. Data affinity, in this case, improves the number of nnz in a local area, which enhances the computational performance for SpMM operations. Furthermore, improving data locality is essential for effectively exploiting the hierarchical caching mechanisms available on GPUs. Consequently, employing the data affinity on GNN aggregation is promising for training improvement. There exist some approaches [144, 145] in traditional graph learning that also utilize the affinity nature, but they aim for CPU processing with limited parallelization. GNNAdvisor [107] similarly exploits graph clusters but focuses on redundant data loading in GNN computing.

Intra-layer Affinity-aware Computation. In light of the above, we exploit the performance benefits of graph affinity for intra-layer GNN computation as shown in the upper part of Figure 5.5. Specifically, to improve spatial locality, we apply a lightweight reordering approach that splits the sparse matrix into compact clusters and rearranges them, ensuring the original connectivity remains intact. Figure 5.5(a) is the adjacency matrix visualization of the original graph, which shows an irregular pattern (yellow blocks). For example, the update of node-0 depends on node-1,-3, and -5 stored in discontinuous memory. In contrast, with affinity-aware node reordering, the reordered graph is shown in Figure 5.5(b) and get better data density with more locality in the adjacency matrix. Proximate node identifiers tend to be allocated to

neighboring GPU computing units, where their processing takes place. In this case, the system can benefit from locality since nodes in each cluster are close to each other. Now we can hit the feature memory of the 3 nodes in contiguous indexes and thus improve the computing efficiency in each GNN layer.

As for the reordering method, we leverage METIS [58], a community-based graph reordering technique for great cluster locality and ease of integration with parallelism. Specifically, it uses multilevel recursive bipartitioning to divide and coarsen the graph while preserving the essential structure. Then, it applies partitioning algorithms to find an optimal node ordering that reduces the cost of accessing neighboring nodes. We optimize the implementation of METIS for a lower cost: we capture the cluster information of graphs and map such locality from the upper level to the underlying GPU kernels, which also enables us to leverage the L1 & L2 cache for refined cluster capturing. After all nodes have been reordered, we get the reordered topology in Figure 5.5(b).

Inter-layer Flow Parallelism. To increase the scalability of LM4Graph methods and support large GNN models, intuitively we split the message flows between GNN layers and dispense the computation across devices. Current LM4Graph methods only adopt distributed training for the LM phase and leave the GNN phase in a primitive status, resulting in present fragmented learning patterns and difficulties in optimizations. There have been extensive studies in parallelism technologies [14–16, 157, 158] for LLMs (e.g., GPipe [156]) to support large model training. However, simply grafting current parallelism methods on this graph learning paradigm triggers two challenges: (1) *not supporting graph inputs*; (2) *deteriorating model accuracy*. In traditional LMs, the input is only a sequence concatenating tokens in a sentence without 2D structures. In contrast, GNNs rely on graph topology for each model input and intermediate computation. In addition, current model parallelism methods further split batches to numerous microbatches for higher efficiency, which is non-trivial for GNNs since over-segmentation on graphs loses connectivity and hurts model accuracy substantially.

Based on these challenges, we design an Inter-layer Flow Parallelism ***specialized for LM4Graph methods***. Figure 5.5(c) depicts the detailed workflow of computing 4 microbatches on 4 devices. Considering a L-layer GNN model, UniTG partitions the message flows between GNN layers into P subsets of consecutive model layers, or P stages, and places them on P workers respectively. Then each worker just holds L/P layers. We utilize the point-to-point communication operations at each layer boundary to transfer the intermediate embeddings in the forward pass and gradients in the backward pass. During the forward pass (blue blocks), the Flow Parallelism first splits the original graph with N nodes into M subgraphs, referred to as ***micro-batches*** hereafter, by considering the affinity nature. Thus, few edges are cut and

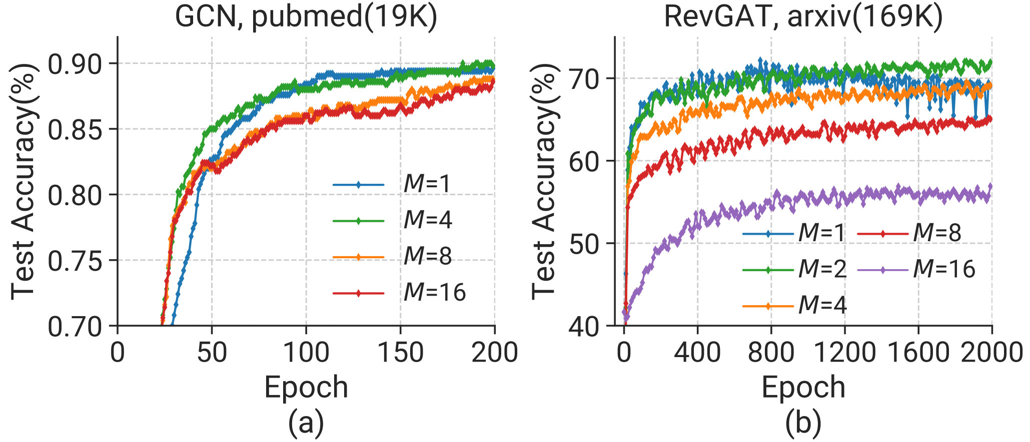


FIGURE 5.6: The test accuracy of GNNs in Flow Parallelism when trained with different numbers of microbatches M. Large M results in degraded model accuracy.

data locality is still maintained in each microbatch, thus preserving training quality. This will be discussed elaborately later in the design of Auto Scheduler (§5.3.4). Then these microbatches are pipelined across P workers and the computation of different microbatches is overlapped to shorten bubble time. In the backward pass (green blocks), gradients of the L/P layers are computed and accumulated regularly from microbatches to update the sub-model parameters.

Note that here we adopt the F-then-B strategy in GPipe [156], instead of using the common 1F1B strategy [157, 158, 172], since *GPipe better fits the memory and time optimizations when the number of microbatches is small.* To explain this in detail, Figure 5.6 shows the test accuracy of training GNNs in the Flow Parallelism with different numbers of microbatches. Clearly, there is a tendency for more microbatches to lead to worse model convergence and downgraded accuracy (e.g., orange and red curves). For example, there is a severe accuracy drop of up to 18% for RevGAT on 16 microbatches compared to that on 1 or 2 microbatches. Therefore, the number of microbatches applicable for pipelined GNN should be limited to a small scale, which is typically smaller than or equal to the parallelism degree ($M \leq P$). However, the power of 1F1B strategy is only unleashed when there exist massive microbatches to be pipelined simultaneously [156, 173, 174]. When $M \leq P$, F-then-B shares comparable parallelism efficiency with 1F1B. In addition, F-then-B also surpasses in more balanced memory consumption per worker. Thus, to better explore resource optimizations and match the LM phase, we adopt the F-then-B strategy in UniTG, whose bubble resources will be coped with in §5.3.4.

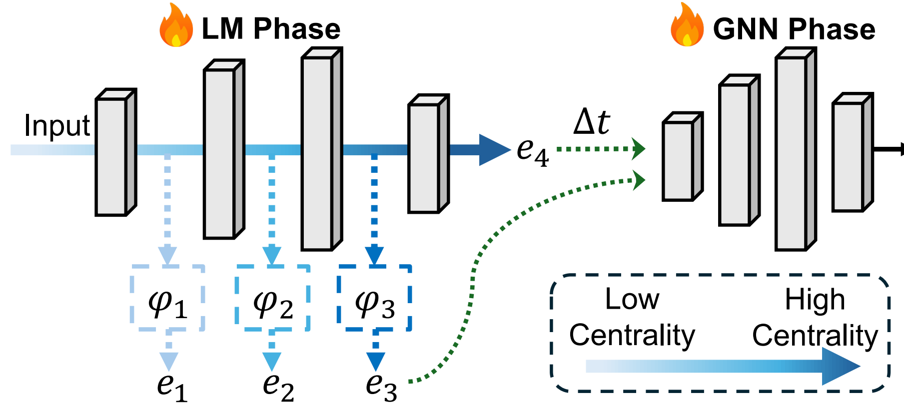


FIGURE 5.7: Illustration of the proposed Centrality-guided Lazy Tuner, and how it interacts with the GNN phase. Each grey bar represents a sequence of model layers. Both phases are trained in the collaborative learning mechanism.

5.3.3 Dual-modality Collaborative Learning

Current LM4Graph methods fine-tune pre-trained LMs such as DeBERTa [46] to generate node embeddings in the context of TAGs. Specifically, an LM is employed to encode the original text attributes associated with each node and learn a representation that extracts the semantic meaning of the text. However, current methods treat the LM fine-tuning in graph learning the same as ordinary tasks in the NLP domain. This is prohibitively time-expensive since the LM is fine-tuned clumsily on all input data, neglecting important graph natures. There have been extensive studies for efficient fine-tuning of LMs by either adding adapter layers [154, 175] or optimizing the input layer activations [176, 177]. However, simply adopting those fine-tuning techniques neglects graph properties and fails to distinguish the LM fine-tuning in the graph domain from that in NLP. Those differences invoke specialized and dedicated system designs for LM4Graph methods towards more aggressive LM fine-tuning and more efficient training paradigms.

Centrality-guided Lazy Tuner. Motivated by the early exiting mechanism [168, 178–180] which shares a similar partial training with the aforementioned fine-tuning methods, we fine-tune the LM by exiting from the middle layers early to generate embeddings. Figure 5.7 shows the detailed mechanism of the proposed Centrality-guided Lazy Tuner. To determine where and how to put early-exit layers, we leverage the benefits of node centrality. Node centrality is a significant graph property that measures how important a node is in the graph. The key insight is that *not every node deserves the same depth of consideration, and one shall opt for fast reaction to marginal nodes without time-consuming overthinking.* Therefore, for graph inputs to the LM, UniTG assigns each node a degree centrality and classifies all nodes into four distinct groups: $\mathcal{G} = \{C_1, C_2, C_3, C_4\}$ based on the centrality per node. Nodes with the lowest centrality (light blue in Figure 5.7) are classified to group C_1, and

those with the highest centrality (deep blue) are sorted to C_4. The model is also evenly divided into 4 parts as exit positions. During the training process, the training of nodes in C_1 leaves earliest from a quarter of the network to an early-exit linear layer $\boldsymbol{\varphi}_1$. Then $\boldsymbol{\varphi}_1$ converts the hidden states into output embeddings $\boldsymbol{e}_1$. The same principle applies to nodes in C_2, C_3, and so forth. After applying the early exit, the final loss is a weighted sum of losses after all early and final layers, which can be formulated as:

$$\min \mathcal{L} = \frac{\sum_{i \in |\mathcal{G}|} \mathcal{L}_i^{\text{exit}}}{|\mathcal{G}|} \tag{5.1}$$

where $|\mathcal{G}|$ is the number of exits, and $\mathcal{L}_i^{\text{exit}}$ is a standard loss function at the i-th exit layer. Early-exit works on LMs [168, 178, 179] prove that the early-exit losses decay at a similar pace to the final-exit counterpart. Though the early-exit losses may be slightly higher than the final-exit loss, they encode less important nodes that play trivial roles in subsequent training. Applying it to UniTG, our Centrality-guided Lazy Tuner can provide robust embeddings for graph learning and obtain convergence comparable to the original one, which is also validated by the convergence curves in Figure 5.11(b).

Modality Collaborator. To convey the encoded embeddings from the LM to the GNN phase, current approaches save the embeddings as checkpoints. Then the GNN task is launched to load them as model inputs. This cold-start way is time-consuming and error-prone, which will be exacerbated when phases alternate and repeat frequently. Therefore, we introduce a new paradigm for coherent and portable graph learning. We fuse two phases' training simultaneously and coordinate them to gradually enhance the learning. To be specific, as shown in Figure 5.7, we initiate both the LM and GNN training at the beginning. During the fine-tuning of the LM, we periodically generate the node embeddings from the last or exit hidden layer of the partially tuned LM at interval Δt. Then we feed the fetched embeddings to the GNN model as input features. For certain nodes whose embeddings have not been encoded when synchronization occurs, we replace them with the primitive numerical embeddings included in the dataset for precision guarantee.

To better ensure the training accuracy, we need to figure out when to deliver the embeddings to the GNN phase. Some works [82, 85] employ similar interactions of enhancing GNN training with LM outputs and prove improved accuracy even on embeddings from pretrained LMs without fine-tuning. Based on this, since the embeddings generated in the early tuning stages are not refined enough to capture the text information, intuitively we can start from less frequent synchronization along the time dimension and gradually increase the frequency. Specifically, we track a Running Average Loss $F_j = 0.8F_{j-1} + 0.2\mathcal{L}_j^{GNN}$ to measure the overall training status of using LM encoded embeddings, where $\mathcal{L}_j^{GNN}$ is the loss of GNN training at epoch j. At

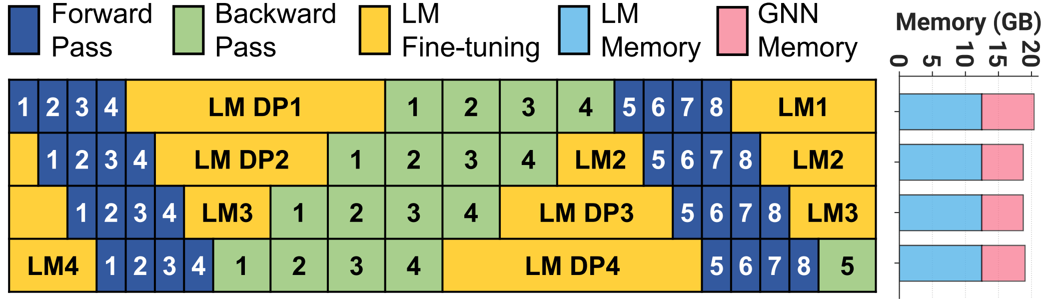


FIGURE 5.8: Illustration of Bubble Interleaver with 4 pipeline stages and 4 microbatches. Right-side is the memory consumption of each GPU.

the beginning Δt is initiated as $0.3J$ where J is the number of epochs for the GNN training. Then it gradually decreases to values in the set $\{0.3J, 0.2J, 0.1J, 0.02J\}$, which is constructed by profiling plentiful cases. When F_j keeps decreasing for 20 epochs, we determine current embeddings can precisely reduce the loss, and UniTG decreases it to the next value in the set to get more frequent synchronization of LM-generated embeddings. Otherwise, the embeddings are not refined enough, and we keep Δt unchanged to enable more stable training. The final accuracy in Table 5.5 indicates the effectiveness of our Modality Collaborator.

5.3.4 Streamlined Pipeline Schedule

This module focuses on execution-level optimizations, including a Bubble Interleaver to improve resource utilization in a time-sharing way, and an Auto Scheduler customized for the Dual-modality Collaborative Learning to adjust training settings and support general usage.

Bubble Interleaver. The Flow Parallelism can significantly reduce the GNN training time, but also starve faster stages of microbatches to run and thus result in resource under-utilization. As shown in Figure 5.5(c), the inter-layer values (activations and gradients) of the partial model are transferred at the forward and backward passes between different workers. In this regime, the backpropagation requires each GPU to stall for gradients from the latter GPUs, thus incurring massive bubbles and failing to continuously utilize the GPU. For worker 1, after the forward pass of the final microbatch (blue block 4), it keeps waiting since gradients of microbatch 1 have not been produced yet (green block 1), leaving the GPU idle for a long time. Other workers also present similar bubble patterns.

UniTG designs Bubble Interleaver in Figure 5.8, which leverages bubbles to greatly extend the tuning job resources in a time-sharing way, almost without hurting the training throughput of the pipelined GNN workload. LM4Graph methods are perfectly suitable for the bubble interleaving execution due to the following unique features:

(1) *Mixed parallel workloads.* Unlike common learning jobs, there are two distinct distributed jobs (i.e., LM and GNN) that coexist in the learning process, which provides opportunities for time-sharing computation. (2) *LM throughput insensitivity.* In LM4Graph methods, usually the LM phase consumes less learning time and is further accelerated by our Lazy Tuner, thus being tolerant of partial throughput slowdown. (3) *Reduced resource demand.* Lazy Tuner instructs the LM to exit from earlier layers along the training. Therefore, the resources needed for LM computation gradually reduce and the bubbles can be utilized for more LM tuning. All the above inspires us to leverage the spare resources of the bubbles and execute the LM phase in a pause-and-resume way.

In Figure 5.8, UniTG interleaves the 4-stage GNN Flow Parallelism with data-parallel LM fine-tuning. Since the LMs are relatively small, data parallelism (DP) or single-GPU training is sufficient for efficient LM fine-tuning. Here we depict LM tuning with DP across 4 GPUs for illustration, with each DP process (denoted as LM DP1∼4) running in FP bubbles accordingly. Note that UniTG also supports interleaving the single-GPU task. Each LM process executes in a pause-and-resume mechanism to leverage the bubbles. To be specific, we register hooks on each operation of the LM to promptly control the pause and resumption in the forward and backward passes of each layer. At the beginning, both LM and GNN phases are started, and UniTG identifies the idle resources in FP bubbles. For the pipelined GNN, our Affinity-aware Flow Parallelism also registers hooks inside each pipeline stage to monitor its training progress and resource consumption. To interleave the LM DP process correctly into bubble intervals, Bubble Interleaver checks the status of the CUDA streams of the NCCL kernels to ensure the LM process only runs within the bubbles. We realize this by sending self-defined signals to control the resumption and suspension of LM processes at the beginning and end of bubbles. By monitoring the underlying stream status, this scheme refrains from interference of CUDA kernels and introduces almost no overhead to GNN training. We also profile the memory footprints with Bubble Interleaver on the right side of Figure 5.8. The memory footprints of LM and GNN phases have already been reduced by the Flow Parallelism and data parallelism respectively, so there is no need to swap out the interleaved memory during colocation, and the GPU utilization is also greatly improved.

Auto Scheduler. Different from typical hybrid parallelism methods [181–183], the parallelisms in our scenario are innately independent and under distinct workloads, which avoid complicated strategy design and can be better scheduled by UniTG. To this end, we design an Auto Scheduler to balance the efficiency-quality trade-off and accommodate various LM4Graph methods to deliver the best performance.

(1) *The number of microbatches M.* Different from the LLM case, the number of microbatches in graph learning is highly limited, as discussed in §5.3.2. The training

efficiency improves if the number of microbatches increases, while the model quality downgrades due to a greater drop of connectivity information. To better determine this value, we profile and record the test scores of training different GNNs w.r.t. different M values like Figure 5.6. We find that all cases suggest $M < 4$ gives ideal performance, with smaller graphs being more tolerant of more microbatches. Therefore, given the Flow Parallelism degree P and graph size N, empirically we can summarize: $M = \{16 : N < 10^4, 8 : 10^4 \leq N < 5 \times 10^4, 4 : N \geq 5 \times 10^4\}$ & $M \leq P$. This strategy maximizes training efficiency while maintaining model accuracy.

(2) *Scenarios where UniTG performs best.* The effectiveness of Bubble Interleaver varies depending on multiple factors (e.g., the scale of LM and GNN training) and we present the most affected ones. (i) *Higher parallelism of both phases.* More stages of a pipelined GNN imply a higher bubble ratio and more spare resources, which typically can interleave the LM tuning on each device and accelerate dozens simultaneously. In this case, the same size of DP for the LM tuning is also a perfect match for maximum resource utilization and overall acceleration. (ii) *Tasks with similar training time* are preferred. Too large LMs of up to 50 billion parameters fail to be allocated on any bubble resources and are memory-consuming in both model and activation size. On the other hand, too small LM models (e.g., 20 million parameters) terminate far earlier than GNN training and are unable to fully utilize the hardware resources. As a result, LMs of middle size like DeBERTa [46] are ideal choices for the LM phase.

5.4 Evaluation

We implement UniTG atop PyTorch 2.1 [117] with about 23K LoC. The point-to-point communication is implemented via `torch.distributed` with the NCCL backend. The Centrality-guided Lazy Tuner and Modality Collaborator rely on HuggingFace [184]. We evaluate the performance of UniTG from the following aspects: (1) efficiency, (2) model accuracy, (3) power consumption and (4) micro-benchmarks and ablation studies to examine the impact of each module independently for a fair comparison.

Testbed. Our experiments are performed on two NVIDIA GPU servers with the following setups: ❶2 servers each with 8 *type A* GPUs (24GB). Intra-server connections (CPU-GPU, GPU-GPU) are based on PCIe 4.0x16 lanes and inter-server connections are via 1Gbps Ethernet. ❷2 servers each with 8 *type B* GPUs (96GB) with NVLink and 3.2Tbps InfiniBand.

Datasets and Models. We evaluate UniTG on versatile real-world graph datasets with multiple scales. The detailed information is shown in Table 5.3. For fair comparisons with baseline TAPE [59], we conducted experiments on a subset of the ogbn-products dataset the same as TAPE. We use three large GNNs commonly adopted

TABLE 5.3: Detailed information of evaluated datasets, where ogbn-products* stands for a subset of the original dataset.

Datasets	# Vertices	# Edges	# Features	Task	Metric
ogbn-arxiv	169,343	1,166,243	128	40-class classif.	Accuracy
ogbn-products*	54,025	74,420	100	47-class classif.	Accuracy
pubmed	19,717	44,338	500	3-class classif.	Accuracy
cora	2,708	5,429	1,433	7-class classif.	Accuracy

TABLE 5.4: Hyperparameters for GNNs. For cells with two values, the first value corresponds to the first dataset and the rest are similar. Otherwise datasets share the same value.

Models	ogbn-arxiv, -products			pubmed, cora		
	# Layers	# Hidden	Epoch	# Layers	# Hidden	Epoch
RevGCN	112	224	1000, 500	448	80	1000
RevGAT	20	256	2000, 500	40	768	1000, 500
ResGEN	56	128	2000, 1000	112	64	500, 1000

as the GNN backbone, including RevGCN [102], RevGAT [164], and ResGEN [163]. For the LM backbone, we exemplify DeBERTa [46] in evaluations. Note that UniTG can be also applied to other LM4Graph approaches with versatile LM and GNN backbones. Some critical hyperparameters are listed in Table 5.4. We refer to the configurations reported in the original papers.

Baselines. Due to the lack of existing LM4Graph systems, we adopt popular algorithms in their original implementations as baselines for comparisons. (1) GLEM [82]: a method that trains the LM and GNN alternatively with many rounds. (2) TAPE [59]: leveraging an LLM to capture textual information, which is then translated via the LM several times to get different encodings. Both baselines only implement one GNN, so we meticulously develop other GNNs on them as well.

5.4.1 End-to-end Performance

Learning speedup. We compare the end-to-end total training time and test accuracy of UniTG with all baselines on one server, as shown in Table 5.5. Since LM4Graph learning typically includes two phases and each baseline has repetitive training sessions on the same phase, we record and compare the total training time of each method. The number of stages for GNN FP is 4 and so is the degree of LM DP. The speedup in brackets is the relative throughput of each method on the basis of GLEM. UniTG substantially outperforms all baselines by 4.5$\sim$17.3$\times$. The makespan of GLEM can prolong to prohibitive 23 hours with RevGCN on the ogbn-arxiv dataset. In contrast, UniTG successfully reduces the makespan to only 1.3 hours, which critically facilitates learning efficiency and saves possible resource costs. Moreover, we

TABLE 5.5: Detailed comparison of total time and test accuracy of methods when training on one GPU-A server. OOM means the method runs out of memory. UniTG always outperforms baselines in makespan and achieves comparable accuracy on all models and datasets.

Model	Method	ogbn-products		ogbn-arxiv		pubmed		cora	
		Total Time(s)	Test Acc.(%)	Total Time(s)	Test Acc.(%)	Total Time(s)	Test Acc.(%)	Total Time(s)	Test Acc.(%)
RevGCN	GLEM	7593.20(1.00×)	79.36	81945.19(1.00×)	72.52	9604.11(1.00×)	94.59	5943.11(1.00×)	86.66
	TAPE	6289.28(1.21×)	80.38	71694.70(1.14×)	73.45	8604.29(1.12×)	94.37	5767.62(1.03×)	87.64
	UniTG	1453.84(**5.22×**)	79.34	4732.49(**17.32×**)	72.37	2123.67(**4.52×**)	95.06	1278.38(**4.65×**)	86.53
RevGAT	GLEM	3962.34(1.00×)	79.82	46924.24(1.00×)	74.93	13550.53(1.00×)	94.71	1392.88(1.00×)	87.96
	TAPE	2761.58(1.43×)	79.87	38542.30(1.22×)	76.02	12224.39(1.11×)	94.65	1071.87(1.30×)	87.08
	UniTG	621.83(**6.37×**)	79.05	5347.00(**8.78×**)	75.80	1304.79(**10.39×**)	94.83	277.69(**5.02×**)	87.82
ResGEN	GLEM	OOM	-	OOM	-	2507.30(1.00×)	94.41	804.89(1.00×)	85.42
	TAPE	OOM	-	OOM	-	1645.64(1.52×)	93.94	316.82(2.54×)	86.35
	UniTG	**433.46**	76.55	**4099.88**	72.39	401.04(**6.25×**)	94.73	92.06(**8.74×**)	86.72

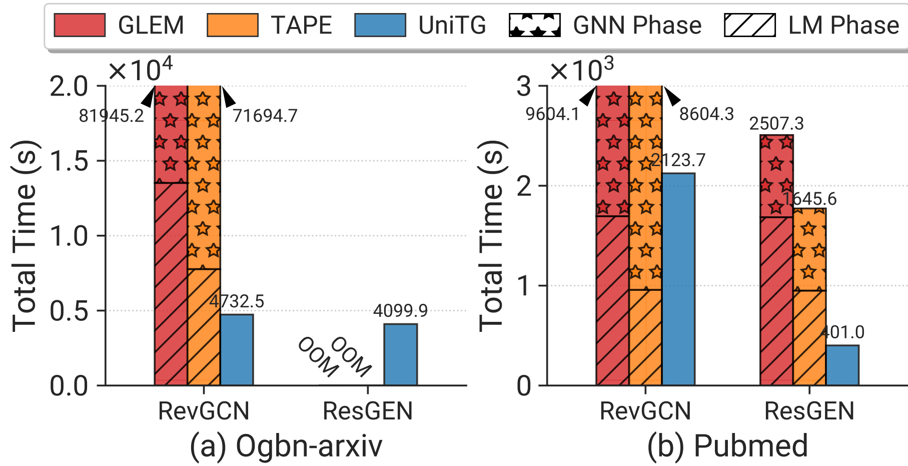


FIGURE 5.9: Detailed time breakdown of LM and GNN phases for all methods on two models. UniTG far exceeds baselines in training time within one integral procedure.

also observe that the effect of UniTG is more evident over larger datasets with more intensive computation. This reflects that UniTG is more suitable for large-scale jobs with limited resources, which is hard to handle by existing methods. In most cases, TAPE offers only a minimal speed improvement compared to GLEM since they both train LM and GNN one at a time. Additionally, the two baselines run out of memory (OOM) on both ogbn datasets on ResGEN, due to their incapability of training large GNNs.

To investigate the specific time consumption of each phase, we record the time breakdown of methods in Figure 5.9. In most cases, the GNN phase dominates over 89% of the total time, which hinders system efficiency and blocks resource usage under the decoupled paradigm. Even when GNN is faster than LM (e.g., RevGCN in Figure 5.9(b)), the total time is still large due to the accumulated time of two phases. In contrast, UniTG directly completes the learning quickly in one step, with a makespan even lower than a single LM phase in baselines (e.g., RevGCN on ogbn-arxiv), thanks to our intra-phase acceleration and inter-phase interleaving.

TABLE 5.6: GPU energy consumption comparisons of UniTG and baselines (units: J).

	ogbn-arxiv		**pubmed**	
	RevGCN	RevGAT	RevGCN	RevGAT
GLEM	27.72M	15.43M	3.04M	4.62M
TAPE	24.49M(1.1×)	12.86M(1.2×)	2.75M(1.1×)	4.20M(1.1×)
UniTG	1.56M(**17.8**×)	1.65M(**9.4**×)	0.69M(**4.4**×)	0.46M(**10.0**×)

Final accuracy. Table 5.5 also summarizes the test accuracy achieved by all methods. UniTG successfully provides comparable model accuracy to baselines, though sometimes slightly falling behind TAPE, which utilizes extra LLM explanations for data augmentation. Surprisingly, we find UniTG can improve the final model quality in some cases. For instance, UniTG always achieves accuracy improvement of up to 0.79% on pubmed, which is mainly due to the data-aware LM tuning. In summary, UniTG is sufficient to provide equivalent accuracy with other complex baselines.

Power consumption. UniTG designs a time-sharing paradigm, which eliminates the decoupled pattern where each phase monopolizes the hardware resources and causes energy waste. In Table 5.6, we compare the total energy consumption of UniTG on each GPU against baselines over several datasets. UniTG demonstrates up to 17.8× and 15.7× energy savings over GLEM and TAPE, respectively. GLEM spends the most GPU power because of its tedious LM and GNN steps for multiple rounds. Our system cleverly fuses two phases' training, dramatically cutting down unnecessary energy costs and promoting sustainable computing for graphs.

5.4.2 Effect of Dependency-aware Flow Parallelism

Training speed w.r.t. different techniques. Figure 5.10(a) illustrates the training throughput of the GNN phase when varying the applied techniques. Obviously, the proposed Flow Parallelism (blue bar) reduces the training time by a large margin of 3× on the basis of vanilla GNN training. Applying affinity-aware reordering on top of it (green bar) further shortens the epoch time by 3.4×, owing to its effectiveness in reducing irregular memory access. UniTG can support large GNNs like ResGEN which suffers OOM issue on baseline methods. Figure 5.10(a) also reveals an interesting observation that UniTG can only achieve very limited improvement over vanilla GNN if FP is disabled. This is because sparse computation dominates in large GNNs and graphs, exhibiting no evident speedup even if we cluster the graph. Therefore, it is important to combine FP and affinity-aware reordering to achieve the desired performance.

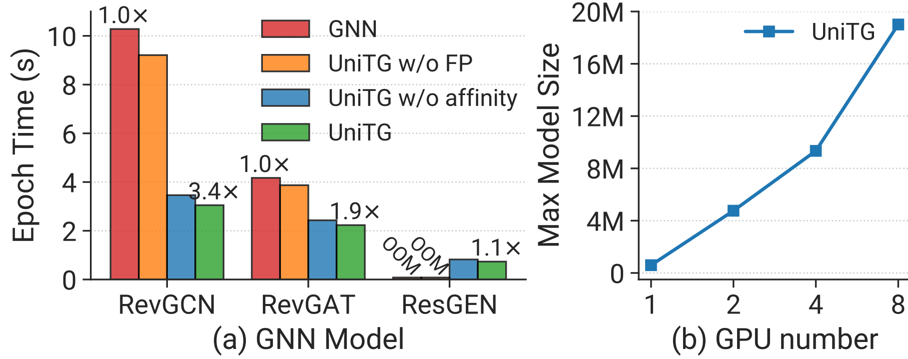


FIGURE 5.10: Ablation study of Affinity-aware FP on ogbn-arxiv on one type-A GPU server. (a) Effect of each technique. (b) The supported maximum GNN size w.r.t. GPU count.

Model size w.r.t. number of GPUs. In Figure 5.10(b), we examine the maximum GNN model size that can be trained on 1∼8 GPUs with UniTG. Since all baselines do not support large GNN training on multiple GPUs, we only plot the results of our system. The maximum GNN size of UniTG can reach to 19 million on 8 GPUs, which is unavailable for baselines employing vanilla training. Moreover, the model size of UniTG almost scales linearly w.r.t. the number of GPUs, indicating a certain degree of system scalability.

5.4.3 Effect of Dual-modality Collaborative Learning

Centrality-guided Lazy Tuner. In collaborative learning, we first investigate the LM phase after applying Centrality-guided Lazy Tuner in §5.3.3. In detail, we explore the effect on tuning time and model quality on one type-A GPU server.

(1) *Efficiency of LM fine-tuning*. Figure 5.11(a) clearly presents the total time of the original LM fine-tuning without and with Centrality-guided Lazy Tuner on three datasets. We record the total time instead of iteration time since Lazy Tuner adaptively adjusts the exit layer, leading to different iteration durations along the tuning procedure. Compared with the raw LM fine-tuning, Lazy Tuner can significantly reduce the tuning time by 3.8∼4.3×. For instance, the original tuning time on ogbn-products is 701s while UniTG reduces it to 165s, validating the efficiency of Lazy Tuner. Due to the page limit, we only select DeBERTa-base [46] as the LM backbone for presentation because of its relatively obvious efficacy of Lazy Tuner.

(2) *Loss curves and convergence*. Since Lazy Tuner only trains a partial model for certain inputs, we record the loss curves of standard and early-exit LMs in Figure 5.11(b). Apparently, the loss curve with Lazy Tuner (blue curve) decays at a faster pace at the beginning, owing to early exits of unimportant nodes from the network. Unsurprisingly, when approaching the end of training, the early-exit losses are slightly

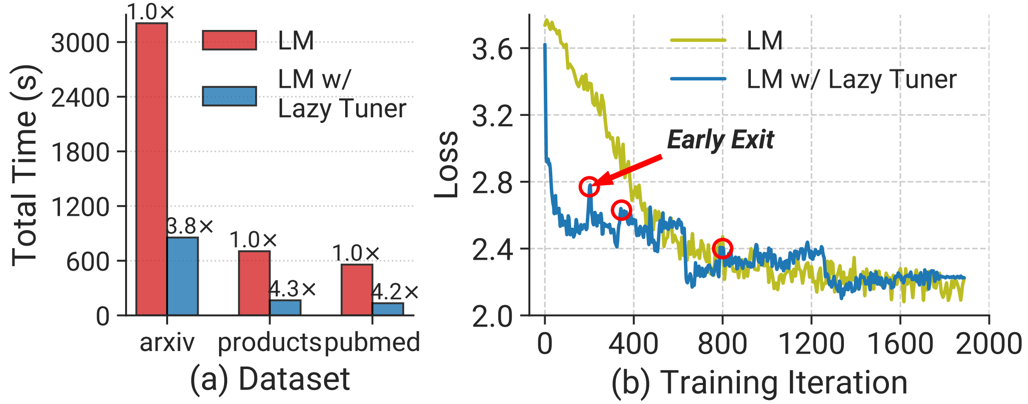


FIGURE 5.11: Effect of Lazy Tuner on DeBERTa-base. (a) Total time effect on LM fine-tuning. (b) Loss curve comparisons with some highlighted exit points.

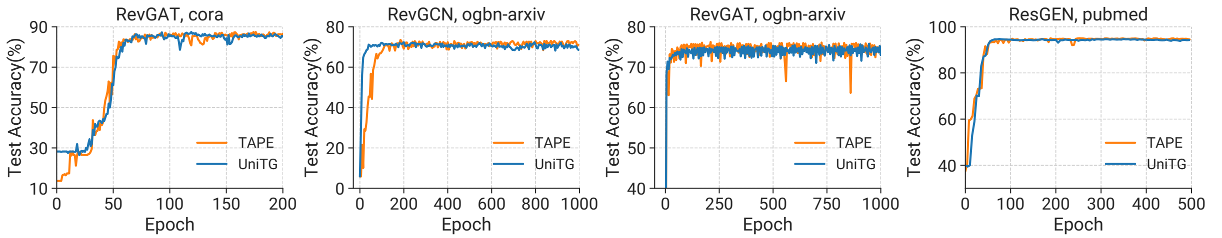


FIGURE 5.12: The convergence curve comparisons of UniTG and TAPE on different models and datasets on one type-B server.

higher than the standard loss (olive curve). Notably, we can observe some spikes in the loss curves, highlighted as red circles. Those spikes stand for the occurrence of exiting at a different layer, e.g., 16th layer other than the original final layer on the first spike. Though there exist many spikes for the early-exit loss during the whole process, it shows an overall convergence trend and the system is tolerant to less precise embeddings for less important nodes.

Model accuracy of Modality Collaborator. We regularly fetch the encoded embeddings from the LM to enhance GNN training. Therefore, we examine the model accuracy and convergence curves of UniTG in Figure 5.12. Here we only consider the TAPE baseline for fair comparisons, since it also employs GNN as the final predictor for node classification. We can see the curves of UniTG are almost identical to that of TAPE and all obtain high final accuracy, verifying that the collaborative learning well preserves the model quality. Particularly, training RevGCN on ogbn-arxiv with UniTG even achieves a much faster convergence speed, which is also in line with Figure 5.11(b) where LM tuning with Lazy Tuner delivers a quicker convergence.

5.4.4 Effect of Streamlined Pipeline Schedule

GPU utilization of Bubble Interleaver. We explore the effects of this module on one type-B GPU server. For better illustration of the impact of Bubble Squeezer, we interleave the LM tuning in data parallelism into the GNN training in FP containing 4 pipeline stages, which is implemented based on PyTorch [117]. To better measure the precise GPU utilization, we employ NVIDIA DCGM [185] to record `SM Activity` as the utilization metric. We capture the utilization traces on one GPU for 40 seconds with and without Bubble Interleaver on two models in Figure 5.13. Two traces are collected separately and we align them manually according to the time axis.

Taking Figure 5.13(a) as an example, for the GNN training only with FP (blue curve), since the only active kernel in the bubble is the NCCL kernel for communication, we observe the extremely poor SM activity of about 7% during the bubble. On the contrary, UniTG inserts each DP process of the LM phase into the bubbles. Hence, it utilizes the spare SMs, accomplishing a relatively higher SM utilization at about 40%. In Figure 5.13(b), the SM activity is increased to around 45% in bubbles as well. Moreover, in both figures, the bubble-interleaved curve matches well at the computation intervals (where SM activity $> 80\%$) and bubble intervals, with no evident delay compared to the GNN FP curve. This validates that Bubble Interleaver can greatly improve hardware utilization with negligible interference on the original GNN task. We also test the throughput influence of directly collocating two phases and find it causes unacceptable interference (about 40% slowdown for both models).

To prove that each LM DP process is correctly interleaved and executed, we also investigate the GPU utilization on other GPUs, namely other pipeline stages of GNN FP. As shown in Figure 5.14, the taller spike at almost 100% SM activity stands for the forward pass and the backward pass reaches nearly 80% SM activity. For stage 1 in Figure 5.14(a), a long bubble occurs right after the forward pass and a short bubble after the backward pass. We can observe the LM DP process on this GPU executes exactly in these two bubble intervals, raising the SM activity to about 40%, and meanwhile not affecting the GNN FP process. The same phenomenon can be seen in the overlapped curves on stage 2. Both validate UniTG can utilize the idle resources timely and accurately. In summary, UniTG successfully achieves much higher GPU utilization during bubble time, and substantially improves training efficiency in a time-sharing way owing to the superior capability of exploiting intermittent GPU resources.

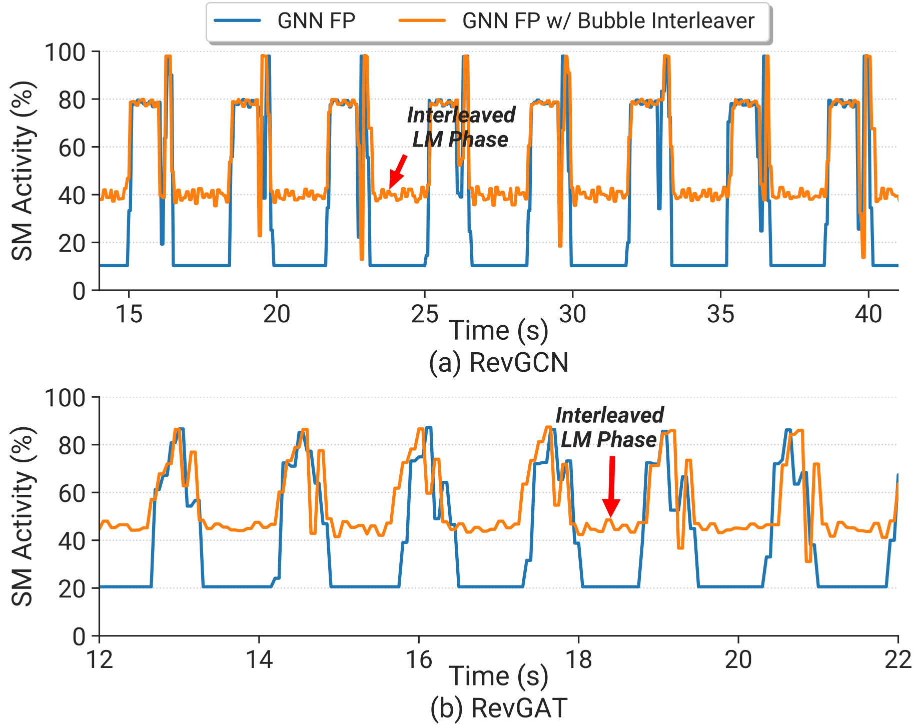


FIGURE 5.13: Visualization of GPU utilization via DCGM on (a) RevGCN and (b) RevGAT. Several iterations of the first pipeline stage are presented. The execution periods of the interleaved LM phase are highlighted by red arrows.

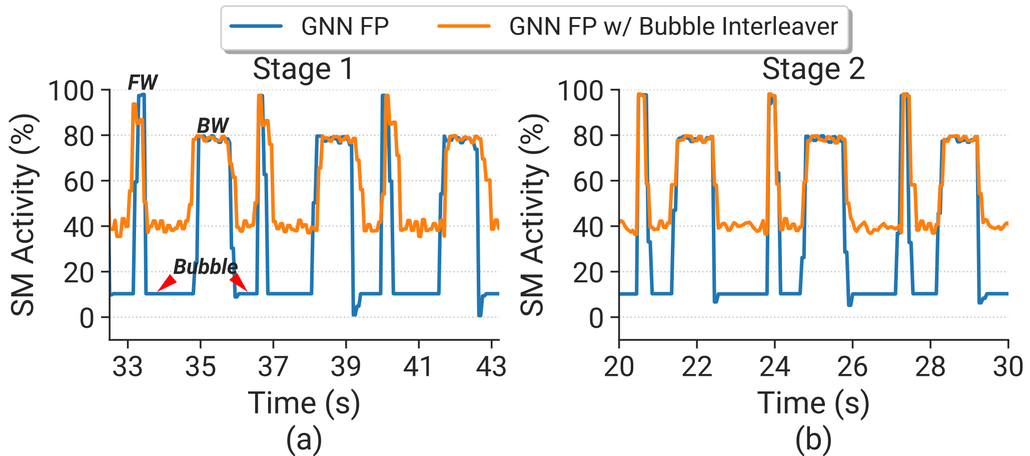


FIGURE 5.14: GPU utilization on the second and third stage when training RevGCN in FP.

5.5 Conclusion

This work proposes UniTG, a unified and efficient system for textual graph learning in one step. The key idea is to realize efficient textual graph learning with algorithm-system co-design techniques. We propose three modules in UniTG to optimize at runtime, algorithm and execution levels respectively. Evaluations show UniTG can boost training by up to 17.3× while achieving comparable final accuracy.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude our research work by summarizing our contributions and analyzing some latent limitations. Then, we discuss the future research directions.

6.1 Conclusion

This thesis makes significant contributions to the field of large-scale graph learning by systematically addressing the critical challenges of computational efficiency and scalability through an innovative algorithm-system co-design paradigm. Our research follows a carefully structured progression across three major graph learning paradigms, each building upon the insights of the previous work while introducing novel solutions to unsolved problems of new algorithms. The sequential development of Sylvie, TorchGT, and UniTG represents a comprehensive approach to advancing the state-of-the-art in graph learning systems.

Our journey begins with Sylvie, a groundbreaking system designed to overcome the fundamental limitations in GNN training for massive graphs. Traditional GNN systems face an inherent trade-off between computational efficiency and model accuracy, particularly when dealing with deeper architectures and larger datasets. Sylvie balances this trade-off well by introducing a sophisticated three-dimensional optimization framework that simultaneously targets data organization and temporal execution patterns. The system's ability to analyze and leverage the inherent structural properties of both the input graph and the model architecture enables unprecedented performance gains. Through extensive experimentation, we demonstrate that Sylvie achieves remarkable speedups of up to 17.2 $\times$ across both shallow and deep GNN architectures, while maintaining and in some cases even improving model accuracy. This work not only establishes a new benchmark for GNN training systems but also provides crucial insights that inform our subsequent research directions.

Building on the foundations laid by Sylvie, we next confront the emerging challenges in graph transformer architectures through TorchGT. While transformers have revolutionized various domains of machine learning, their application to graph-structured data has been severely limited by system constraints, particularly when dealing with long node sequences in large graphs. TorchGT represents the first comprehensive solution to these challenges, combining algorithmic innovations with system-level optimizations. At its core, the Dual-interleaved Attention mechanism fundamentally rethinks how attention computation should be performed on sparse graph structures, achieving both computational efficiency and mathematical equivalence to standard attention. The system architecture of TorchGT introduces a novel parallelization strategy that minimizes communication overhead while maximizing hardware utilization. Furthermore, our dynamic kernel optimization techniques significantly reduce memory access latency, enabling the processing of graphs with sequence lengths of up to 1 million nodes. The experimental results are striking, with TorchGT delivering training speedups of up to 62.7 $\times$ compared to existing approaches, while maintaining full model accuracy. This work not only makes large-scale graph transformers practically feasible but also establishes new directions for research in attention-based graph learning.

The final part of our research addresses the burgeoning field of language model-based graph learning through UniTG. While recent years have seen growing interest in combining the power of large language models with graph-structured data, existing approaches suffer from fundamental limitations in efficiency, flexibility, and graph-awareness. UniTG represents a paradigm shift in this domain, offering the first truly unified framework that seamlessly integrates textual and graph modalities. Our system overcomes the critical defects of previous decoupled architectures by introducing collaborative training tasks and optimizing the entire pipeline for maximal resource utilization. The key innovation lies in UniTG's ability to maintain the rich semantic understanding of language models while effectively capturing structural graph properties, all within a single, end-to-end trainable framework. Experimental results demonstrate that UniTG reduces training makespan by up to 17.3 $\times$ compared to existing LM-based graph learning approaches, without any compromise in model quality. This work opens new possibilities for applications that require joint understanding of textual and relational data, from knowledge graphs to social network analysis.

The collective impact of Sylvie, TorchGT, and UniTG extends far beyond their individual technical contributions. Together, they form a cohesive ecosystem that addresses the full spectrum of modern graph learning challenges, from traditional GNNs to cutting-edge transformer architectures and language model integration. Our work demonstrates that careful algorithm-system co-design can overcome what were

previously considered fundamental limitations in the field, enabling graph learning at scales and efficiencies that were previously unimaginable.

6.2 Limitations

While this thesis has made significant strides in advancing large-scale graph learning systems through the development of Sylvie, TorchGT, and UniTG, several important limitations merit careful consideration as they point to valuable directions for future research. These limitations span multiple dimensions of our work and highlight opportunities for further innovation in the field.

Hyperparameter Selection. One fundamental limitation lies in the hyperparameter selection process for our proposed systems. While we introduce several novel parameters to optimize performance - such as attention sparsity thresholds in TorchGT and the number of centrality groups in UniTG- their current empirical determination may not generalize optimally across diverse scenarios. The static nature of these parameters fails to account for the dynamic characteristics of graph learning processes, where optimal configurations might evolve during training or vary across different graph structures. This limitation suggests the need for more sophisticated adaptive tuning mechanisms that could automatically adjust parameters based on specific characteristics.

Applicable Models and Datasets. The scope of supported models presents another important constraint. While our work comprehensively addresses GNNs, graph transformers, and LM-based approaches, the rapidly evolving field of graph learning continues to introduce new architectures with distinct computational patterns. Temporal GNNs for dynamic graphs, graph transformers with adapted attention and LM4Graph models with new LM integration all exhibit unique characteristics that may require specialized optimization strategies beyond our current framework. This limitation urges for the need for more flexible system designs that can accommodate an expanding landscape of graph learning models. Our experimental validation, while extensive, remains bounded by the characteristics of existing benchmark datasets. Real-world applications increasingly demand capabilities to handle trillion-edge social networks, continuously evolving temporal graphs, and richly structured heterogeneous graphs. The current systems may face unforeseen challenges when applied to these new scenarios, particularly in terms of scalability and robustness. This limitation underscores the importance of developing next-generation benchmarking frameworks that better reflect the complexity of practical applications.

Communication Optimization. At the system level, communication efficiency emerges as a persistent challenge, particularly for distributed training scenarios. While

we optimize intra-device computation, the communication patterns for global attention operations and cross-device synchronization remain potential bottlenecks. The static nature of current communication strategies fails to fully exploit the dynamic characteristics of graph structures and training processes. More suitable approaches that adapt to graph learning conditions could yield significant additional improvements.

Computation Kernel Tailored for Graphs. The foundational sparse computation operations, while effective, may not fully capitalize on the structural properties of real-world graphs. Current implementations use generic sparse formats that do not specifically optimize for common graph characteristics like community structure or hierarchical organization. This represents a missed opportunity for further performance gains through more specialized sparse computation kernels that could better align with graph topology and hardware capabilities.

These limitations collectively highlight the rich potential for future work in graph learning systems. Addressing them will require deeper theoretical understanding of graph computation patterns, more sophisticated hardware-software co-design, and the development of more comprehensive evaluation frameworks. By tackling these challenges, future research can build upon the foundations established in this thesis to enable even more powerful and efficient graph learning systems, ultimately expanding the boundaries of what is possible in graph-structured AI applications.

6.3 Future Work

Building upon the contributions of Sylvie, TorchGT, and UniTG, we envision the next generation of graph learning systems evolving toward greater unification, adaptivity, and hardware ubiquity. We outline this roadmap through three consolidated research themes.

Toward Unified Abstractions for Graph Foundation Models. The rapid evolution from GNNs to Graph Transformers and LM-based methods calls for a move beyond specialized systems for specific models. *Unified Programming Interface*: Future research should aim to develop a single, modular framework capable of natively supporting the diverse spectrum of graph learning—from the sparse aggregation of GNNs to the dense attention of Transformers and the multimodal processing of LMs. This involves creating plug-and-play operators and a unified intermediate representation (IR) that allows compilers to automatically optimize novel computation patterns. *End-to-End Evaluation Infrastructure*: To support this unification, the field requires next-generation benchmarking infrastructure. This includes curating comprehensive

datasets spanning trillion-edge social networks to molecular structures, alongside synthetic generators for systematic stress-testing. Such infrastructure should be complemented by standardized evaluation protocols specifically designed for emerging challenges in heterogeneous or temporal graph learning.

Hardware-Aware Generalization and Scale. While this thesis focused on GPU-centric optimizations, future systems must generalize across diverse hardware backends and unprecedented scales. *Cross-Platform Co-design*: We propose investigating hardware-native sparse formats and kernels co-designed not just for GPUs, but for emerging accelerators. This includes developing predictive sparse computation techniques that anticipate valuable operations, skipping unnecessary calculations at the hardware level without sacrificing quality. *Extreme-Scale Distributed Training*: A critical direction involves validating scalability on clusters of up to 100+ GPUs. Such experiments are essential to reveal optimization opportunities that only emerge at extreme scales, systematically investigating the interplay between model complexity and parallelization efficiency.

Intelligent and Adaptive Runtime Systems. To handle the complexity of real-world deployments, systems must transition from static optimizations to dynamic, self-tuning environments. *Online Meta-Learning for System Configuration*: We envision intelligent systems capable of automatically tuning hyperparameters (e.g., communication quantization levels in SYLVIE or cluster transition threshold in TorchGT) during operation. These systems would continuously monitor performance metrics and adjust computation patterns in real-time based on evolving graph characteristics. *Communication-Centric Design*: Future work should fundamentally rethink distributed training through topology-aware communication protocols. By designing collective operations that match the graph's community structure and utilizing dynamic gradient compression, systems can minimize bandwidth bottlenecks in increasingly distributed environments.

Practical Deployment and Edge Intelligence. Finally, bridging the gap between research and production requires robust deployment frameworks. Future efforts must focus on auto-scaling solutions for cloud environments and incremental learning systems capable of handling streaming graph data. Furthermore, extending these efficient principles to edge-compatible graph learning will be crucial for enabling widespread adoption in latency-sensitive applications.

The future of graph learning systems lies in developing more adaptive, scalable, and comprehensive frameworks that can keep pace with the field's rapid evolution. Key directions include creating self-optimizing systems through meta-learning techniques, building flexible architectures to support diverse graph models, and establishing robust benchmarking ecosystems. These collective efforts will be crucial for unlocking graph learning's full potential across scientific and industrial applications.

Looking forward, this research establishes several important directions for future work. The principles developed in these systems - particularly in terms of sparsity-aware computation, memory hierarchy optimization, and modality integration - can be extended to other emerging areas of graph learning. Furthermore, the success of our approach underscores the importance of interdisciplinary research that bridges machine learning algorithms, systems engineering, and hardware awareness. As graph-structured data continues to grow in importance across scientific and industrial applications, the techniques and frameworks developed in this thesis will serve as foundational elements for the next generation of graph intelligence systems.

Bibliography

[1] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020. 1, 7, 8, 9, 40, 51, 66, 67, 76, 80, 83

[2] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, 2016. 1, 7, 8, 40, 51, 67, 80

[3] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017. 2, 8, 13, 21, 22, 23, 40, 41, 50, 52, 60, 76, 80

[4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR '16, 2016. 8, 9, 10, 12, 21, 22, 41, 50, 51, 77, 80

[5] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2021. 8, 9, 11, 14, 15, 18, 19, 51, 52, 54, 56, 67

[6] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021-12-06. 2, 8, 9, 11, 12, 14, 15, 18, 19, 51, 52, 54, 58, 67

[7] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023. Association for Computing Machinery, 202023. 2, 22, 23, 24

[8] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20. Association for Computing Machinery, 2020. 23

[9] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International*

Conference on Knowledge Discovery & Data Mining, KDD '19, 2019. 2, 13, 21, 22, 80

[10] Lowik Chanussot, Abhishek Das, Siddharth Goyal, Thibaut Lavril, Muhammed Shuaibi, Morgane Riviere, Kevin Tran, Javier Heras-Domingo, Caleb Ho, Weihua Hu, Aini Palizhati, Anuroop Sriram, Brandon Wood, Junwoong Yoon, Devi Parikh, C. Lawrence Zitnick, and Zachary Ulissi. The open catalyst 2020 (oc20) dataset and community challenges. *CoRR*, abs/2010.09990, 2021. 3, 18, 51

[11] Zhiwen Fan, Tianlong Chen, Peihao Wang, and Zhangyang Wang. Cadtransformer: Panoptic symbol spotting transformer for cad drawings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10986–10996, 2022. 18

[12] Erxue Min, Yu Rong, Tingyang Xu, Yatao Bian, Peilin Zhao, Junzhou Huang, Da Luo, Kangyi Lin, and Sophia Ananiadou. Neighbour interaction based click-through rate prediction via graph-masked transformer. *CoRR*, abs/2201.13311, 2022.

[13] Yi-Lun Liao and Tess Smidt. Equiformer: Equivariant graph attention transformer for 3d atomistic graphs. In *International Conference on Learning Representations*, ICLR '23, 2023. 3, 18, 51

[14] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.134. URL https://aclanthology.org/2023.acl-long.134. 3, 18, 52, 56, 61, 87

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *CoRR*, abs/2309.14509, 2023. 62, 78

[16] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems*, MLSys '23, 2023. 61, 87

[17] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023. 3, 18, 52, 56

[18] Meng Zhang, Qinghao Hu, Peng Sun, Yonggang Wen, and Tianwei Zhang. Sylvie: 3d-adaptive and universal system for large-scale graph neural network training. In *2024 IEEE 38th International Conference on Data Engineering (ICDE)*, 2024. 4, 17, 21, 50, 65, 76, 78, 80, 82

[19] Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, and Tianwei Zhang. Torchgt: A holistic system for large-scale graph transformer training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024. 5, 19, 50

[20] Fragkiskos D. Malliaros and Michalis Vazirgiannis. Clustering and community detection in directed networks: A survey. *CoRR*, abs/1308.0971, 2013. 7, 21

[21] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022-12-06. 8, 11, 14, 18, 51, 52, 54, 55, 56

[22] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. Exphormer: Sparse transformers for graphs. In *Proceedings of the 40th International Conference on Machine Learning*, ICML '23, pages 31613–31632, 2023. 8, 11, 14, 18, 19, 51, 52, 54, 55, 56, 60

[23] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '08, 2008. 9

[24] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, ICLR '18, 2018. 9, 10, 21, 22, 23, 41, 50, 51, 60, 76

[25] Qitian Wu, Wentao Zhao, Zenan Li, David Wipf, and Junchi Yan. Nodeformer: A scalable graph structure learning transformer for node classification. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022. 9, 11, 15, 18, 51, 52, 54, 55, 58

[26] Scott Freitas, Yuxiao Dong, Joshua Neil, and Duen Horng Chau. A large-scale database for graph representation learning. *arXiv preprint arXiv:2011.07682*, 2020. 9, 15, 66, 67

[27] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Trans. Graph.*, 38:1–12, 2019. 9

[28] Dongkwan Kim and Alice Oh. How to find your friendly neighborhood: Graph attention design with self-supervision. *CoRR*, abs/2204.04879, 2022. URL `https://arxiv.org/abs/2204.04879`. 9

[29] Yijun Tian, Huan Song, Zichen Wang, Haozhu Wang, Ziqing Hu, Fang Wang, Nitesh V. Chawla, and Panpan Xu. Graph neural prompting with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '24, 2024. 9, 19, 77, 82

[30] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17. JMLR.org, 2017. 10, 13

[31] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *CoRR*, abs/1806.03536, 2018. 10, 41, 80

[32] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning*, ICML '19, pages 6861–6871, 2019. 10, 41

[33] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. Neutronstar: Distributed gnn training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22. Association for Computing Machinery, 2022. 10, 22

[34] Meng Liu, Hongyang Gao, and Shuiwang Ji. Towards deeper graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20. Association for Computing Machinery, 2020. 10, 22, 41

[35] Zizhao Zhang, Xin Wang, Chaoyu Guan, Ziwei Zhang, Haoyang Li, and Wenwu Zhu. Autogt: Automated graph transformer architecture search. In *International Conference on Learning Representations*, ICLR '23, 2023. 11

[36] Jinsong Chen, Kaiyuan Gao, Gaichao Li, and Kun He. Nagphormer: A tokenized graph transformer for node classification in large graphs. In *International Conference on Learning Representations*, ICLR '23, 2023. 11, 18, 19, 51, 52, 55

[37] Md Shamim Hussain, Mohammed J. Zaki, and Dharmashankar Subramanian. Global self-attention as a replacement for graph convolution. *CoRR*, abs/2108.03348, 2022.

[38] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-bert: Only attention is needed for learning graph representations. *CoRR*, abs/2001.05140, 2020.

[39] Devin Kreuzer, Dominique Beaini, William L. Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021. 51, 60

[40] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 1548–1554. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/214. URL https://doi.org/10.24963/ijcai.2021/214. Main Track. 11

[41] Jianan Zhao, Chaozhuo Li, Qianlong Wen, Yiqi Wang, Yuming Liu, Hao Sun, Xing Xie, and Yanfang Ye. Gophormer: Ego-graph transformer for node classification. *CoRR*, abs/2110.13094, 2021. 11, 18, 52, 54, 55

[42] Ling Min Serena Khoo, Hai Leong Chieu, Zhong Qian, and Jing Jiang. Interpretable rumor detection in microblogs by attending to user interactions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '20, 2020. 11

[43] Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. Structure-aware transformer for graph representation learning. In *Proceedings of the 39th International Conference on Machine Learning*, ICML '22, pages 3469–3489, 2022. 11, 51, 54

[44] Ziwei Zhang, Haoyang Li, Zeyang Zhang, Yijian Qin, Xin Wang, and Wenwu Zhu. Graph meets llms: Towards large graph models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems GLFrontiers Workshop*, NIPS'23 GLFrontiers Workshop, 2023. 12

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423/. 12, 76

[46] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. In *International Conference on Learning Representations*, ICLR '21, 2021. 12, 83, 89, 93, 94, 97

[47] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, ICLR '20, 2020. 13, 21, 22, 23, 40, 52

[48] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NIPS'19, 2019.

[49] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, ICLR '18, 2018. 52

[50] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, 2018. 13, 21

[51] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of Machine Learning and Systems*, MLSys '20, 2020. 13, 16, 21, 23, 26, 41, 80

[52] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In *Proceedings of Machine Learning and Systems*, MLSys '22, 2022. 13, 17, 21, 22, 23, 24, 26, 39, 40, 41

[53] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, 2021.

[54] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *CoRR*, abs/2010.05337, 2021. 16, 22, 23, 26

[55] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12:2094–2105, 2019. 16, 26

[56] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *International Conference on Learning Representations*, ICLR '22, 2022. 17, 22, 27, 29, 38, 40, 41

[57] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15:1937–1950, 2022. 13, 17, 22, 23, 24, 27, 29

[58] Karypis George and Kumar Vipin. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997. 14, 40, 61, 87

[59] Xiaoxin He, Xavier Bresson, Thomas Laurent, Adam Perold, Yann LeCun, and Bryan Hooi. Harnessing explanations: Llm-to-lm interpreter for enhanced text-attributed graph representation learning. In *International Conference on Learning Representations*, ICLR '24, 2024. 16, 19, 20, 76, 77, 78, 80, 81, 82, 83, 93, 94

[60] Eli Chien, Wei-Cheng Chang, Cho-Jui Hsieh, Hsiang-Fu Yu, Jiong Zhang, Olgica Milenkovic, and Inderjit S Dhillon. Node feature extraction by self-supervised multi-scale neighborhood prediction. In *International Conference on Learning Representations*, ICLR '22, 2022. 16, 19, 76, 77, 78, 80, 81, 82

[61] Shichang Zhang, Atefeh Sohrabizadeh, Cheng Wan, Zijie Huang, Ziniu Hu, Yewen Wang, Jason Cong, Yizhou Sun, et al. A survey on graph neural network acceleration: Algorithms, systems, and customized hardware. *arXiv preprint arXiv:2306.14052*, 2023. 16

[62] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, 2019. 16, 23, 24, 26

[63] Morteza Ramezani, Weilin Cong, Mehrdad Mahdavi, Mahmut Kandemir, and Anand Sivasubramaniam. Learn locally, correct globally: A distributed algorithm for training graph neural networks. In *International Conference on Learning Representations*, ICLR '22, 2022. 16, 26

[64] Hesham Mostafa. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In *Proceedings of Machine Learning and Systems*, MLSys '22, 2022. 17, 23, 26, 41

[65] Borui Wan, Juntao Zhao, and Wu Chuan. Adaptive message quantization and parallelization for distributed full-graph gnn training. In *Proceedings of Machine Learning and Systems*, MLSys '23, 2023.

[66] Alok Tripathy, Katherine Yelick, and Aydın Buluç. Reducing communication in graph neural network training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020. 17, 22, 23, 24, 26, 27, 41

[67] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. Sgquant: Squeezing the last bit on graph neural networks with specialized quantization. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020. 17, 27, 30, 32

[68] Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. Learned low precision graph neural networks. *CoRR*, abs/2009.09232, 2020. 17, 27

[69] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, ICLR '21, 2021. 17, 21, 27, 30, 32, 40

[70] Yuke Wang, Boyuan Feng, and Yufei Ding. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, 2022. 17, 27

[71] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. Ec-graph: A distributed graph neural network system with error-compensated compression. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 648–660, 2022. doi: 10.1109/ICDE53745.2022.00053. 17, 27

[72] Shyam A. Tailor, Javier Fernandez-Marques, and Nicholas D. Lane. Degree-quant: Quantization-aware training for graph neural networks. *CoRR*, abs/2008.05000, 2021. 17, 28, 30, 32, 34, 40, 78, 82

[73] Yuxin Ma, Ping Gong, Jun Yi, Zhewei Yao, Cheng Li, Yuxiong He, and Feng Yan. Bifeat: Supercharge gnn training via graph feature quantization. *CoRR*, abs/2207.14696, 2023. 17, 28

[74] Chuang Liu, Yibing Zhan, Xueqi Ma, Liang Ding, Dapeng Tao, Jia Wu, and Wenbin Hu. Gapformer: Graph transformer with graph pooling for node classification. In Edith Elkind, editor, *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pages 2196–2205. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/244. URL https://doi.org/10.24963/ijcai.2023/244. Main Track. 18, 55

[75] Qitian Wu, Wentao Zhao, Chenxiao Yang, Hengrui Zhang, Fan Nie, Haitian Jiang, Yatao Bian, and Junchi Yan. Simplifying and empowering transformers for large-graph representations. In *Advances in Neural Information Processing Systems*, NeurIPS '23, 2023. 18, 51, 52, 55

[76] Qitian Wu, Chenxiao Yang, Wentao Zhao, Yixuan He, David Wipf, and Junchi Yan. Difformer: Scalable (graph) transformers induced by energy constrained diffusion. In *International Conference on Learning Representations*, ICLR '23, 2023. 18, 51, 52, 55

[77] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, ICLR '23, 2023. 18, 52, 55, 58

[78] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020. 18, 52, 58, 63

[79] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768, 2020.

[80] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, ICLR '20, 2020. 18, 55

[81] H. Wang, Z. Zhang, and S. Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society. doi: 10.1109/HPCA51647.2021.00018. URL https://doi.ieeecomputersociety.org/10.1109/HPCA51647.2021.00018. 18, 55

[82] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. Learning on large-scale text-attributed graphs via variational inference. In *International Conference on Learning Representations*, ICLR '22, 2022. 19, 76, 77, 78, 80, 81, 82, 83, 90, 94

[83] Jason Zhu, Yanling Cui, Yuming Liu, Hao Sun, Xue Li, Markus Pelger, Tianqi Yang, Liangjie Zhang, Ruofei Zhang, and Huasha Zhao. Textgnn: Improving text encoder via graph neural network in sponsored search. In *Proceedings of the Web Conference 2021*, WWW '21. Association for Computing Machinery, 2021. 19, 76, 77, 78

[84] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. Gpt-gnn: Generative pre-training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20. Association for Computing Machinery, 2020. 77, 81

[85] Keyu Duan, Qian Liu, Tat-Seng Chua, Shuicheng Yan, Wei Tsang Ooi, Qizhe Xie, and Junxian He. Simteg: A frustratingly simple approach improves textual graph learning. *CoRR*, abs/2308.02565, 2023. URL https://arxiv.org/abs/2308.02565. 77, 78, 80, 82, 90

[86] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. Qa-gnn: Reasoning with language models and knowledge graphs for question answering. *CoRR*, abs/2104.06378, 2021. URL https://arxiv.org/abs/2104.06378.

[87] Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. Greaselm: Graph reasoning enhanced language models. In *International Conference on Learning Representations*, ICLR '22, 2022.

[88] Yijian Qin, Xin Wang, Ziwei Zhang, and Wenwu Zhu. Disentangled representation learning with large language models for text-attributed graphs. *CoRR*, abs/2310.18152, 2023. URL https://arxiv.org/abs/2310.18152. 19, 77

[89] Jiong Zhang, Wei-cheng Chang, Hsiang-fu Yu, and Inderjit S. Dhillon. Fast multi-resolution transformer fine-tuning for extreme multi-label text classification. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21. Curran Associates Inc., 2021. 19

[90] Costas Mavromatis, Vassilis N. Ioannidis, Shen Wang, Da Zheng, Soji Adeshina, Jun Ma, Han Zhao, Christos Faloutsos, and George Karypis. Train your own gnn teacher: Graph-aware distillation on textual graphs. In *Machine Learning and Knowledge Discovery in Databases: Research Track: European Conference, ECML PKDD 2023, Turin, Italy, September 18–22, 2023, Proceedings, Part III*, Machine Learning and Knowledge Discovery in Databases: Research Track: European Conference, ECML PKDD 2023, Turin, Italy, September 18–22, 2023, Proceedings, Part III. Springer-Verlag, 202023,18–22,2023. 19

[91] Michihiro Yasunaga, Antoine Bosselut, Hongyu Ren, Xikun Zhang, Christopher D Manning, Percy S. Liang, and Jure Leskovec. Deep bidirectional

language-knowledge graph pretraining. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022-12-06. 19, 77

[92] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23. Curran Associates Inc., 2023. 20

[93] Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. Gpt4graph: Can large language models understand graph structured data ? an empirical evaluation and benchmarking. *CoRR*, abs/2305.15066, 2023. URL `https://arxiv.org/abs/2305.15066`.

[94] Jiawei Zhang. Graph-toolformer: To empower llms with graph reasoning ability via prompt augmented by chatgpt. *CoRR*, abs/2304.11116, 2023. URL `https://arxiv.org/abs/2304.11116`. 20

[95] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2019. 21, 23, 50

[96] Pengcheng Li, Yixin Guo, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Xu Liu. Graph neural networks based memory inefficiency detection using selective sampling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. 76

[97] Qingxiao Sun, Yi Liu, Hailong Yang, Ruizhe Zhang, Ming Dun, Mingzhen Li, Xiaoyan Liu, Wencong Xiao, Yong Li, Zhongzhi Luan, and Depei Qian. Cognn: efficient scheduling for concurrent gnn training on gpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. 50, 76

[98] Haoyang Li and Lei Chen. Early: Efficient and reliable graph neural network for dynamic graphs. *Proc. ACM Manag. Data*, 1:1–28, 2023. 21

[99] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *CoRR*, abs/2010.00130, 2021. 21, 50

[100] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, NeurIPS '18, 2018. 21

[101] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15:1228–1242, 2022. 22

[102] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. Deepergcn: All you need to train deeper gcns. *CoRR*, abs/2006.07739, 2020. 22, 23, 80, 94

[103] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, NeurIPS '19, 2019. 23

[104] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017. 23, 27

[105] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *CoRR*, abs/1909.01315, 2020. 23, 30, 40, 41

[106] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019. 23

[107] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 515–531. USENIX Association, 2021. 23, 24, 61, 82, 86

[108] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E. Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *Proceedings of Machine Learning and Systems*, MLSys '22, 2022. 23

[109] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *CoRR*, abs/2112.08541, 2021. 23

[110] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018. 27, 29

[111] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '18, 2018. 27, 29

[112] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, 2021. 27, 30, 32, 40

[113] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017. 27

[114] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training. 2018. 38

[115] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, NeurIPS '14, 2014. 27, 38

[116] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '20, June 2020. 30, 32

[117] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, NeurIPS '19, 2019. 30, 40, 66, 93, 99

[118] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, NeurIPS '15, 2015. 33

[119] Alessandra Sala, Haitao Zheng, Ben Y Zhao, Sabrina Gaito, and Gian Paolo Rossi. Brief announcement: revisiting the power-law degree distribution for social graph analysis. In *Proceedings of the 29th ACM SIGACT SIGOPS symposium on Principles of distributed computing*, PODC '10, 2010. 33, 52

[120] Divyansh Jhunjhunwala, Advait Gadhikar, Gauri Joshi, and Yonina C. Eldar. Adaptive quantization of model updates for communication-efficient federated learning. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3110–3114, 2021. doi: 10.1109/ICASSP39728.2021.9413697. 36

[121] Robert Hönig, Yiren Zhao, and Robert Mullins. DAdaQuant: Doubly-adaptive quantization for communication-efficient federated learning. In *Proceedings of the 39th International Conference on Machine Learning*, ICML '22, pages 8852–8866, 2022. 36

[122] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning*, ICML '18, pages 3043–3052, 2018. 38

[123] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme model compression. In *International Conference on Learning Representations*, ICLR '20, 2020. 40

[124] Yinpeng Dong, Renkun Ni, Jianguo Li, Yurong Chen, Jun Zhu, and Hang Su. Learning accurate low-bit deep neural networks with stochastic quantization. *CoRR*, abs/1708.01001, 2017. 40

[125] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *Proceedings of the 37th International Conference on Machine Learning*, ICML '20, pages 1725–1735, 2020. 41, 80

[126] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2017. 41

[127] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '20, 2020. 50

[128] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, ICLR '21, 2021. 50

[129] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. In *International Conference on Learning Representations*, ICLR '22, 2022. 50

[130] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '19, 2019. 50

[131] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017. 50

[132] Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E. Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021-12-06. 51

[133] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, NeurIPS '22, 2022-12-06. 51, 54, 60, 67, 69, 70

[134] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. Association for Computing Machinery, 2013. 52

[135] Hang Liu and H. Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15. Association for Computing Machinery, 2015. 52

[136] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020. 53

[137] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. 53

[138] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *CoRR*, abs/1711.02782, 2017. 53, 63

[139] Jiezhong Qiu, Hao Ma, Omer Levy, Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020. doi: 10.18653/v1/2020.findings-emnlp.232. 53, 63, 64

[140] Chulhee Yun, Yin-Wen Chang, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. O(n) connections are expressive enough: Universal approximability of sparse transformers. *Advances in Neural Information Processing Systems*, 33:13783–13794, 2020. 59, 60

[141] Hamiltonian path. https://mathworld.wolfram.com/HamiltonianPath.html, 2020. 60

[142] Choongbum Lee and Benny Sudakov. Dirac's theorem for random graphs. *CoRR*, abs/1108.2502v3, 2012. URL https://arxiv.org/abs/1108.2502. 60

[143] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010. ISSN 0370-1573. doi: https://doi.org/10.1016/j.physrep.2009.11.002. 61, 85

[144] M. E. J. Newman. Spectral methods for community detection and graph partitioning. *Phys. Rev. E*, 88:042822, Oct 2013. doi: 10.1103/PhysRevE.88.042822. URL https://link.aps.org/doi/10.1103/PhysRevE.88.042822. 61, 85, 86

[145] Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000. ISSN 0167-8191. doi: https://doi.org/10.1016/S0167-8191(00)00048-X. Graph Partitioning and Parallel Computing. 61, 86

[146] S. Gray, A. Radford, and D. P. Kingma. Gpu kernels for block-sparse weights. *CoRR*, abs/1711.09224, 2017. 64

[147] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 24(43):1–48, 2023. 67

[148] Junhan Yang, Zheng Liu, Shitao Xiao, Chaozhuo Li, Defu Lian, Sanjay Agrawal, Amit Singh, Guangzhong Sun, and Xing Xie. Graphformers: Gnn-nested transformers for representation learning on textual graph. In *Advances in Neural Information Processing Systems*, NeurIPS '21, 2021-12-06. 76, 77

[149] Michihiro Yasunaga, Rui Zhang, Kshitijh Meelu, Ayush Pareek, Krishnan Srinivasan, and Dragomir Radev. Graph-based neural multi-document summarization. *CoRR*, abs/1706.06681, 2017. URL https://arxiv.org/abs/1706.06681. 76

[150] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. Adaptive universal generalized pagerank graph neural network. In *International Conference on Learning Representations*, ICLR '21, 2021. 76

[151] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, NeurIPS '13, 2013. 76

[152] Zellig Harris. Distributional structure. In *The philosophy of linguistics*, 1985. 76

[153] Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, and Yongfeng Zhang. Language is all a graph needs. *EACL*, 2024. 78, 81

[154] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, ICLR '22, 2022. 78, 89

[155] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16:3848–3860, 2023. 78

[156] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, NeurIPS '19, 2019. 87, 88

[157] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *CoRR*, abs/1806.03377, 2018. URL https://arxiv.org/abs/1806.03377. 87, 88

[158] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19. Association for Computing Machinery, 2019. 78, 87, 88

[159] Pierre Stock, Benjamin Graham, Rémi Gribonval, and Hervé Jégou. Equi-normalization of neural networks. In *International Conference on Learning Representations*, ICLR '19, 2019. 80

[160] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019. doi: 10.1073/pnas.1903070116. URL `https://www.pnas.org/doi/abs/10.1073/pnas.1903070116`. 80

[161] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, ICLR '20, 2020. 80

[162] Kaixiong Zhou, Xiao Huang, Yuening Li, Daochen Zha, Rui Chen, and Xia Hu. Towards deeper graph neural networks with differentiable group normalization. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020. 80

[163] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, CVPR '19, October 2019. 80, 94

[164] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. Training graph neural networks with 1000 layers. In *Proceedings of the 38th International Conference on Machine Learning*, ICML '21, pages 6437–6449, 2021. 80, 83, 94

[165] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Trans. Graph.*, 38:1–12, 2019. 80

[166] Iro Armeni, Sasha Sax, Amir R. Zamir, and Silvio Savarese. Joint 2d-3d-semantic data for indoor scene understanding. *CoRR*, abs/1702.01105, 2017. URL `https://arxiv.org/abs/1702.01105`. 80

[167] Jingji Chen, Zhuoming Chen, and Xuehai Qian. Gnnpipe: Scaling deep gnn training with pipelined model parallelism. *CoRR*, abs/2308.10087, 2023. URL `https://arxiv.org/abs/2308.10087`. 80

[168] Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. Ee-llm: large-scale training and inference of early-exit large language models with 3d parallelism. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024. 82, 89, 90

[169] Hossein KhademSohi, Mohammadamin Abedi, Yani Ioannou, Steve Drew, Pooyan Jamshidi, and Hadi Hemmati. Selfxit: An unsupervised early exit

mechanism for deep neural networks. In *International Conference on Learning Representations*, ICLR '24, 2024. 82

[170] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, Yan Zeng, Yangang Wang, and Xuebin Chi. Acc-spmm: Accelerating general-purpose sparse matrix-matrix multiplication with gpu tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25. Association for Computing Machinery, 2025. 82, 85

[171] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. doi: 10.1137/1.9781611971538. 86

[172] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. Association for Computing Machinery, 2021. 88

[173] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 88

[174] Fanxin Li, Shixiong Zhao, Yuhao Qing, Xusheng Chen, Xiuxian Guan, Sen Wang, Gong Zhang, and Heming Cui. Fold3d: Rethinking and parallelizing computational and communicational tasks in the training of large dnn models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1432–1449, 2023. doi: 10.1109/TPDS.2023.3247883. 88

[175] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, ICML '19, pages 2790–2799, 2019. 89

[176] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *CoRR*, abs/2103.10385, 2023. URL https://arxiv.org/abs/2103.10385. 89

[177] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353/. 89

[178] Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. FastBERT: a self-distilling BERT with adaptive inference time. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6035–6044, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.537. URL https://aclanthology.org/2020.acl-main.537/. 89, 90

[179] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. In *International Conference on Learning Representations*, ICLR '20, 2020. 90

[180] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, MLSys '23, 2023. 89

[181] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 559–578. USENIX Association, 2022. 92

[182] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. Association for Computing Machinery, 2017.

[183] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22. Association for Computing Machinery, 2022. 92

[184] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/. 93

[185] Nvidia data center gpu manager. https://developer.nvidia.com/dcgm, 2023. 99