Accelerating GNN Inference with Dynamic GPU Feature Caches

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. GNNs are typically used for graphs that associate high-dimensional feature vectors with each node. Prior work has identified that efficiently moving these graph features to GPUs for computation, known as data loading, is a key challenge when training GNNs. Thus, statically caching graph features in GPU memory has been proposed for GNN training systems.

We observe that the data loading problem is exacerbated at inference time and becomes a disproportionate bottleneck when serving inference requests, even with static caches. We motivate the use of dynamic caches that swap node features in and out of GPU memory to exploit graph locality of inference requests. We demonstrate that a simple frequency-based cache admission and eviction policy can achieve better cache hit rates than degree-based static cache baselines. To alleviate overheads due to dynamic cache updates, our system performs cache updates asynchronously, removing these operations from the latency-sensitive request-response path. We extend our approach to also support a logical cache spanning multiple GPUs connected by NVLink and propose a lock-free synchronization mechanism to reduce potential lock contention due to cache updates.

Contents

1 Introduction

Graphs are increasingly popular and highly expressive structures for representing data. In the past decade, significant interest in graph analysis has led to the emergence of Graph Neural Networks (GNNs), class of state-of-the-art machine learning methods for graph representation learning.

GNNs adopt ideas from traditional Deep Neural Networks (DNNs) and combine them with techniques to capture structural information about graphs. Traditional DNNs have excelled in various tasks in areas such as computer vision [17][25] and natural language processing [11][18]. In these domains, however, inputs exhibit fairly regular structure, unlike in graphs.

To bridge the gap between DNNs and graph-structured data, GNNs capture graph structural information by using graph *convolutions*, a technique for aggregating local node neighborhood information. As opposed to traditional graph processing, graphs used with GNNs associate *features* with each node in the graph, which are large multidimensional tensors. 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.

GNNs are deployed in production systems for many applications, including bioinformatics [33] [24], traffic forecasting [26] [5] [15], recommendation systems [32] [30][13][27][29][9], cybersecurity [16] [21], and combinatorial optimization [10][3], among many others. Although there has been significant work on GNN training systems [todo cite stuff], GNN inference is relatively understudied.

Many existing GNN inference systems rely on approximate nearest neighbor approaches or periodic offline inference to keep node embeddings fresh [32] [9]. However, such approaches may not meet accuracy, latency, or throughput demands of real world systems. For example, in 2013 Facebook's graph database was updated roughly eight-six thousand times per second [2].

To understand the current state of the art for online GNN inference, we first examine how key optimizations in GNN training systems can apply to GNN inference. One critical optimization we identify is node *feature caching*, the act of storing some node features in GPU memory to avoid redundant host-device transfers. However, only some caching techniques from GNN training systems are applicable at inference time. The simplest of these is static degree-based caching, which permanently places features corresponding to nodes with highest out-degree in GPU memory [19]. We observe that even when using this static cache, GNN inference still encounters the *data loading* bottleneck, where 30-80% of inference latency is due to copying node features to the GPU. Although improving cache hit rate would help alleviate this bottleneck, no GNN training systems implement dynamic caches, even using traditional LRU or LFU cache eviction policies, due to the associated overhead.

We tackle the challenges associated with efficiently implementing dynamic caches in three 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 then propose a simple frequency-based admission and eviction policy provides 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 separate 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 Motivation

2.1 Graph Neural Networks

Given a graph containing nodes, edges, and features, Graph Neural Networks (GNNs) output a per-node *embedding* for each node in the graph. An embedding is a d-dimensional representation of aggregated feature information from a node's neighborhood. Similarity in the embedding space has different meaning based on the learning task, such as similarity of node type (a node classification task) or likelihood of edge existence (a 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 [8], 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 previous layer embeddings of node v's neighborhood. $\texttt{COMBINE}^{(k)}$ is a function that combines that the result of $\texttt{AGG}^{(k)}$ with the v's own last layer embedding. The first layer embedding for node v, h_k^0, is just its original features.

[todo mention that this stuff is dependent on GNN architecture]

Since for each layer there is a different W^k, GNNs are comprised of k neural networks. Furthermore, each layer requires recursive computation on a node's neighbors. Thus a *k-hop neighborhood* must be constructed to run a k-layer GNN. Generally GNNs use 1-5 layers, with 2 layers being a de facto standard [1]. However, some architectures can use significantly more layers, such as the current SOTA EnGCN model comprising 8 layers [6]. Figure 2.3 illustrates the construction of a *computation graph* for a 2-layer GNN. A computation graph describes how necessary nodes and edges participate in GNN computation.

2.2 Online GNN Inference

Traditionally, GNN inference has been viewed as an *offline* problem, where inference is performed on all nodes in the graph (full graph inference). Full graph inference is typically used for evaluating trained models or computing node embeddings for future lookup. For example, PinSage [32]

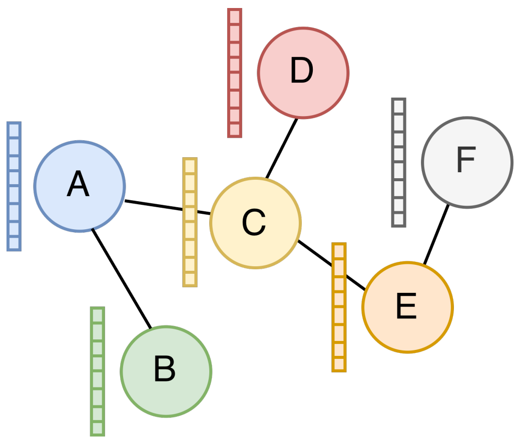

Figure 2.1: Graph structure and associated node features (colored bars).

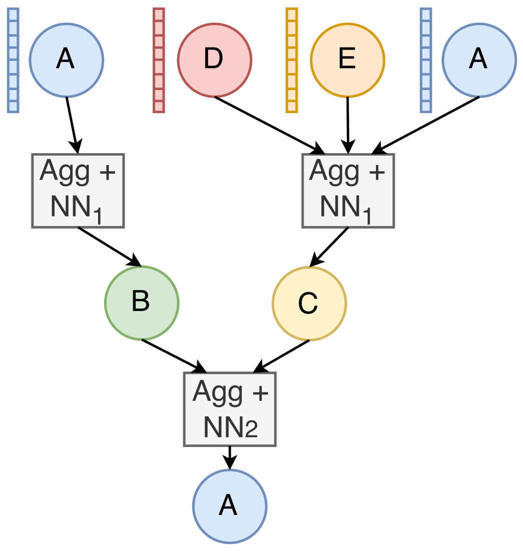

Figure 2.2: 2-hop neighborhood and computation graph for node A, ignoring self loops.

Figure 2.3: Graph and associated computation graph for node A.

first uses MapReduce [4] to perform full graph inference before storing all node embeddings in a database. Then, PinSage uses K-nearest neighbors to compute embeddings for new queries, enabling it to serve online recommendation requests. However, this approach, along with other nearest neighbor approaches, suffers from a loss in accuracy compared to directly using a GNN to compute the new embedding.

Therefore, in this work we will view GNN inference as a *online* problem, where a GNN is given a request to compute an embedding for a node or batch of nodes. In this setting, the GNN must respond to requests that consist of new nodes, their features, and edges connecting them into the existing graph.

In this section we motivate the online inference formulation and present a concrete taxonomy of the stages of GNN inference.

2.2.1 Online Inference Applications

Online inference has many applications depending on domain. For example, in a social network graph, an inference request can correspond to computing the embedding for a new user or recomputing embeddings as a result of a new friendship. Furthermore, there is no strict requirement that a node is truly "new", meaning that an inference request could correspond to an update of node features. For example, in a traffic forecasting application, an inference request can be an update of node features that represent a change in traffic conditions.

Facebook's social network graph in 2013 experienced roughly 86 thousand node or edge updates per second [2]. [todo put this in context]

2.2.2 GNN Inference Stages

While online GNN inference is generally understudied, it shares many similarities with GNN mini-batch training (discussed in Section 2.3).

Thus when understanding the steps required to perform GNN inference, we use established taxonomy from mini-batch training work [19][31][8]. We break down the stages of GNN inference as follows:

1. **Sampling:** Construct k-hop neighborhood for target nodes and build logical computation graph describing GNN computation.
2. **Data Loading:** Moving necessary data to GPU, comprising two steps.
   a) **Feature gather:** Gather node features corresponding to k-hop neighborhood in contiguous CPU buffer.
   b) **CPU-GPU copy:** Copy buffer with node features and computation graph to GPU.
3. **Model execution:** Perform GNN computation on GPU.

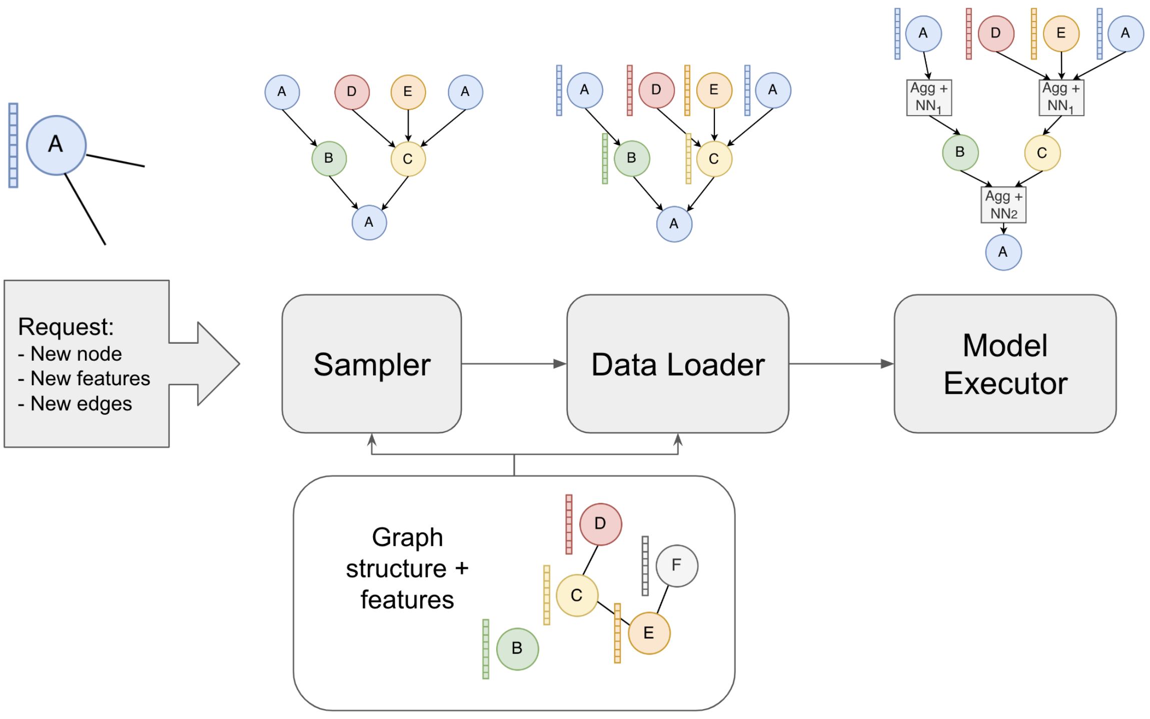

Figure 2.4: Online GNN Inference

2.3 Inference vs. Training

In this section we analyze similarities and differences between the inference and training tasks and examine the effectiveness of relevant GNN training optimizations at inference time.

Mini-batch training is a popular technique for GNN training on large graphs where embeddings are only computed for a random subset of the graph per epoch [20]. This is as opposed to full graph training, where the embeddings and gradients are computed for the entire graph at once, similar to full graph inference. Mini-batch training is analogous to online inference, and, aside from the high-level goal, the difference is that no backpropagation is needed for inference. We briefly look at prior optimizations for the sampling and data loading stages, with particular emphasis on the latter.

Sampling Optimizations

For example, *neighborhood sampling* reduces or avoids the exponential explosion in neighborhood size by randomly selecting a fixed number or percentage of node neighbors during the sampling stage [12]. However, since this can produce a drop in accuracy, works such as NextDoor [14] have proposed performing sampling on GPU rather than CPU, yielding significant speedups.

We are particularly interested in works that target the data loading stage, since movement of features from host memory to GPU memory can easily be bottlenecked by PCIe bandwidth. A key innovation in GNN training has been caching node features in GPU memory so they no longer need to be copied over from host memory, easing data loading bottlenecks. We give a detailed treatment of three relevant works, **PaGraph**, **GNNLab**, and **BGL**.

Data Loading Optimizations

Prior GNN training works have observed that both GPU compute and memory are underutilized while training. **PaGraph** [19] 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** [31] extends static caching 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** [20] introduces a dynamic FIFO cache and iterates over the graph in a roughly-BFS manner to exploit the FIFO cache. However, while these caching approaches work well for training, they are not all directly applicable to the inference case.

2.3.1 GNN Inference Challenges

Leveraging approaches from existing training systems present several key challenges, namely:

1. **Latency is a key metric at inference time**. This is not the case during training. During training, throughput is a far more important metric than latency. For example, many training systems avoid data loading bottlenecks using pipelining. However, pipelining cannot hide latency.

2. **Node ordering cannot be controlled**. While training systems can use

3. **No backpropagation leads to sampling and data loading dominating inference latency**.

Reasons why we optimize data lodaing stage

1. **Data loading comprises 20-80% of inference latency**
2.
3. **Data loading latency cannot be hidden with pipelining**
4. **Data loading overhead scales with feature dimension**

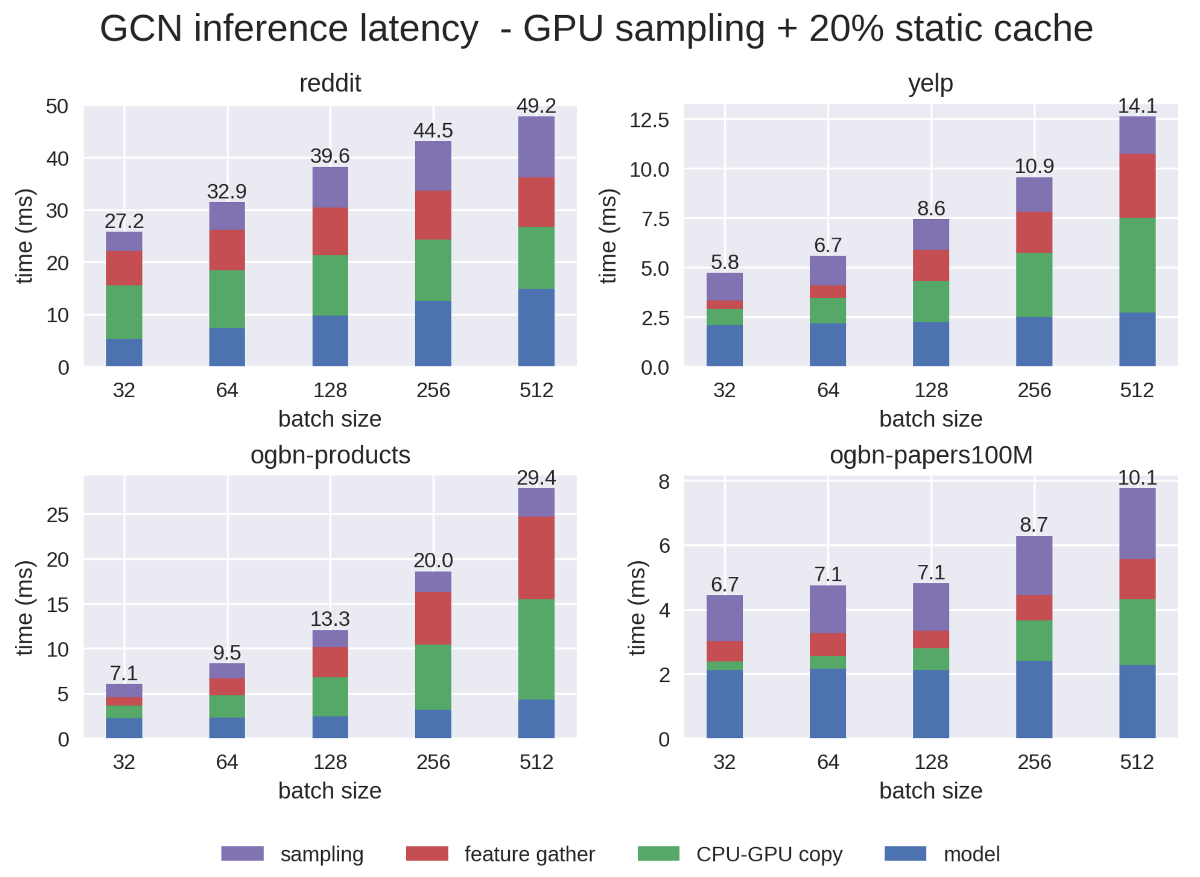

Figure 2.5: Inference latencies for different graph datasets and request batch sizes (number of target nodes in request). Requests are served by a system using GPU sampling and a static cache large enough to hold 20% of the graph's node features.

3 Design

Our system leverages a novel approach to dynamic cache updates to serve GNN inference requests with low-latency. We target single-machine, multi-GPU inference systems and build upon caching techniques in GNN training systems discussed in the previous chapter. In this chapter, we address three key research questions:

RQ1: How can GPU feature caches effectively capture GNN inference patterns?
Section 3.1 describes opportunities for dynamic caches to outperform existing static caches at inference time and highlights shortcomings of naive approaches. Section 3.2 proposes a frequency-based cache admission and eviction policy that produces better cache hit rates that static baselines while offering a path towards efficient update operations.

RQ2: How can the impact of dynamic cache updates on request-response latency be minimized?
Section 3.3 details how we derive an asynchronous cache update mechanism based on profiling of a naive cache update mechanism using "prefetching". By carefully engineering this asynchronous cache update, we are able to hide cache update operations that would otherwise negatively impact tail latency.

RQ3: How can we effectively leverage concurrency (multithreading, multi-GPU) to produce scalable inference?
Since asynchronous updates synchronized using naive locking can produce blocking behaviors when pipelined (and thus no longer be truly asynchronous), we propose a lock-free mechanism to perform cache updates, discussed in Section 3.4. Lastly, Section 3.5 describes how we extend our system to support multiple GPUs connected by NVLinks and share a single logical cache.

3.1 Towards Dynamic Caching

One of our key observations is that effective caching is pivotal to reducing data loading costs, and dynamic cache policies can enable better cache hit rates. We define a *dynamic* cache policy as one that swaps node features in and out of GPU memory over time. We identify two key opportunities that dynamic caches can capture that static caches neglect:

Inference request locality Inference requests generate large neighborhoods during the k-hop neighborhood generation process. Due to this, if inference requests being "close" in the graph have reasonable semantic meaning in the real world, then we can expect inference requests to exhibit locality. For example, in a social network graph there may be clusters of users who are in a similar geographic area. These users may have more activity and generate more inference requests during the daytime, meaning that subgraphs become "hot" at different points in time. Prior work in the graph processing space has also noted the importance of request locality in domains such as traffic prediction or knowledge graph mining [22].

Sampling patterns Since GNNs require k-hop neighborhoods, a policy that caches node features based solely on node out-degree can neglect nodes that are actually likely to be used. For example, a low-degree node that is directly adjacent to several high-degree nodes is likely to be similarly "hot" to its high-degree neighbors. Additionally, certain GNN architectures leverage specific parameters when building neighborhoods, such as by assigning edge weights [cite edge weight].

3.1.1 Why Traditional Dynamic Caches are Ineffective

An intuitive first step towards dynamic caches is to consider using traditional cache eviction policies such as LRU, LFU, or FIFO. However, many of these approaches have too much overhead to be effective for GNN inference.

In the GNN inference case, each request (comprising anywhere from one to several hundred target nodes) can generate k-hop neighborhoods of hundreds of thousands of nodes. As a result, the overhead of cache replacement heuristics can quickly overtake any performance gains from improved cache hit rates. For example, the traditional implementation of an LRU cache using a linked list and hash table to track elements will severely bottleneck GNN inference. Consider the following back of the envelope calculation. We find that a naive GNN inference system can generally serve requests with < 100 ms latency. Assuming a cache put or get takes only 500 ns, as is the case with many publicly available LRU cache implementations [todo cite https://github.com/hashicorp/golang-lru], for one request this requires $100,000$ nodes $*$ 500 ns $= 50$ ms. Given that this would add at least 50% to our original inference latency, such overhead is clearly unacceptable.

To handle potentially huge neighborhood sizes, a key requirement for a cache policy is to be easily parallelizable. A frequency-based heuristic meets this criteria and has been shown to be effective in the GNN setting a training time, with GNNLab's pre-sampling approach [31]. Even so, the traditional LFU policy can still struggle versus a static cache baseline as it requires some kind of sorting or top-k operation for each request served.

Furthermore, large neighborhood sizes require us to also consider a cache admission policy. If only a cache eviction policy is used, it is easy for a "one-hit wonder" to be brought in among hundreds of thousands of other nodes and waste cache space.

Note that static caches do not suffer from these performance problems since checking for cache hits is easily implemented using tensor operations.

3.2 Frequency-based Admission & Eviction Policy

Motivated by our observations in the previous section, in this section we propose a simple frequency-based cache admission and eviction policy. Then, we briefly illustrate the improved cache hit rates of this policy when using a naive, strawman cache update mechanism.

The goal of our policy is straightforward: to admit the most frequently occuring node features and evict the least frequent node features.

In our implementation node frequencies are tracked in a buffer in GPU memory. By tracking frequencies on the GPU rather than the host, our system avoids an additional device to host copy, since computation graphs are built on GPU. The frequency buffer has length equal to the number of nodes in the graph. Each index in the buffer corresponds to a node, and the value is a counter that gets incremented whenever the node's feature is required. To reduce memory usage, this buffer uses only one byte for each node. However, the size of the buffer still scales with the number of nodes in the graph. We note that frequencies can also be tracked using a probabilistic data structure like a counting bloom filter or count-min sketch, but we do not implement this. Using such a probabilistic data structure actually makes it easier to add new nodes into the graph, since there is no buffer that needs to be resized. We leave this as future work.

To capture changes in node frequencies, the count buffer must decay over time. This is implemented by periodically dividing all counts in the buffer by two, a technique adapted from TinyLFU [7] which produces exponential decay. A nice property of exponential decay is that it is easy to bound the maximum possible count and fit it within the one byte constraint. Additionally, the decay can be implemented as a bit shift for better performance and still works with a count-min sketch or counting bloom filter.

There are many policy choices regarding how exactly to choose which nodes to evict or admit. For example, a policy could weight both a node's degree as well as its recent frequency when making admission decisions. The weighting of node degree versus frequency is a user-adjustable knob in our system, but for the evaluation in Chapter 5, we use only frequency when evaluating frequency-based approaches.

3.2.1 Strawman Prefetching Mechanism

Given the above policy, an actual implementation must choose some mechanism by which to perform cache updates and perform the frequency calculations. For example, LFU is traditionally implemented by tracking frequency using a heap and evicting/admitting into the cache per-request, but as we saw earlier this can negatively harm inference latency. We present an alternative strawman mechanism essentially replaces the static cache with a new one every k requests according to the above policy. We call this alternative baseline mechanism our *prefetching* strawman.

In particular, every k requests the cache is entirely replaced with the most common nodes that appeared in the previous k requests. This means that node features are pulled to the GPU feature cache and request handling must be paused as necessary.

The key insight with this strawman is that while for most requests it maintains the low overhead nature of a static cache with improved cache hit rates, but every k requests it incurs a large penalty due to a cache update. This cache update penalty produces significant tail latency, which we analyze in the next section.

3.3 Asynchronous Cache Update mechanism

To eliminate tail latencies associated with the prefetching strawman, our system uses an asynchronous cache update mechanism, moving cache update operations off the critical path when responding to inference requests. Table 3.1 provides a breakdown of average cache update overheads for a selected dataset, ogbn-products, during a single-threaded inference execution.

Breakdown of Cache Update Overhead

Operation	**Time (ms)**	**Percent of Update Time**
Update cache metadata	2.7	47%
Feature copy	2.2	38%
Compute most common features	0.6	10%
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.

The largest contributors to cache update overhead in the prefetching strawman are copying new features from host memory to GPU memory and updating cache metadata. Using this profiling information, we motivate three design decisions. **(1)** Rather than prefetching features to move into GPU memory, we move features into the cache only when they are needed by inference requests, similar to traditional cache eviction policies. **(2)** To sidestep overheads due to updating cache metadata, we move metadata updates to a separate host thread and CUDA stream (synchronization is discussed in Section 3.4). **(3)** Lastly, we can compute the most common features (in practice a top-k operation) in a separate CUDA stream.

3.3.1 Computing Cache Candidates

To enable **(1)** moving features into the cache only when they are needed by inference requests while adhering to the desired policy, we introduce the idea of *cache candidates*, a set of node ids computed every k requests. When a cache miss occurs and new node features are copied to the GPU from host memory, they are checked against the set of cache candidates. If a feature corresponds to a node that is a cache candidate, then it will a non-cache candidate present in the cache. This is possible since at any given point in time, the number of cache candidates is equal to the size of the cache.

3.3.2 Performing Cache Updates

The actual cache update itself is handled by a separate host thread and CUDA stream than the one handling inference requests.

One important aspect of this approach is that the cache update should occur during the model forward pass. The key insight is that node features are already present in GPU memory for the model forward pass and thus are assumed to fit in GPU memory fine. However, if the asynchronous update thread holds on to these tensors longer than the model forward pass would normally take, memory usage can be inflated. We avoid this problem by allowing the model forward pass to have full ownership of any node features required for computation. If the model forward pass for a given inference request is completed and the cache update has not started, then the cache update will be ignored.

To further avoid contention of GPU compute due to cache updates happening concurrently with model computation, we assign the cache update operations to a lower priority CUDA stream than model computation.

3.4 Lock-free Cache Updates

Asynchronous cache updates naturally raise concerns about correctness and performance due to concurrency. While naive locking may initially be adequate, at scale this may not be the case. In this section we look at a novel lock-free approach using *masking* to avoid lock contention due to cache updates.

Consider the case where we would like our system to be pipelined to maximize throughput. Since the data loading stage requires reading from the cache, we must be careful about synchronization between cache updates and data loading. A naive approach is to use mutual exclusion, but this can lead asynchronous updates having equivalent performance to the original synchronous variety. Figure 3.1 illustrates this effect. In this example, by enforcing mutual exclusion between the data loading thread and cache update thread, the second inference request is forced to wait on the cache update from the first inference request, meaning the latency of the first cache update was simply "passed along".

3.4.1 Masked Updates

To our cache data structures we add a *mask* tensor, which indicates for each node in the graph whether it is present in the cache or not.

Readers

1. Increment atomic
2. Check cache mask
3. Perform cache reading
4. Increment atomic

writers

1. Write 0 to cache mask
2. Check atomics
3. Wait on atomics
4. Perform cache update
5. Write 1 to cache mask

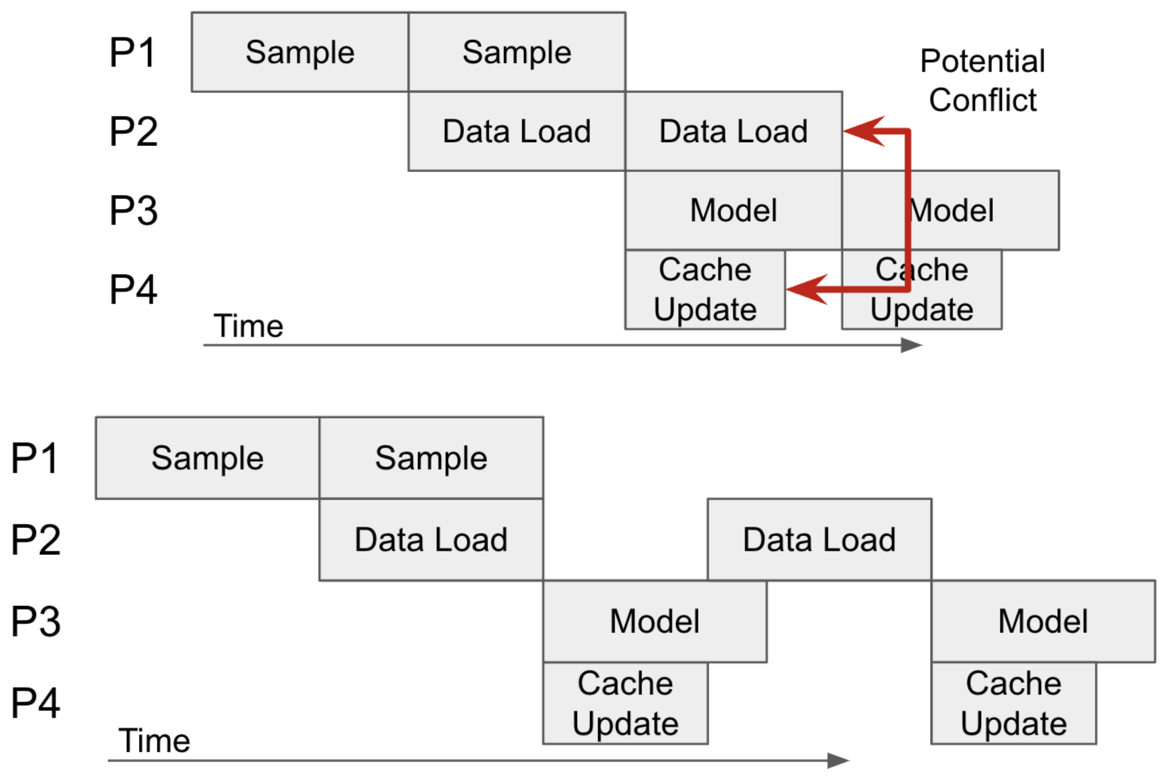

Figure 3.1: [todo Lock conflict p99 latency]

3.5 Multi-GPU Cache Sharing

We extend our solution to support a single logical feature cache that is shared among multiple
say how fast NVLink is

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

2.

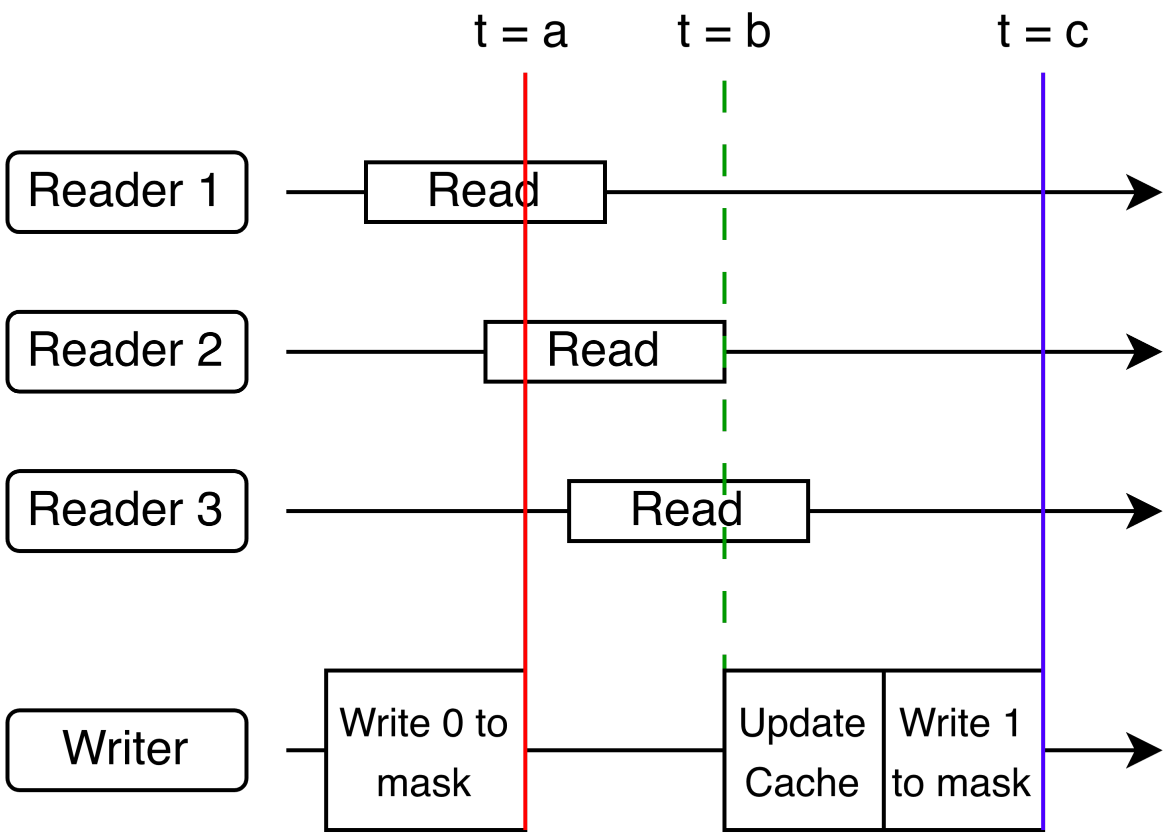

Figure 3.2: Readers and writers with lock-free cache update

4 Implementation

We implement our design using Deep Graph Library (DGL) [28], a GNN training framework; PyTorch [23] a tensor operation library, and a mix of Python and C++. Our current implementation consists of roughly 12,000 SLOC of Python and 1,000 SLOC of C++, and is publicly available here [todo hyperlink].

1. **Persistent pinned memory buffer for feature transfer:**
2. **GPU Sampling**
3. **CUDA Multi-Process Serivce (MPS)**

GPU sampling Feature gather pinned buffer Multi process service

Figure 4.1: [todo add picture of this system]

4.1 Limitations

[TODO move this elsewhere] Our system currently does not combine new inference requests into the existing graph or retrain the GNN to accommodate for new requests. Instead, we look only at GNN computation and investigate how to efficiently compute new embeddings. Integrating new nodes into the existing graph and dealing with challenges such as consistency are both orthogonal and out of scope of this work, but would be an interesting and natural extension.

5 Evaluation

5.1 Experimental Setup

1. List all hardware, os, pytorch version, Dgl version

1. Data transfer size graphs also?

5.2 Datasets

Dataset	Nodes	Edges	Features	Avg. Degree
reddit	200K	111M	602	492
yelp	700K	13M	500	10
ogbn-products	2.4M	124M	100	51.7
ogbn-papers100M	111M	1.6B	128	14.4

Table 5.1: Information about graph datasets used in evaluation

5.3 Policy Evaluation

5.3.1 Latency

Figure 5.1: [todo throughput]

5.3.2 Cache Hit Rate

Figure 5.2: [todo throughput]

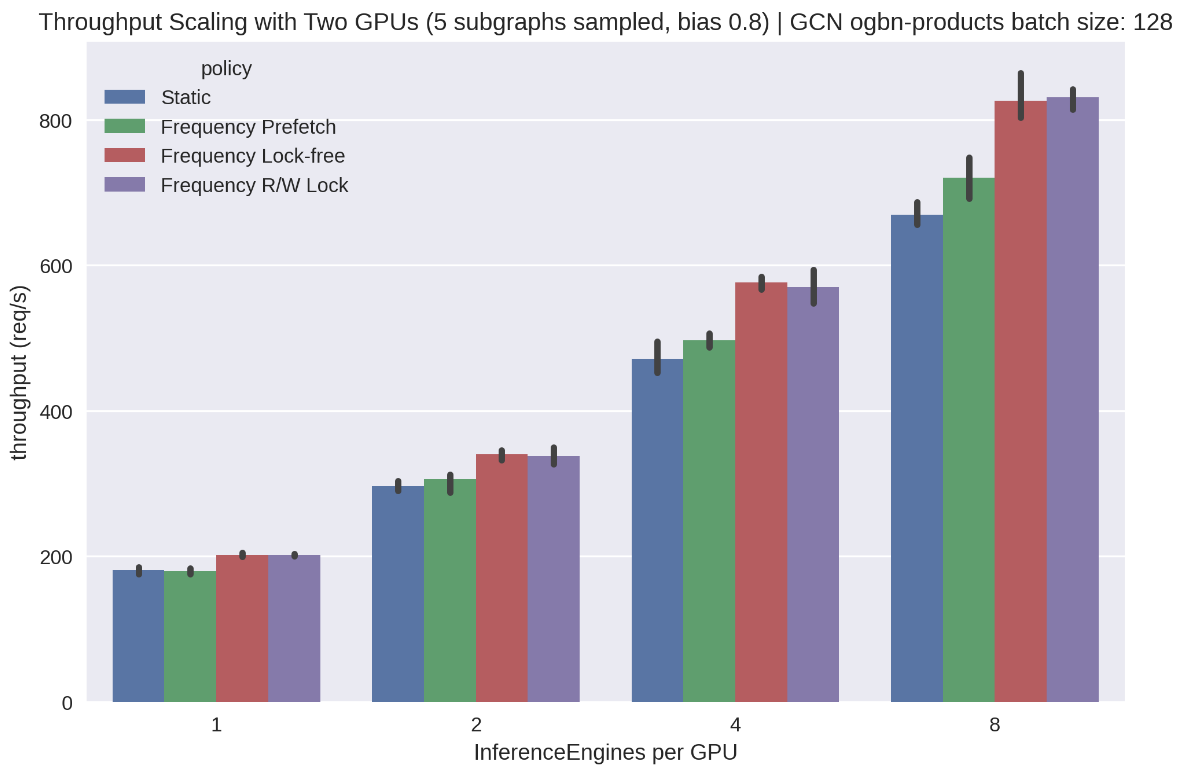

Figure 5.3: [todo throughput]

5.3.3 Throughput

5.4 Locking vs. Lock-free

Figure 5.3 compares the throughput of our system using different cache replacement policies and mechanisms. ...

Although there is little discernable difference in throughput between the lock-free and R/W lock approaches, there is an impact on P99 latency. Figure 5.4 demonstrates these differences. Note the log scale.

To evaluate our system Results are similar for the subgraph biased case.

5.5 PyTorch Direct Experiments

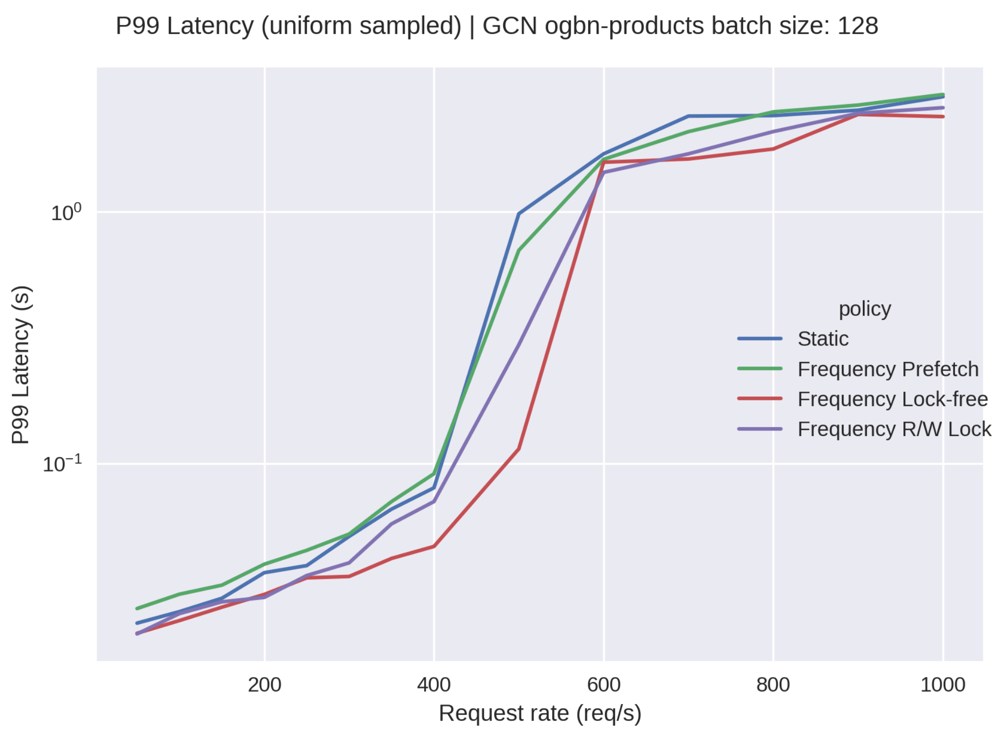

Figure 5.4: 99th percentile latencies for varying request rates. Tested on system using both GPUs and eight InferenceEngines per GPU.

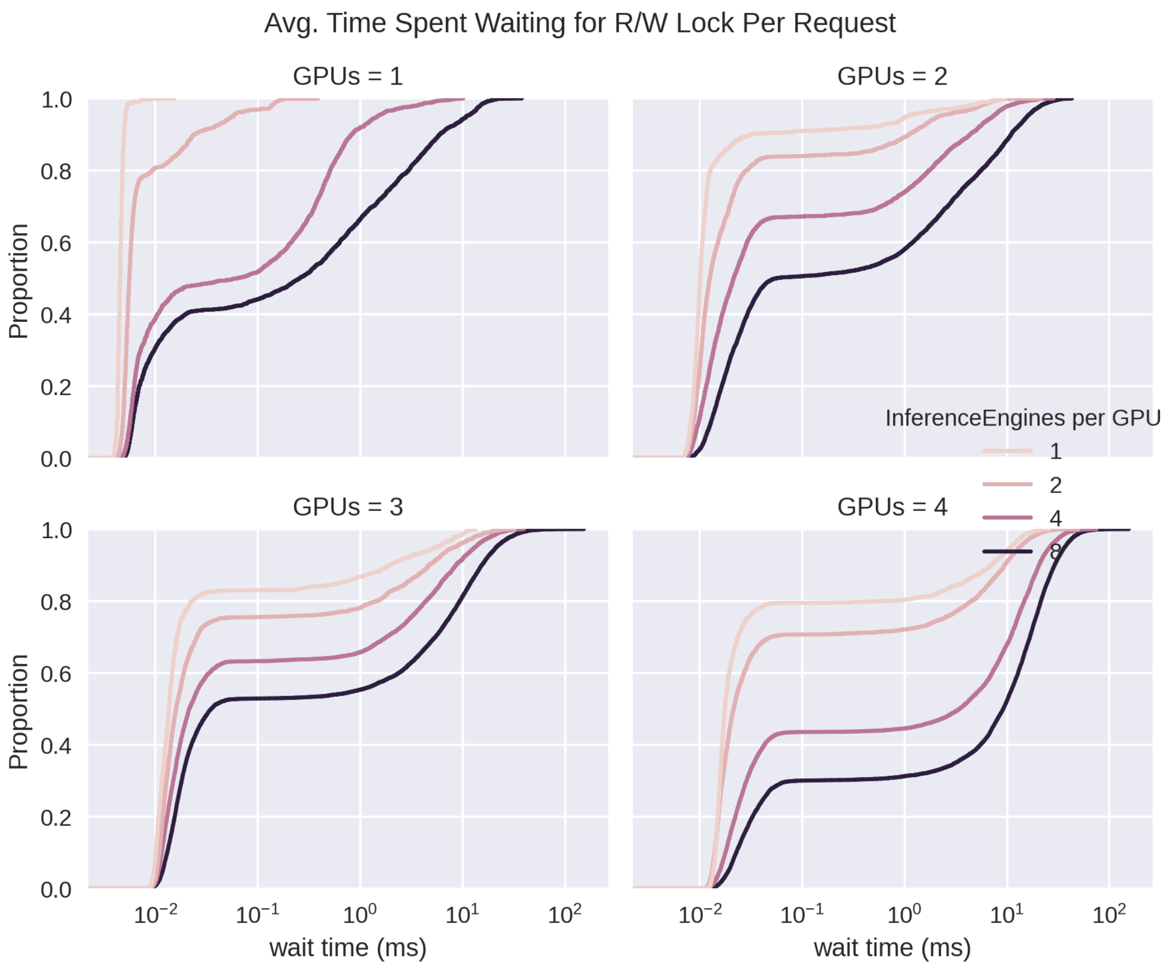

Figure 5.5: [todo Lock conflict microbenchmark]

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. W. Y. H. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. Elbyed. "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems". *Distributed Parallel Databases* 38:2, 2020, pp. 495–530. DOI: 10.1007/s10619-019-07276-9. URL: https://doi.org/10.1007/s10619-019-07276-9.
3. 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.
4. J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.
5. 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.
6. 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].
7. G. Einziger and R. Friedman. "TinyLFU: A Highly Efficient Cache Admission Policy". In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2014, pp. 146–153. DOI: 10.1109/PDP.2014.34.
8. 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.
9. 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.

10. 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.

11. A. Graves. "Generating Sequences With Recurrent Neural Networks". *CoRR* abs/1308.0850, 2013. arXiv: 1308.0850. URL: http://arxiv.org/abs/1308.0850.

12. 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.

13. 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.

14. A. Jangda, S. Polisetty, A. Guha, and M. Serafini. "Accelerating Graph Sampling for Graph Machine Learning Using GPUs". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Association for Computing Machinery, Online Event, United Kingdom, 2021. ISBN: 9781450383349. DOI: 10.1145/3447786.3456244. URL: https://doi.org/10.1145/3447786.3456244.

15. 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.

16. 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.

17. 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.

18. 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.

19. 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.

20. 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.

21. 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.

22. C. Mayer, R. Mayer, J. Grunert, K. Rothermel, and M. A. Tariq. "Q-Graph: Preserving Query Locality in Multi-Query Graph Processing". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA '18. Association for Computing Machinery, Houston, Texas, 2018. ISBN: 9781450356954. DOI: 10.1145/3210259.3210265. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3210259.3210265.

23. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

24. 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.

25. 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.

26. 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.

27. 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.

28. M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks". *arXiv preprint arXiv:1909.01315*, 2019.

29. 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.

30. 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.

31. 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.

32. 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.

33. 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.