Accelerating GNN Inference with GPU Feature Caching

by

Henry Liu

henry.liu@utexas.edu

Advised by Dr. Aditya Akella

A document submitted in partial fulfillment of the requirements for the degree of
Technical Report
at
University of Texas at Austin

Abstract

Graph Neural Networks (GNNs) have gained significant popularity due to their excellent performance in many graph representation learning tasks. Graphs used for GNN computation usually associate high-dimensional feature vectors with each node, which must be copied to the GPU for GNN execution. This work investigates how GNN inference systems can dynamically cache graph features in GPU memory to reduce request-response latencies.

Existing GNN literature has identified that efficiently moving graph features to GPUs, known as data loading, is an important challenge at training time. To reduce data loading overheads, state of the art GNN training systems statically cache graph features on the GPU. We observe that the data loading problem is exacerbated at inference time and becomes a disproportionate bottleneck when serving inference requests. We improve upon existing static caching techniques to be better suited for inference.

In particular, we motivate the use of dynamic cache policies to exploit graph locality of inference requests and explore mechanisms to reduce the impact of dynamic cache overheads on latency and throughput. Our system targets a single-machine, multi-GPU setting and performs cache updates asynchronously to remove cache update operations from the latency-sensitive request-response path. We show how a simple frequency-based cache admission and eviction policy can achieve better cache hit rates than existing static cache baselines. We then leverage interconnects such as NVLink to allow multiple GPUs to share a single logical cache and propose a lock-free synchronization mechanism to reduce lock contention due to cache updates.

Contents

Part I

A good part

You can also use parts in order to partition your great work into larger ‘chunks’. This involves some manual adjustments in terms of the layout, though.

1 Introduction

Graphs are flexible and powerful representations for data across many domains. Recently, Graph Neural Networks (GNNs) have emerged as a class of state-of-the-art (SOTA) machine learning methods for graph representation learning. GNNs adopt ideas from traditional Deep Neural Networks (DNNs) and combine them with techniques that capture structural information about graphs.

Traditional DNNs have excelled in various tasks across domains such as computer vision [14][20] and natural language processing [8][15]. In these domains, inputs exhibit a fairly regular structure.

GNNs are deployed in production systems for many applications, including bioinformatics [28] [19], traffic forecasting [21] [3] [11], recommendation systems [27] [25][10][22][24][6], cybersecurity [12] [18], and optimization [7][2], among many others. Although there has been significant work on GNN training systems [todo cite stuff], GNN inference is relatively understudied.

As opposed to traditional graph processing, graphs used with GNNs associate *features* with each node in the graph. Features are commonly represented as multidimensional tensors, often ranging from the hundreds to thousands. GNNs use these features to compute *embeddings* for each node in the graph by recursively aggregating each node's neighboring features. The resulting embeddings can be used for tasks such as node classification, link prediction, or graph classification. Different GNN architectures such as Graph Convolutional Networks (GCN) [13], GraphSAGE [9], and Graph Attention Networks (GAT) [23] perform different types of aggregations and operations on features, but share the same neighborhood aggregation principle.

1.1 Contributions

In this work, we analyze the outsized impact of feature movement from CPU to GPU on GNN inference. Even when using static caching techniques borrowed from GNN training, this data loading step still remains a major bottleneck in achieving low latency inference. Gathering features in host memory and copying to the GPU comprises roughly 50%-80% of end-to-end inference latency. We tackle this problem by improving GPU feature caches in the following ways.

First, we motivate the use of dynamic cache policies by noting that GNN inference requests create opportunities to exploit graph locality not present at training time. Assuming mini-batch training, GNNs are trained mini-batches that are generally sampled uniformly randomly from the entire graph. However, at inference time, GNNs need to produce answers for new requests that connect into the existing graph. These requests can exhibit locality in graph structure due to factors such as user trends, leading to "hot" subgraphs.

We show how a simple frequency-based admission and eviction policy can provide better cache hit rates than static caches or traditional policies such as LFU. We demonstrate that this holds across inferences traces corresponding to uniformly sampled requests as well as requests concentrated in hot subgraphs.

Second, we use asynchrony as a technique to avoid overheads due to dynamic cache updates. Although dynamic caches can provide better cache hit rates, end-to-end performance can be eroded due to the overhead of actually performing cache updates. By moving cache update operations to different host threads and CUDA streams, we take these operations off of the critical path when responding to inference requests.

Lastly, we propose a lock-free mechanism for performing cache updates. In a system with many inference threads and pipelining, an asynchronous cache update can produce end-to-end performance equivalent to

that of a synchronous one if it blocks pipeline stages due to naive locking. Furthermore, our system supports sharing of a single logical cache among multiple GPUs connected by NVLink, and the blocking of inter-GPU communication can exacerbate locking overheads. To address this we show how the cache can be *masked* to allow for wait-free readers and lock-free writers.

2 Background and Motivations

2.1 GNN Background

Given a graph containing nodes, edges, and features, GNNs output an embedding for each node in the graph. An embedding is a d-dimensional representation of the aggregated feature information from a node's neighborhood. Depending on the learning task, similarity in the embedding space can represent different things, such as similarity of node type (node classification task) or likelihood of edge existence (link prediction task) [todo cite].

In this work we will consider the case where graphs are homogeneous (one node type) and only nodes have associated features.

Borrowing notation from P3 [5], we can generally represent the embedding for a node v at layer k as h_v^k, where

$$h_v^k = \sigma\Big(W^k \cdot \texttt{COMBINE}^{(k)}\Big(h_v^{k-1}, \texttt{AGG}^{(k)}\Big(\{h_u^{k-1} \mid u \in N(v)\}\Big)\Big)\Big) \tag{2.1}$$

$$\begin{aligned} \sigma &= \text{Nonlinear function} \\ W^k &= \text{Trainable weight matrix for layer } k \\ N(v) &= \text{Neighborhood of node v} \end{aligned}$$

$\texttt{AGG}^(k)$ is a function that aggregates the last layer embeddings of a node's neighborhood. $\texttt{COMBINE}^(k)$ is a function that combines that the aggregated neighborhood embeddings with the node's own last layer embedding. The first layer embedding for a node v, h_k^0, is just the original features of that node.

Since for each layer there is a different W^k, GNNs are comprised of k neural networks. Generally GNNs use 1-5 layers, with 2 layers being fairly standard [1]. However, some architectures can use significantly more layers, such as the current SOTA EnGCN model comprising 8 layers [4].

2.1.1 GNN Inference Task

In this work we look at GNN inference as a *online* problem, where given a request comprising a new node (or batch of nodes), its features, and edges connecting into the existing graph, the GNN needs to compute the embedding for the new node. This is as opposed to traditional *offline*, full-graph inference, where inference is performed on all nodes in the existing graph, usually for training evaluation purposes or for saving embeddings for later retrieval. The online inference formulation has many uses depending on domain. For example, in a social network graph, an inference request can correspond to computing the embedding for a new user. Additionally, this formulation naturally works with temporal graphs, where features or graph structures change over time. For example, an inference request could contain the node id and edges of an existing node in the graph, but contain a new feature vector. Thus our online inference formulation can capture time-evolving behaviors of both graph structure and graph features.

Our system currently does not combine new inference requests into the existing graph or retrain the model to accommodate for new requests. Instead, we look specifically at the *computation* aspect, understanding how

to efficiently compute a new embedding. Modifying the existing graph and dealing with challenges such as consistency are outside the scope of this work, but would be an interesting and natural extension.

2.2 GNN Computation Challenges

The process for performing a GNN forward pass is roughly as follows. Assuming graph structure and features fit in a single machine's CPU memory, if we want to compute the embedding for a set of *target* nodes:

1. **Sampling:** Construct k-hop neighborhood for target nodes and build logical computation graph describing GNN computation.
2. **Feature gather:** Gather node features corresponding to k-hop neighborhood in contiguous CPU buffer/
3. **CPU-GPU copy:** Copy buffer with node features and computation graph to GPU.
4. **Model execution:** Perform GNN computation on GPU.

Figure 2.1 shows how these steps would be executed on a baseline system. In a training setting the set of target nodes would be a minibatch, and the model execution step would include backpropagation. In the inference settings, the set of target nodes come from an inference requests, and the model execution step only includes the forward pass.

Figure 2.1: [todo add picture of strawman GNN inference system]

2.2.1 Neighborhood Explosion Problem

The need to recursively build node neighborhoods for GNN execution means that GNNs suffer from a *neighborhood explosion* problem. As the number of GNN layers increase, the amount of nodes needed in the first layer increase exponentially. This leads

To combat this, many GNN training systems and models leverage *neighborhood sampling*, a technique where a fixed number or certain percentage of nodes are sampled at each layer [9] [todo other sampling works]. However, neighborhood sampling results in a drop in accuracy that may be unacceptable for some applications.

Neihgborhood explosion Feature transfer size GPU Sampling

2.3 GNN Inference Bottlenecks

Figure 2.2: [todo add graphs showing latency breakdown for different graphs]

Figure 2.3: [todo add graphs showing latency breakdown for different graphs]

2.4 Existing Caching Solutions

Prior GNN training works have observed that both GPU compute and memory are underutilized while training. As a result, node features can be cached in GPU memory so that they no longer need to be copied over from host memory, easing data loading bottlenecks. PaGraph [16] used this opportunity to introduce *static feature caching*, proposing a policy where the features of the highest degree nodes in the graph are stored on the GPU prior to training. GNNLab [26] extended this idea to include a "pre-sampling" phase, where warmup training epochs are run to determine what nodes are most often used and thus should be stored in the cache. BGL [17] introduces a dynamic FIFO cache and iterates over the graph in a roughly-BFS manner to exploit the FIFO cache.

However, for inference, a system cannot constrain when requests come in or how requests connect to the existing graph. Therefore BGL's ordering based approach cannot be adapted to inference, while GNNLab's pre-sampling technique cannot be directly applied. However, GNNLab's pre-sampling technique provides the insight that a frequency based approach to identifying hot nodes is effective. We adapt this idea to create an online version that is suitable for inference, discussed in section [todo add a ref].

Figure 2.4: [todo add graphs showing latency breakdown for different graphs]

3 Implementation

3.1 System Overview

3.2 Frequency-based admission & eviction policy

1. Counts are stored on GPU in 1 byte

3.3 Asynchronous cache update mechanisms

Operation	Update Time (ms)	Percent of Update Time
Compute most common features	0.6	10%
Feature copy	2.2	38%
Update cache metadata	2.7	47%
Misc. (locking, device sync, etc.)	0.2	3.5%
Total	5.7	

Table 3.1: Breakdown of time spent on operations when performing cache update. These are the operations that contribute to significant tail latency with the prefetching policy.

1. Python process places torch tensor into queue, picked up by C++ thread
2. Removed from queue when model execution finishes to avoid ballooning memory usage
3. Also experimented with updating counts/doing other operations in C++ thread

3.4 Lock-free cache updates

With a fully saturated pipeline, can nearly fully block with locking and thus will lose any gains from asynchrony.

3.5 Multi-GPU cache sharing

1. Mention pinned memory buffer reused between requests, show pinned mem vs pagable mem graph
2.

4 Evaluation

4.1 PyTorch Direct Experiments

Acronyms

GNN	Graph Neural Network
SOTA	State-of-the-art

Glossary

LaTeX	A document preparation system
$\mathbb{R}$	The set of real numbers

Bibliography

1. S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón. "Computing Graph Neural Networks: A Survey from Algorithms to Accelerators". *ACM Comput. Surv.* 54:9, 2021. ISSN: 0360-0300. DOI: 10.1145/3477141. URL: https://doi.org/10.1145/3477141.
2. Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković. "Combinatorial Optimization and Reasoning with Graph Neural Networks". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Z.-H. Zhou. International Joint Conferences on Artificial Intelligence Organization, 2021, pp. 4348–4355.
3. A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, P. W. Battaglia, V. Gupta, A. Li, Z. Xu, A. Sanchez-Gonzalez, Y. Li, and P. Velickovic. "ETA Prediction with Graph Neural Networks in Google Maps". *CoRR* abs/2108.11482, 2021. arXiv: 2108.11482. URL: https://arxiv.org/abs/2108.11482.
4. K. Duan, Z. Liu, P. Wang, W. Zheng, K. Zhou, T. Chen, X. Hu, and Z. Wang. *A Comprehensive Study on Large-Scale Graph Training: Benchmarking and Rethinking*. 2023. arXiv: 2210.07494 [cs.LG].
5. S. Gandhi and A. P. Iyer. "P3: Distributed Deep Graph Learning at Scale". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021, pp. 551–568. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/gandhi.
6. C. Gao, Y. Zheng, N. Li, Y. Li, Y. Qin, J. Piao, Y. Quan, J. Chang, D. Jin, X. He, and Y. Li. "A Survey of Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions". *ACM Transactions on Recommender Systems (TORS)*, 2022.
7. M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. "Exact Combinatorial Optimization with Graph Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 2019, pp. 15554–15566. URL: https://proceedings.neurips.cc/paper/2019/hash/d14c2267d848abeb81fd590f371d39bd-Abstract.html.
8. A. Graves. "Generating Sequences With Recurrent Neural Networks". *CoRR* abs/1308.0850, 2013. arXiv: 1308.0850. URL: http://arxiv.org/abs/1308.0850.
9. W. L. Hamilton, R. Ying, and J. Leskovec. "Inductive Representation Learning on Large Graphs". *CoRR* abs/1706.02216, 2017. arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216.
10. X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang. "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '20. Association for Computing Machinery, Virtual Event, China, 2020, pp. 639–648. ISBN: 9781450380164. DOI: 10.1145/3397271.3401063. URL: https://doi.org/10.1145/3397271.3401063.
11. W. Jiang and J. Luo. "Graph Neural Network for Traffic Forecasting: A Survey". *CoRR* abs/2101.11174, 2021. arXiv: 2101.11174. URL: https://arxiv.org/abs/2101.11174.

12. H. Kim, B. S. Lee, W.-Y. Shin, and S. Lim. "Graph Anomaly Detection With Graph Neural Networks: Current Status and Challenges". *IEEE Access* 10, 2022, pp. 111820–111829. DOI: 10.1109/ACCESS.2022.3211306.

13. T. N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". *CoRR* abs/1609.02907, 2016. arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

14. A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. Burges, L. Bottou, and K. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

15. G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. "Neural Architectures for Named Entity Recognition". *CoRR* abs/1603.01360, 2016. arXiv: 1603.01360. URL: http://arxiv.org/abs/1603.01360.

16. Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. 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, Virtual Event, USA, 2020, pp. 401–415. ISBN: 9781450381376. DOI: 10.1145/3419111.3421281. URL: https://doi.org/10.1145/3419111.3421281.

17. T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. "BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing". *CoRR* abs/2112.08541, 2021. arXiv: 2112.08541. URL: https://arxiv.org/abs/2112.08541.

18. Y. Liu, Z. Sun, and W. Zhang. "Improving Fraud Detection via Hierarchical Attention-Based Graph Neural Network". *J. Inf. Secur. Appl.* 72:C, 2023. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2022.103399. URL: https://doi.org/10.1016/j.jisa.2022.103399.

19. M. Réau, N. Renaud, L. C. Xue, and A. M. J. J. Bonvin. "DeepRank-GNN: a graph neural network framework to learn patterns in protein–protein interfaces". *Bioinformatics* 39:1, 2022.

20. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

21. A. Roy, K. K. Roy, A. A. Ali, M. A. Amin, and A. K. M. M. Rahman. "SST-GNN: Simplified Spatio-temporal Traffic forecasting model using Graph Neural Network". *CoRR* abs/2104.00055, 2021. arXiv: 2104.00055. URL: https://arxiv.org/abs/2104.00055.

22. J. Sun, Y. Zhang, W. Guo, H. Guo, R. Tang, X. He, C. Ma, and M. Coates. "Neighbor Interaction Aware Graph Convolution Networks for Recommendation". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '20. Association for Computing Machinery, Virtual Event, China, 2020, pp. 1289–1298. ISBN: 9781450380164. DOI: 10.1145/3397271.3401123. URL: https://doi.org/10.1145/3397271.3401123.

23. P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].

24. J. Wu, X. Wang, F. Feng, X. He, L. Chen, J. Lian, and X. Xie. "Self-Supervised Graph Learning for Recommendation". In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '21. Association for Computing Machinery, Virtual Event, Canada, 2021, pp. 726–735. ISBN: 9781450380379. DOI: 10.1145/3404835.3462862. URL: https://doi.org/10.1145/3404835.3462862.

25. L. Wu, J. Li, P. Sun, R. Hong, Y. Ge, and M. Wang. "DiffNet++: A Neural Influence and Interest Diffusion Network for Social Recommendation". *IEEE Transactions on Knowledge and Data Engineering* 34:10, 2022, pp. 4753–4766. DOI: 10.1109/TKDE.2020.3048414.

26. J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou. "GNNLab: A Factored System for Sample-Based GNN Training over GPUs". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Association for Computing Machinery, Rennes, France, 2022, pp. 417–434. ISBN: 9781450391627. DOI: 10.1145/3492321.3519557. URL: https://doi.org/10.1145/3492321.3519557.

27. R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. "Graph Convolutional Neural Networks for Web-Scale Recommender Systems". *CoRR* abs/1806.01973, 2018. arXiv: 1806.01973. URL: http://arxiv.org/abs/1806.01973.

28. X.-M. Zhang, L. Liang, L. Liu, and M.-J. Tang. "Graph Neural Networks and Their Current Applications in Bioinformatics". *Frontiers in Genetics* 12, 2021.