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
B.S. Computer Science, Turing Scholars Honors
at
The 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 with 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 1

2 Background and Motivation 3
- 2.1 Graph Neural Networks . . . 3
- 2.2 Online GNN Inference . . . 3
  - 2.2.1 Online Inference Applications . . . 4
  - 2.2.2 GNN Inference Stages . . . 4
- 2.3 Inference vs. Training . . . 5
  - 2.3.1 GNN Inference Challenges . . . 6

3 Design 8
- 3.1 Towards Dynamic Caching . . . 8
  - 3.1.1 Why Traditional Dynamic Caches are Ineffective . . . 9
- 3.2 Frequency-based Admission & Eviction Policy . . . 9
  - 3.2.1 Strawman Prefetching Mechanism . . . 10
- 3.3 Asynchronous Cache Update mechanism . . . 10
  - 3.3.1 Computing Cache Candidates . . . 11
  - 3.3.2 Performing Cache Updates . . . 11
- 3.4 Lock-free Cache Updates . . . 11
  - 3.4.1 Masked Updates . . . 12
- 3.5 Multi-GPU Cache Sharing . . . 13

4 Implementation 15
- 4.1 Limitations . . . 16

5 Evaluation 17
- 5.1 Experimental Setup . . . 17
  - 5.1.1 Environment . . . 17
  - 5.1.2 Datasets & Workloads . . . 17
  - 5.1.3 Policies Evaluated . . . 18
- 5.2 End-to-End Latency . . . 18
- 5.3 Locking vs. Lock-free . . . 20
  - 5.3.1 Throughput . . . 21
  - 5.3.2 P99 Latency . . . 22
  - 5.3.3 Lock Conflict Microbenchmark . . . 22

6 Conclusion 24
- 6.1 Future Work . . . 24
- 6.2 Acknowledgements . . . 24

Bibliography 25

1 Introduction

Graphs are highly expressive and increasingly popular structures for representing data. In the past decade, significant interest in graph analysis has led to the emergence of Graph Neural Networks (GNNs), a 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 [26][38] and natural language processing [16][27]. 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 using graph *convolutions*, a technique for aggregating local node neighborhood information. As opposed to traditional graph processing, graphs used with GNNs have per-node *features*, which are large multidimensional tensors. GNNs use these features to compute *embeddings* for each node by recursively aggregating each node's neighboring features and feeding these aggregations into traditional neural networks. The resulting embeddings can be used for tasks such as node classification, link prediction, or graph classification.

GNNs have practical applications in many domains, including bioinformatics [49] [37], traffic forecasting [39] [8] [22], recommendation systems [46] [44][18][40][43][13], cybersecurity [24] [30], and combinatorial optimization [14][4], among many others. Although there has been significant work on GNN training systems [12][28][29][45], 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 [46] [13]. 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 eighty-six thousand times per second [2]. Maintaining up-to-date GNN-generated node embeddings would require similar throughput from a GNN inference system.

An intuitive approach for efficient inference is to borrow optimizations from GNN training systems, but existing approaches are not suitable for inference. One critical optimization we identify is *feature caching*, the act of storing node features in GPU memory to avoid redundant host-device transfers. The simplest approach to this is static, degree-based caching, which permanently places features corresponding to the highest out-degree nodes in GPU memory [28]. We observe that even when using this static cache approach, GNN inference still encounters a data loading bottleneck, where 30-80% of inference latency comes from copying node features to the GPU.

We tackle the challenges associated with improving feature caches and reducing data loading overheads 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. We posit that inference requests may actually have graph locality due to correlation with real world trends. For example, a traffic forecasting application may experience a drastic rise in inference requests around a city center due to rapidly shifting traffic patterns during rush hour. Dynamic caches can capture the "hot" subgraph corresponding to said city center while static caches cannot. Following this observation, 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 inference traces corresponding to uniformly random requests as well as requests concentrated in hot subgraphs.

Second, we use asynchrony as a mechanism to avoid overheads due to dynamic cache updates. While dynamic caches can have better cache hit rates, existing GNN training systems have found that traditional cache eviction policies such as LRU or LFU have unacceptable overheads [28] [45]. Thus, end-to-end performance gains will be eroded by cache update and management overheads. We propose a mechanism that periodically computes a set of *cache candidates* and performs actual cache update operations to separate host threads and CUDA streams, taking these operations off the critical path.

Finally, we propose a lock-free mechanism for performing cache updates. In a highly concurrent system with multiple GPUs and many inference threads, naive locking can potentially lead to poor inference latency and throughput. For example, our system maximizes cache capacity by sharing a single logical cache among multiple GPUs connected by NVLink, and the blocking of inter-GPU communication will lead to increased latencies. Furthermore, a blocking, asynchronous cache update can degenerate into a synchronous one due to contention with other threads. To address this, we show how the cache can be *masked* to allow for wait-free cache readers and lock-free cache 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) [25] or likelihood of edge existence (a link prediction task) [48].

Borrowing notation from P3 [12], we can generally represent the embedding for a node v at layer k as h_v^k, where

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

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

$\texttt{AGG}^{(k)}$ 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.

The choice of $\texttt{AGG}^{(k)}$ and $\texttt{COMBINE}^{(k)}$ are depend on specific GNN architecture. Graph Convolutional Networks (GCN) [25], GraphSAGE [17], and Graph Attention Networks (GAT) [41] perform different types of aggregations and operations on features, but share the same general aggregation principle.

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 [9]. Figure 2.1 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 [46] first uses MapReduce [7] 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.

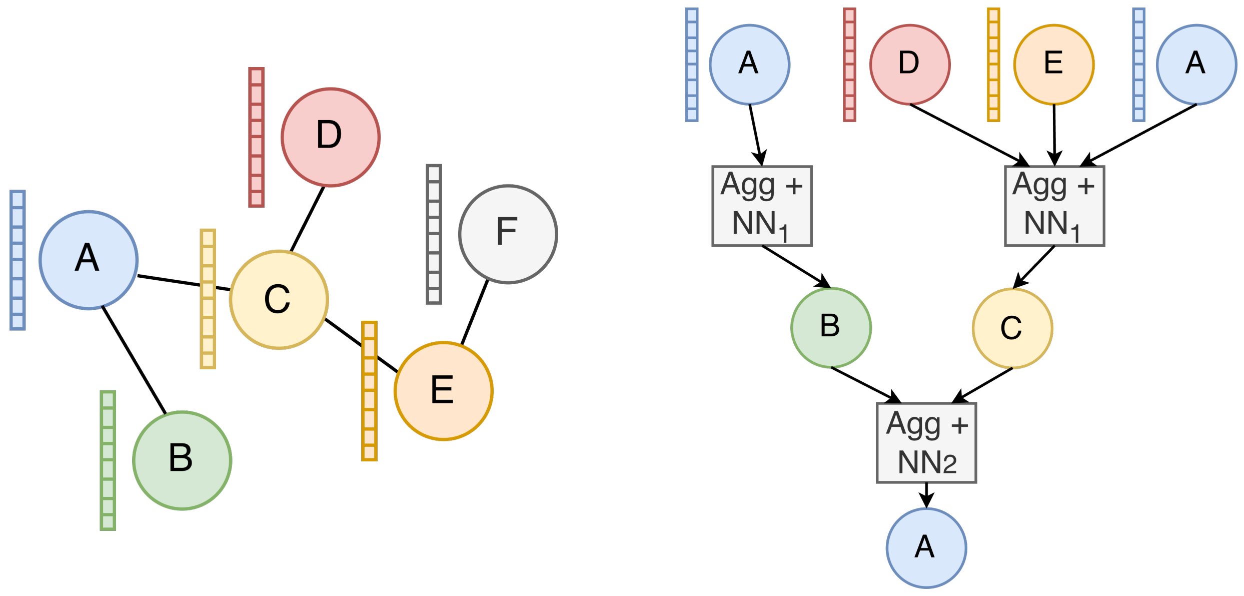


Figure 2.1: Graph and associated computation graph for node A.

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, requests consist of nodes, their features, and edges connecting them into the existing graph.

In this section we motivate this 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.

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 [28][45][12]. 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.

These stages are illustrated in FIgure 2.2.

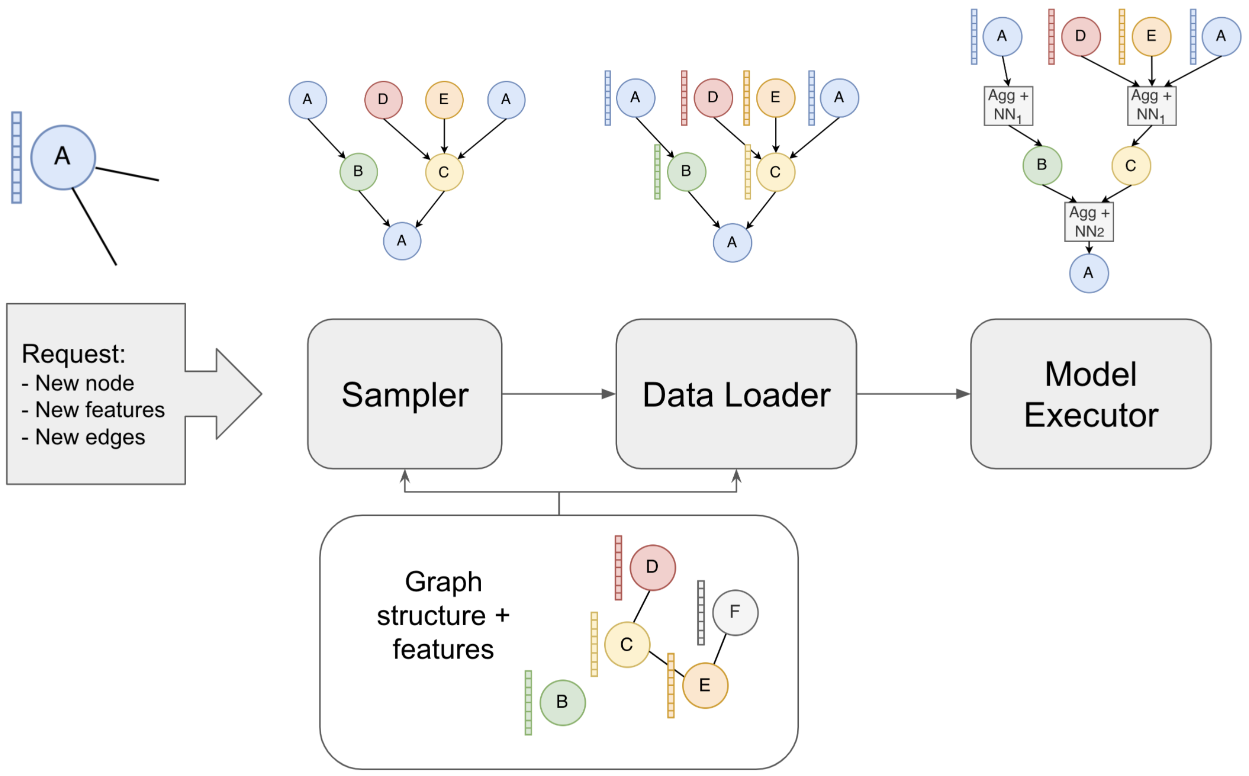


Figure 2.2: Online GNN Inference

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 [29]. This is as opposed to full graph training, where node 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

The *neighborhood explosion* problem is a well known issue in the sampling stage. Since the size of k-hop neighborhoods are inherently exponential, constructing these neighborhood and corresponding computation graphs can be expensive. *Neighborhood sampling* helps alleviate this problem by randomly selecting a fixed number or percentage of node neighbors during the sampling stage [17]. However, since this can produce a drop in model accuracy, works such as NextDoor [21] have proposed performing sampling on GPU rather than CPU, yielding significant speedups. In our system we will leverage this GPU sampling approach.

Data Loading Optimizations

Prior GNN training works have observed that data loading can also be a significant bottleneck. Bus bandwidth between host memory and GPUs can easily be saturated by GNN dataloading. For reference, PCIe 3.0 16x and PCIe 4.0 16x unidirectional bandwidth is 16 GB/s and 32 GB/s respectively. Meanwhile, given exponential neighborhood sizes and large feature dimensions, it is easy for transfers of several hundred megabytes to be required for each minibatch. Therefore, gathering these features in CPU memory and copying them to the GPU can bottleneck training pipelines.

Since GNN models have relatively few parameters compared to traditional DNNs, GPU compute and memory can actually be underutilized during training. Thus several works have proposed caching node features in GPU memory so they no longer need to be copied over from host memory. We are particularly interested in these following GNN training systems implementing feature caches, since we find that data loading is a several bottleneck during inference.

PaGraph [28] introduces *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. This cache is *static* since these features features remain permanently in GPU memory until training concludes. A static cache can be used for inference and is low overhead, but has worse cache hit rates than dynamic caches.

GNNLab [45] extends static caching to include a pre-sampling phase, where warmup epochs are run to determine what nodes are most often used and thus should be stored in the cache. Although the pre-sampling approach cannot be directly applied to inference, we build upon the idea of using frequency as a feature for determining cache residents in Section 3.2.

BGL [29] uses a dynamic FIFO cache and iterates over the graph in a roughly-BFS manner to exploit the FIFO cache. BGL's approach is reliant on controlling the node order during training time and thus cannot be applied to inference. However, BGL does introduce using NVLinks between GPUs to share cache resources, which we improve upon by enabling greater concurrency in Sections 3.5 and 3.4.

2.3.1 GNN Inference Challenges

We observe that after adopting the GPU sampling and static caching techniques from GNN training systems, inference latency is still dominated by data loading operations. Figure 2.3 illustrates these results.

We identify key challenges with GNN inference it critical to carefully apply data loading optimizations. In particular,

1. **Latency is a key metric at inference time**. During GNN training, throughput is far more important than latency. For example, many training systems try to hide data loading overheads by pipelining additional compute during data transfers. However, pipelining cannot hide latency. As a result, an efficient inference system must directly reduce data loading overhead.

2. **Data loading is the key inference bottleneck**. As Figure 2.3 shows, the data loading step is the bottleneck in terms of inference latency. This is actually mostly inference-specific, since at inference time there is no need to perform backpropagation, meaning that sampling and data loading comprise a larger percentage of inference latency. Furthermore, modern deep learning frameworks such as PyTorch [36] and TensorFlow [31] offer optimizations that can be enabled at inference time, such as not tracking gradients, which skews this proportion further.

As a result, we believe it is important to reduce data loading overheads by improving GPU feature caches.

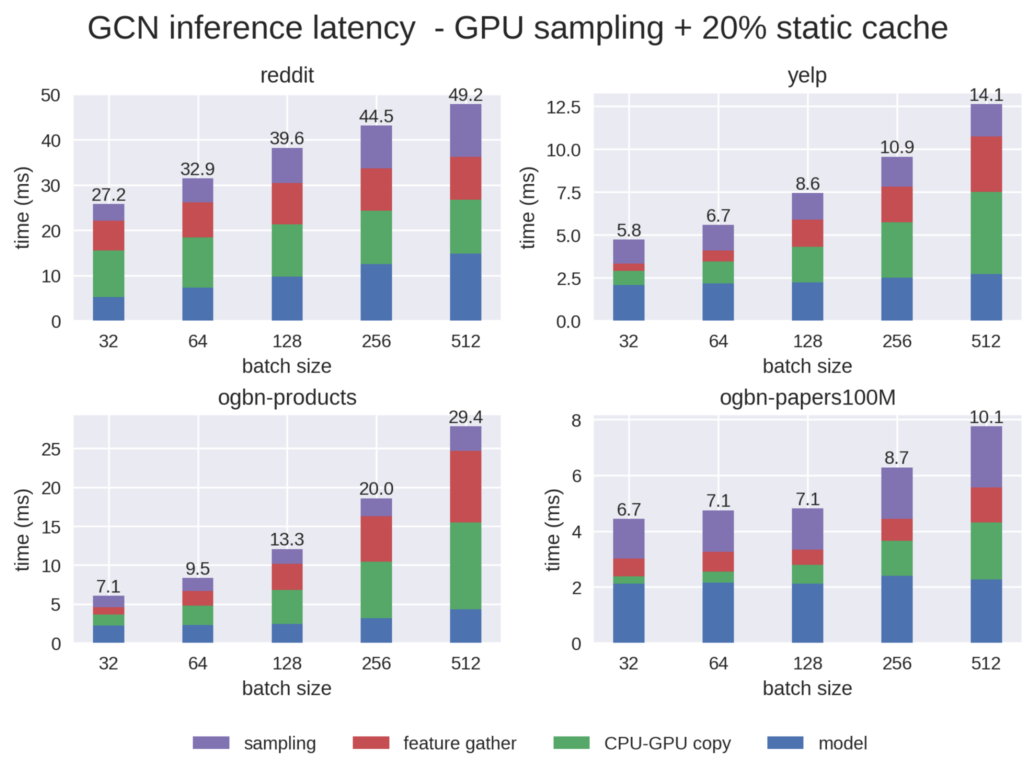


Figure 2.3: Inference latencies for different graph datasets and request batch sizes (number of target nodes in request). Requests are served by a system using GPU sampling and a static cache large enough to hold 20% of each graph dataset's node features. The test platform is detailed in Section 5.1.1

3 Design

Our system leverages a novel approach for efficient dynamic cache updates for improved inference latency and throughput. We design a single-machine, multi-GPU inference system and build upon caching techniques in GNN training systems discussed in the previous chapter. By focusing on single-machine inference, we can avoid network overheads and achieve better latency and throughput than distributed systems when graph size is appropriate.

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 [32].

Sampling patterns A degree-based policy only attempts to approximate the likelihood of encountering a node during inference. 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, due to GNNs requiring multi-hop neighborhoods. Additionally, certain GNN architectures may sample node neighbors using some other heuristics, such as sampling based on edge weight [41].

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 [15], 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 [45]. 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 occurring node features and evict the least frequent node features within a particular time window. Implementing this heuristic requires tracking node frequencies and decaying them over time.

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 GPU 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 [11] or count-min sketch

[5], 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 [10] 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.

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 top-k frequency calculations. For example, LFU is traditionally implemented by tracking most common elements using a heap and evicting/admitting into the cache per-request; however, as we saw earlier this can 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 idea is that this strawman maintains the low overhead nature of a static cache while improving 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. The prefetching-based update produces significant tail latency, in this case increasing response latency by more than a third when the update occurs.

Breakdown of Average 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	

Average inference latency without update: 12.7 ms

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 support 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, the new features are checked against the set of cache candidates. If a feature corresponds to a node that is a cache candidate, then it will replace 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

Cache metadata updates actually comprise the majority of the tail latency we are trying to avoid. The main metadata update is simply correctly computing which indices of the cache correspond to new features and correctly updating another tensor that maps node ids to their index in the cache.

We perform these metadata-related operations, along with the cache replacement described in the previous section, in separate host thread 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 since cache updates are not atomic. A case such as the one illustrated in Figure 3.1 can lead to cache readers reading the wrong feature from the cache if a cache update changes the cache buffer before the reader completes.

A naive approach is to use mutual exclusion, such as with a reader-writer lock, but this can lead to asynchronous updates having equivalent performance to the original synchronous variety. Figure 3.2 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".

Even with a read-preferring or write-preferring reader-writer lock, eventually it must be the case that a cache read waits for a cache update, causing this "latency passing" behavior.

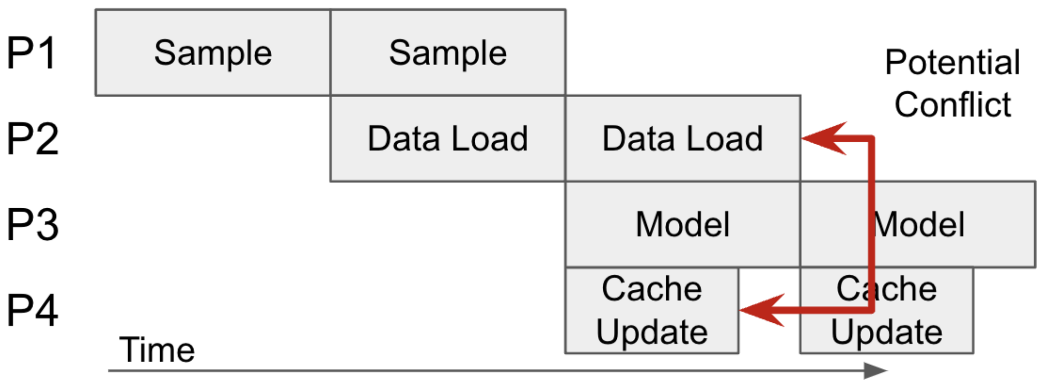


Figure 3.1: Ideal pipelining with potential race condition

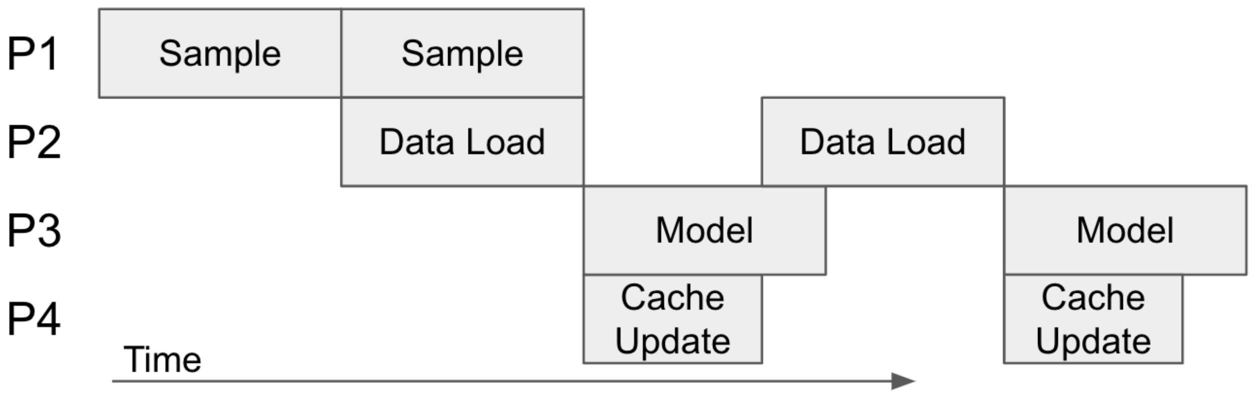


Figure 3.2: Pipeline stall due to enforcement of mutual exclusion between cache update and data loading stages

3.4.1 Masked Updates

Motivated by the key observation that readers should be wait-free but writers can wait, we introduce a novel alternative to mutual exclusion in this context. In our approach, which we call *masking* cache updates, update threads preemptively *mask away* cache entries from cache readers and only perform updates once these cache entries will no longer be used by readers.

Note that we assume only one thread performs a cache update at a time (per GPU), which is enforced by a simple mutex per GPU. If a cache update thread fails to acquire the lock, the update is thrown away.

To perform masked updates, we first add a *mask* tensor to our cache metadata, which indicates for each node in the graph whether it is present in the cache (1 for present, 0 for not).

Then, for each logical thread of execution in the system, we initialize a start and finish atomic integer that just tracks whether the thread is currently performing a cache read. When reading from the cache, threads will first increment their respective start atomic and then check the cache mask and only look in the cache for node ids where the mask indicates it is present. Once the cache read is finished, the finish atomic is incremented. Algorithm 1 summarizes this procedure.

When a cache update needs to occur, the writer will first blind write zeros into the cache mask for any node ids that will be evicted from the cache. Once the blind write has completed, the writer will then capture the value of all start atomics. The writer capturing these values serves as a linearization point, as the writer will wait on the finish atomics until any in progress reads complete. At this point the cache writer is certain that any cache indices that will be replaced are no longer in use by any cache readers. Algorithm 2 summarizes this procedure.

Algorithm 1 Cache Read

```
1: procedure READ($node\_ids$)
2:     $i \leftarrow$ Reader thread id
3:     $S[i] \leftarrow S[i] + 1$
4:     parallel for $node\_id \in ids$ do
5:         if $mask[node\_id] = 1$ then
6:             Do cache read for $node\_id$
7:         end if
8:     end parallel for
9:     $F[i] \leftarrow F[i] + 1$
10: end procedure
```

Algorithm 2 Cache Write

```
1: procedure UPDATE($admit\_nids, evict\_nids$)
2:     $mask[evict\_nids] = 0$
3:     for $reader\_id \in reader\_ids$ do
4:         $s'[reader\_id] \leftarrow s[reader\_id]$
5:     end for
6:     while $\exists id \in reader\_ids : s'[id] < f[id]$ do
7:         Wait
8:     end while
9:     Do cache update
10:    $mask[admit\_nids] = 1$
11: end procedure
```

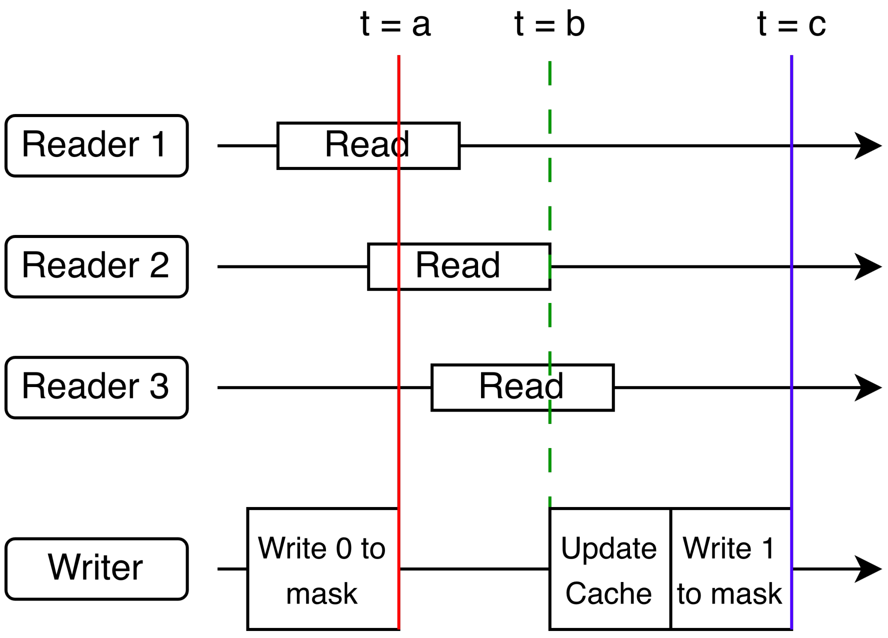


Figure 3.3: Readers and writers with masked, lock-free cache update

Figure 3.3 provides a concrete example of this mechanism in a system with four threads of execution. At time a, the writer thread will capture the start atomics of each of the readers and wait for Reader 1 and Reader 2 to finish. At time b, the writer thread can start the cache update. Note that there is no actual conflict between Reader 3 and the writer thread since Reader 3 will have observed the initial cache mask update of the writer. However, Reader 3 may be subject to some false negatives as a result. Once the write procedure completes at time c completes, the newly updated cache features are now globally visible.

3.5 Multi-GPU Cache Sharing

To maximize usage of available GPU memory, we support sharing of a single logical cache shared among multiple GPUs when connected by NVLink. We use a simple hash partitioning scheme to partition node ids among multiple GPU caches. This means a node's feature can only ever be present in a single GPU's cache.

BGL implements a similar shared cache solution, but serializes all reads and writes to a particular GPU in a single thread to avoid synchronization problems. However, with the lock-free cache update solution described in the previous section, we allow for unlimited concurrent readers for any GPU's cache at a given time. The only restriction is that there can only be a single cache update in progress per GPU at a time.

Enabling lock-free updates is particularly important in the multi-GPU case, as every request must gather node features in a single GPU using peer to peer transfers during the data loading phase. With naive locking, this could lead to blocking on n cache updates, where n is the number of GPUs in the system. However, with the masked cache update approach, these peer to peer transfers will never be blocked.

4 Implementation

We implement our design using Deep Graph Library (DGL) [42], a GNN training framework; PyTorch [36] a tensor operation library; and a mix of Python and C++. Our current implementation consists of roughly 5,000 SLOC of Python and 1,000 SLOC of C++, and is publicly available at `https://github.com/henryliu5/thesis`. We describe some important implementation details of our system.

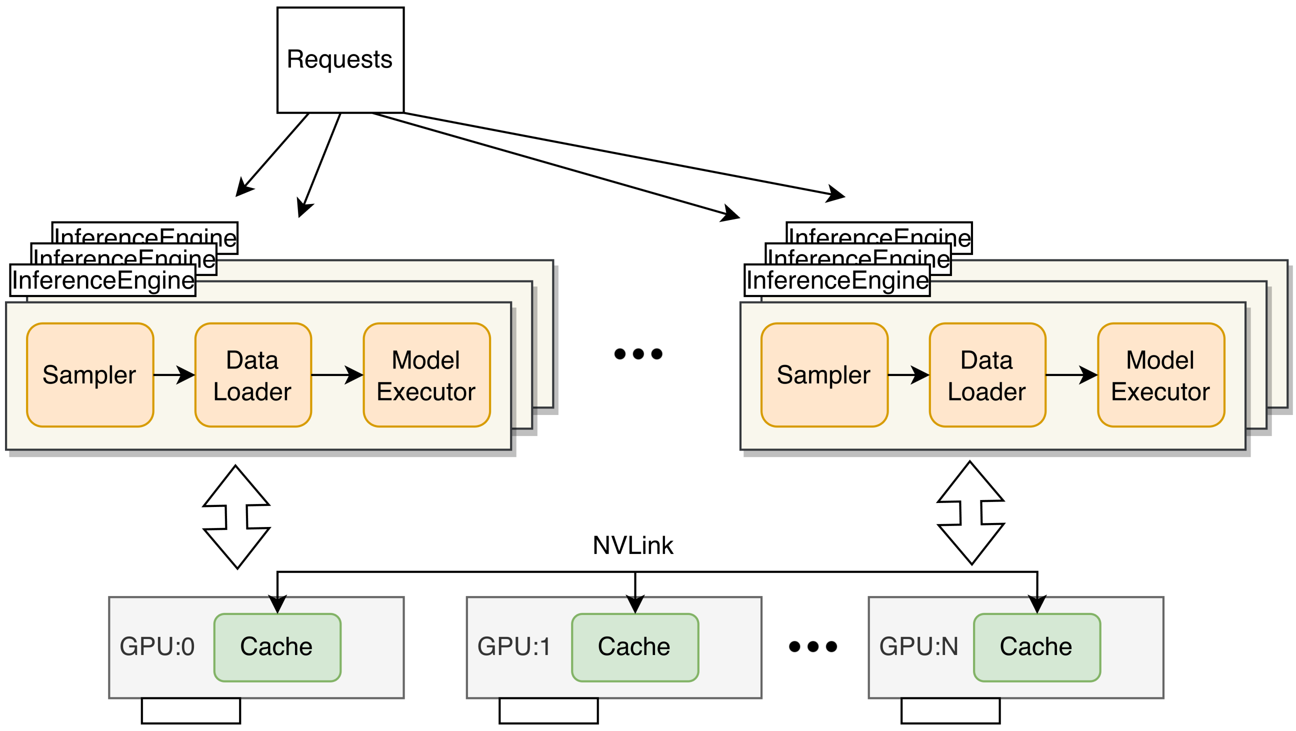


Figure 4.1: System diagram

InferenceEngine Abstraction Due to Python's global interpreter lock, we leverage multiprocessing as the primary vehicle for inter-request concurrency. Our InferenceEngine abstraction (Figure 4.1) represents a single process, which performs all operations for a single inference request - sampling, data loading, and model execution - sequentially. By binding InferenceEngines to a single GPU and containing these steps within a single process, we avoid any overheads due to IPC serialization. This is as opposed to a design where the system is pipelined with many sampler, data loader, and model executor processes that communicate using IPC.

Multiple InferenceEngines can be created per GPU, which share GPU and host memory for resources such as model weights and graph features. All InferenceEngines share a request queue and response queue to receive inference requests and send back results. The InferenceEngine API supports any models or graph datasets that are built using DGL.

Asynchronous Cache Update Thread Control Each InferenceEngine has a dedicated C++ cache update thread linked using pybind11 [20] to perform cache update operations. Atomic integers required for masked cache updates are placed in shared memory using Boost [3]. We also have one shared memory mutex per GPU to allow only one InferenceEngine's update thread to actively write at once. If the update thread fails to immediately acquire the lock, the update is skipped.

Pinned Memory Buffer Reuse Transferring features to GPU memory requires gathering them into a contiguous host memory buffer. Our system has a fixed size buffer in pinned memory that is reused between requests. Transferring data from host pinned memory to GPU memory is significantly faster than transferring from host pageable memory. If a request requires features that are larger than the pinned buffer size, it is resized as needed. The main tradeoff is an increase in pinned memory usage, which can be a problem when there is host memory pressure.

GPU Sampling We use DGL's implementation of GPU sampling to perform the sampling stage. This requires graph structure (no features) to somehow be present in GPU memory. If graph structure can fit in a single GPU's memory, then the entire graph structure is copied to GPU memory during system initialization. If graph structure does not fit in GPU memory, we fall back to DGL's zero-copy functionality that allows GPUs to directly pull graph structure information from host memory when necessary [33].

CUDA Multi-Process Serivce (MPS) [6] To enable concurrent concurrent GPU kernel execution among InferenceEngines, we use NVIDIA CUDA MPS to provide a single CUDA context for all InferenceEngine processes. This is crucial for maximizing GPU utilization and avoiding unnecessary serialization of kernels, as many GPU kernels in our system do not fully occupy all SMs.

4.1 Limitations

Our system currently does not combine new inference requests into the existing graph or retrain the GNN to account for new nodes. 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

We conduct several experiments to understand the latency and throughput of our system under various conditions. Our key takeaways are as follows:

1. Our approach using asynchronous, lock-free cache updates can improve end-to-end inference latency up to 28%. Our frequency-based admission and eviction policy can outperform LFU in terms of cache hit rate, and our asynchronous update mechanism can avoid additional overheads.
2. By splitting a logical cache among GPUs connected by NVLink, our system's throughput can scale linearly with the number of GPUs in the system.
3. Our lock-free approach improves P99 latency over naive locking using a reader-write lock. While we don't see a noticable improvement in peak throughput when using locking vs. lock-free, a microbenchmark in Section 5.3.3 highlights how lock contention worsens as the number of GPUs in the system increase.

5.1 Experimental Setup

5.1.1 Environment

We evaluate our system on a single machine equipped with two 16-core Intel Xeon Silver 4314 CPUs 2.4 GHz, 256 GB main memory, and two NVIDIA 80 GB A100 GPUs connected with NVLinks. We run on Ubuntu 20.04 with kernel version 5.15.0-25-generic, PyTorch 13.1, DGL 1.0, and CUDA 11.7.

5.1.2 Datasets & Workloads

We evaluate our system using four graph datasets, summarized in Table 5.1. Since we could not find any

Dataset	Nodes	Edges	Features	Avg. Degree
reddit [17]	200K	111M	602	492
yelp [47]	700K	13M	500	10
ogbn-products [19]	2.4M	124M	100	51.7
ogbn-papers100M [19]	111M	1.6B	128	14.4

Table 5.1: Graph datasets used in evaluation

publicly available GNN inference traces, we create our own by removing 10% of nodes from each graph to serve as *inference target nodes*. We create requests from these nodes by using their original features and edges.

To emulate different types of inference traces, we generate two varieties:

Uniform sampled requests are generated by uniformly randomly sampling nodes from our pool of inference target nodes. These traces share similarity with mini-batch training and should be fairly challenging for dynamic caches, since there is little locality.

Subgraph biased requests are generated by first partitioning the graph into k partitions, and then cycling through each of the k partitions, making them "hot". For the current "hot" partition, requests are drawn from that partition with a certain probability x. In our experiments we fix $k = 5$ and $x = 0.8$, although these parameters do not need to be fixed. We use DGL's built-in METIS [23] implementation for graph partitioning.

Since our optimizations are orthogonal to model performance, we conduct experiments primarily using a 2-layer GCN architecture. Our system generates full 2-hop neighborhoods when serving requests.

5.1.3 Policies Evaluated

Throughout our evaluation we compare several combinations of cache policies and mechanisms, summarized in Table 5.2. The most important ones are in the first four rows. The Frequency Lock-free approach is our approach as described in Chapter 3, cache candidates computed every 10 requests, cache updates happening in a separate host thread, and cache synchronization handled using our lock-free, mask-based approach. We do not include a no cache baseline since the static cache approach already produces 2-3x speedups in

Policy name	**Overview**	**Dynamic?**	**Cache replacement strategy**	**Sync. Mechanism**
Static	PaGraph static, degree-based cache	No	N/A	N/A
Frequency Prefetch	Prefetching strawman 3.1.1	Yes	Replace cache every k requests	R/W Lock
LFU	LFU	Yes	Evict least frequent in cache	R/W Lock
Frequency Lock-free ⋆	Asynchronous, lock-free updates	Yes	Replace entries via *cache candidates*	Lock-free 3.4
Frequency R/W Lock	Asynchronous updates with R/W Lock	Yes	Replace entries via *cache candidates*	R/W Lock
Frequency Synchronous	Synchronous updates	Yes	Replace entries via *cache candidates*	R/W Lock

⋆ is our approach

Table 5.2: Policy/mechanism combinations evaluated

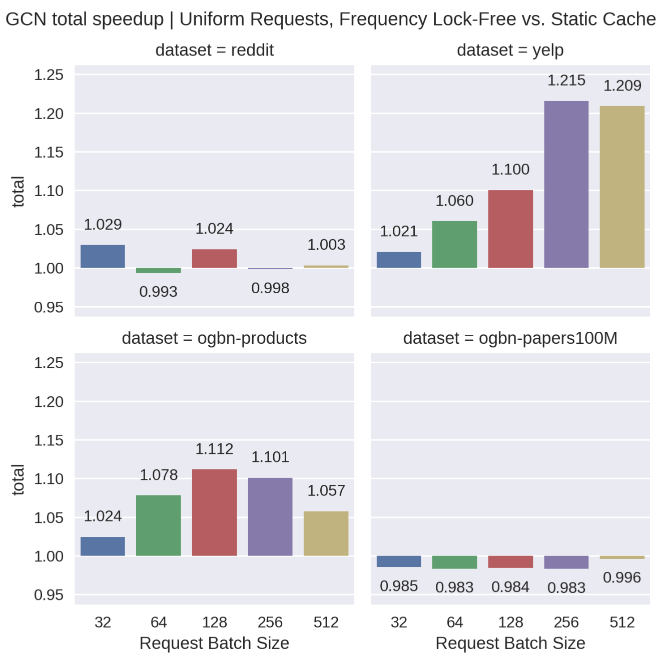
inference latency.

5.2 End-to-End Latency

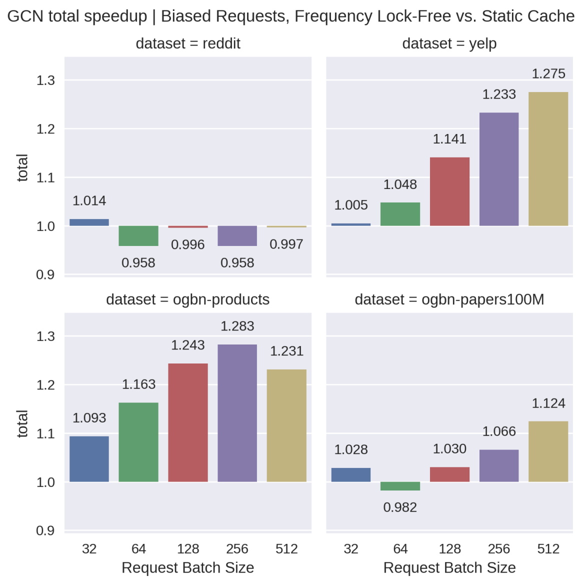
We first examine best-case request-response latencies by using only one GPU and initializing a single InferenceEngine. For each dataset we fix the cache size to 20% of the dataset's total graph feature size.

Figure 5.1 shows the speedup of our approach versus a static, degree-based policy for both uniform and subgraph biased requests. For the yelp and ogbn-products datasets, our approach has speedups ranging from $0-28\%$. Interestingly, since the reddit dataset is incredibly dense, the frequency-based cache update heuristic is not able to achieve better cache hit rates, and this we do not see any speedup for either inference trace.

The ogbn-papers100M dataset is challenging because of its scale and sparsity, along with small feature dimension. Furthermore, the data loading phase was not that large of a proportion of end-to-end latency in



Uniformly sampled requests

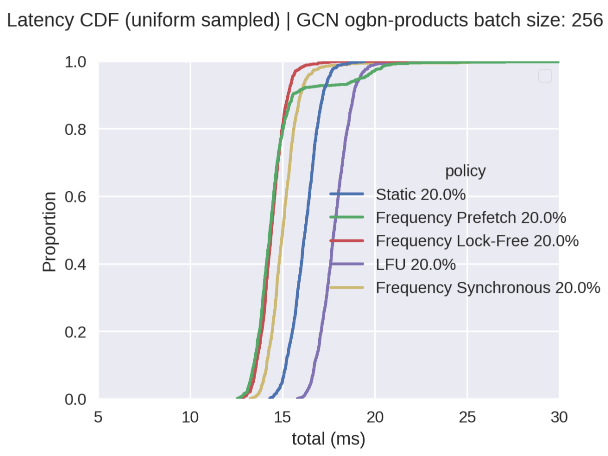


Subgraph biased requests

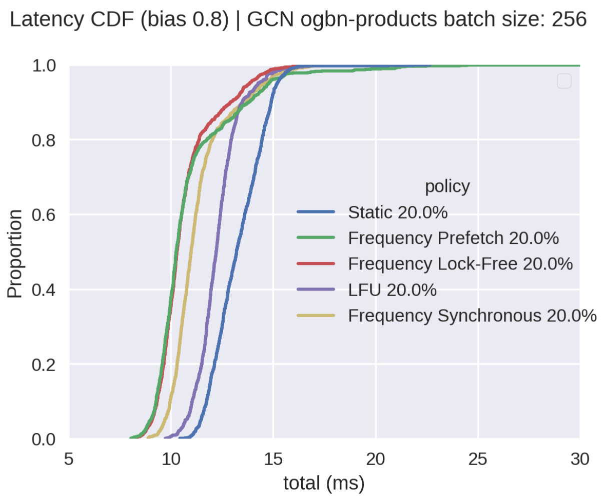
Figure 5.1: End-to-end latency speedups for our lock-free asynchronous approach versus a static cache. Cache sizes are set to 20% of total graph feature size.

the first place, as shown in Figure 2.3. In this case, increasing the request batch size helps improve inference latency with the dynamic cache since the frequency signal is stronger.

To understand these results better, in Figure 5.2 we also examine the CDFs of the same experiment but for all of the policies in Table 5.2 on the ogbn-products dataset with batch size 256. Figure 5.3 shows the cache hit rates over time for the same experiments. We omit the Frequency Synchronous policy in the hit rate figures since it is the same as Frequency Lock-Free.

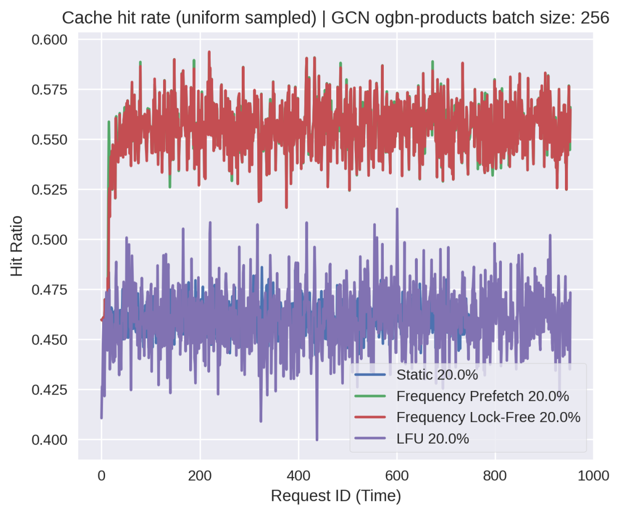


Uniformly sampled requests

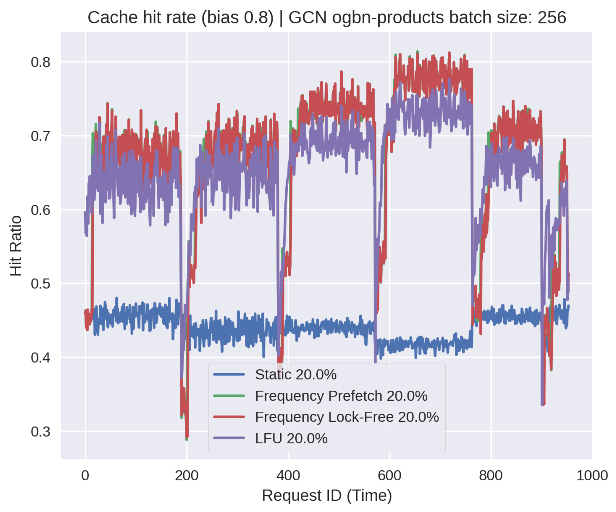


Subgraph biased requests

Figure 5.2: Latency CDFs for ogbn-products batch size 256



Uniformly sampled requests



Subgraph biased requests

Figure 5.3: Cache hit rates for ogbn-products batch size 256

Comparing LFU and the static baseline, we immediately see that LFU struggles in the uniform sampled case since there is little cache hit rate improvement. Thus there is a median slowdown of around 10% due to the overhead of computing LFU updates with no benefit.

Our Frequency Prefetch strawman has the best median latencies overall, since it produces the best cache hit rates and has the same minimal overhead as a static cache. However, in the uniform sampled case the tail latency is apparent, with the 90$^{\text{th}}$ percentile latency increasingly significantly due to cache updates.

The frequency-based *admission* that the frequency-based policies use are crucial, as compared to LFU they offer better cache hit rates in general. While LFU can only match the static baseline cache hit rates in the uniform sampled case, adding the admission policy enables better cache hit rates even in this difficult case.

When examining the Frequency Lock-Free and Frequency Synchronous approaches, they both exhibit relatively little tail latency as a result of the cache candidate mechanism discussed in Section 3.3.1. The Frequency Synchronous performs well relative to the static baseline, but is still 3-5% slower on average, than our Frequency Lock-Free approach. Our Frequency Lock-Free approach is able to match the low-overhead median latencies of the Frequency Prefetch strawman with its asynchronous update mechanism. The time required to perform cache updates somewhat scales with the number of nodes in the graph, and thus the disparity between asynchronous and synchronous updates will increase with larger graphs.

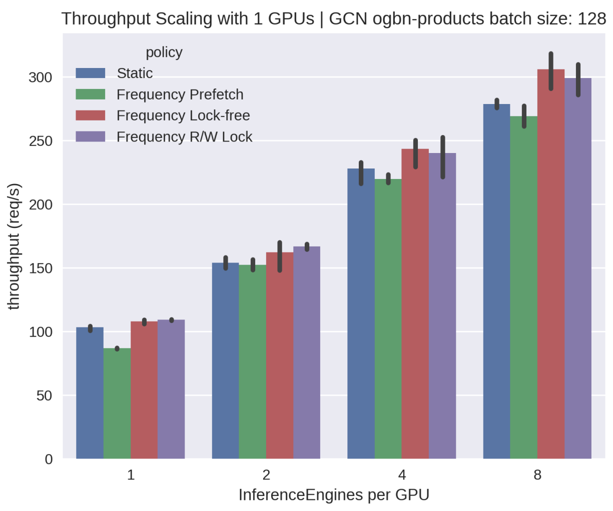
5.3 Locking vs. Lock-free

As described in Section 3.4, we expected that using a reader-writer lock with multiple InferenceEngines and GPUs would pose a problem for both throughput in latency. This is because cache updates would have a blocking effect on serving inference requests. To evaluate this hypothesis, we conduct several experiments comparing the Frequency Lock-free and Frequency R/W Lock policies in this section. We fix the cache size at 20% of total graph features for all experiments.

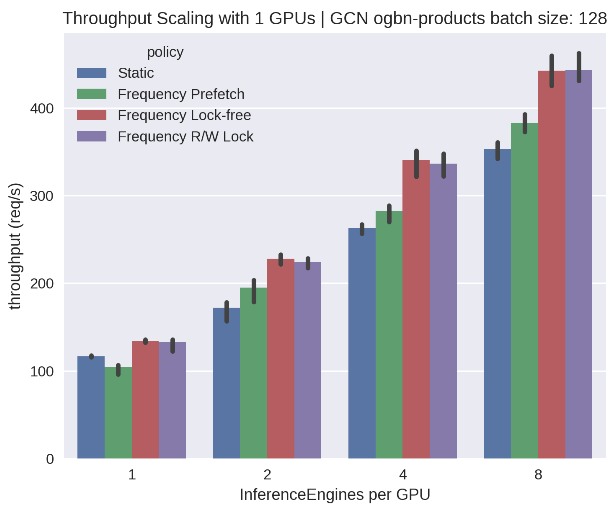
5.3.1 Throughput

We measure the maximum possible throughput of the Static, Frequency Prefetch, Frequency Lock-free, and Frequency R/W Lock approaches. Figure 5.4 shows these results using only 1 GPU and Figure 5.5 shows these results for 2 GPUs. In the 2 GPU case a single logical cache is partitioned between both GPUs.

Comparing these two cases, we observe that doubling the number of GPUs yields roughly linear speedup in terms of peak throughput. We do not have access to a system that has more GPUs all connected using NVLink/NVSwitch and thus do not evaluate scaling to more GPUs. Although we did have a system with four GPUs, only pairs of GPUs on different NUMA nodes were connected with NVLink. In such a situation it is better not to share a single logical cache among all GPUs, but rather have two separate caches for each pair of GPUs. Creating a logical cache between GPUs not connected using NVLink simply incurs more PCIe bus traffic.

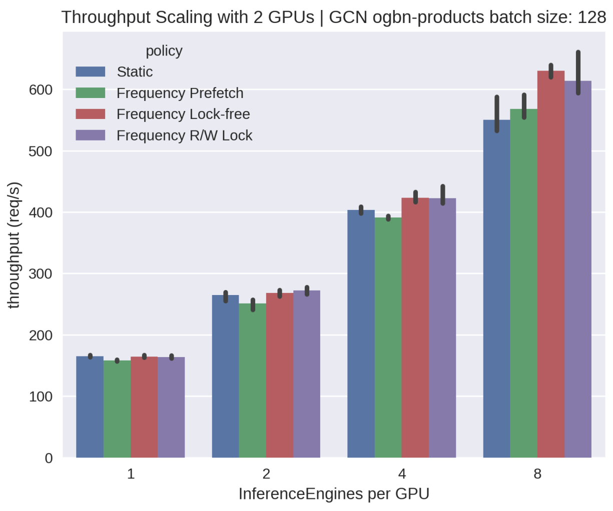


Uniformly sampled requests

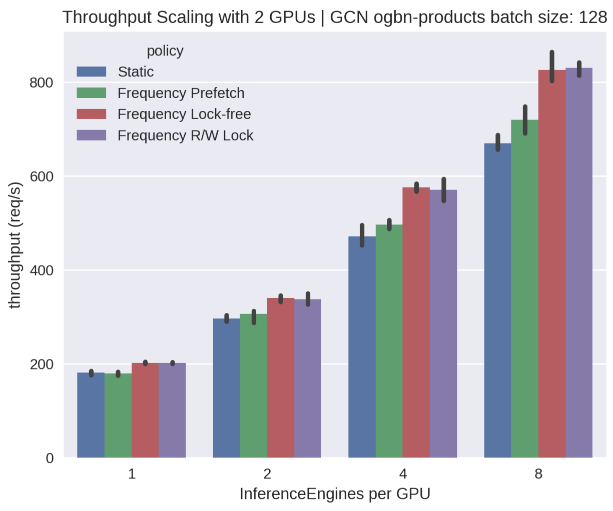


Subgraph biased requests

Figure 5.4: Peak throughput using one GPU and varying number of InferenceEngines per GPU,



Uniformly sampled requests



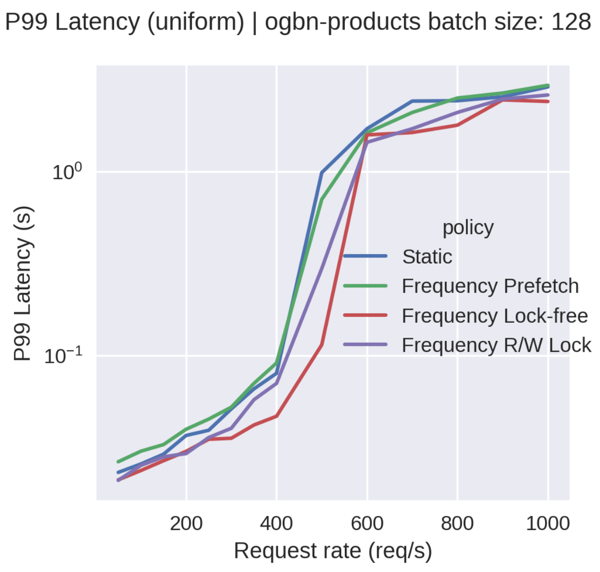
Subgraph biased requests

Figure 5.5: Peak throughput using one GPU and varying number of InferenceEngines per GPU,

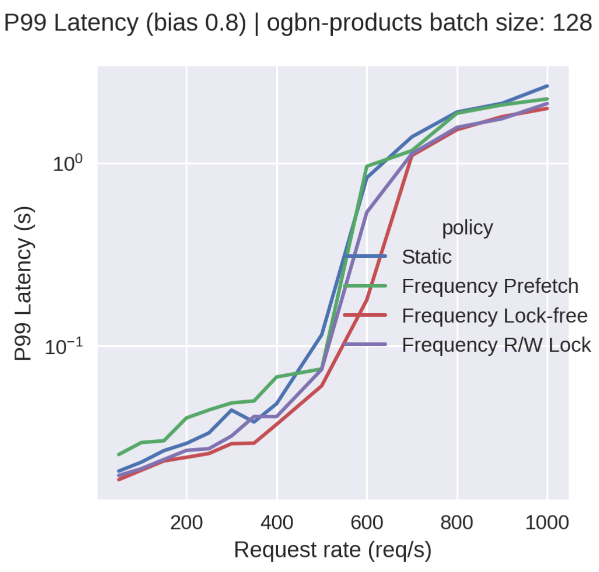
We found that there is actually little discernable difference in peak throughput between the lock-free and R/W lock approaches. Both of these approaches share the cache candidate mechanism, frequency-based admission and eviction, and asynchronous updates, and thus are able to outperform the static and Frequency Prefetch polices. We believe we did not see the throughput difference betwen the lock-free and reader writer lock approaches because of our InferenceEngine architecture. A throughput bottleneck would likely be observed if many data loaders were always trying to read from the cache, but with our architecture this only happens for a brief period of time per process. Thus there is not that much lock contention in the first place.

5.3.2 P99 Latency

Although there is little impact of lock contention on peak throughput, there is an impact on P99 latency. We conducted an experiment where we varied the request rate, with delays between requests drawn from an exponential distribution, and measured the P99 latency for the same policies. Figure 5.6 shows P99 latencies for these policies when the mean request rate varies. In this test we use both GPUs and eight InferenceEngines per GPU. Note the log scale.



Uniformly sampled requests



Subgraph biased requests

Figure 5.6: 99th percentile latencies for varying request rates. Tested on system using both GPUs and eight InferenceEngines per GPU.

We see that using the lock-free approach reduces P99 latency across the board. Lock contention can occasionally produce slow responses, which the lock-free approach avoids.

5.3.3 Lock Conflict Microbenchmark

We also conduct a small microbenchmark to identify the scale of potential lock contention in a larger, fully pipelined system. We remove the sampling and model execution stages from the InferenceEngines and instead send requests that already contain k-hop neighborhoods at maximum throughput. This effectively simulates a fully saturated pipeline where the data loader is fully occupied at all times. We then measure the amount of time spent waiting to acquire read locks for the Frequency R/W lock policy. In this experiment we use a machine that is equivalent in all ways to our original machine but has two additional GPUs. Figure 5.7 illustrates these results. Note the log scale. By increasing the number of GPUs, the number of cache writ-

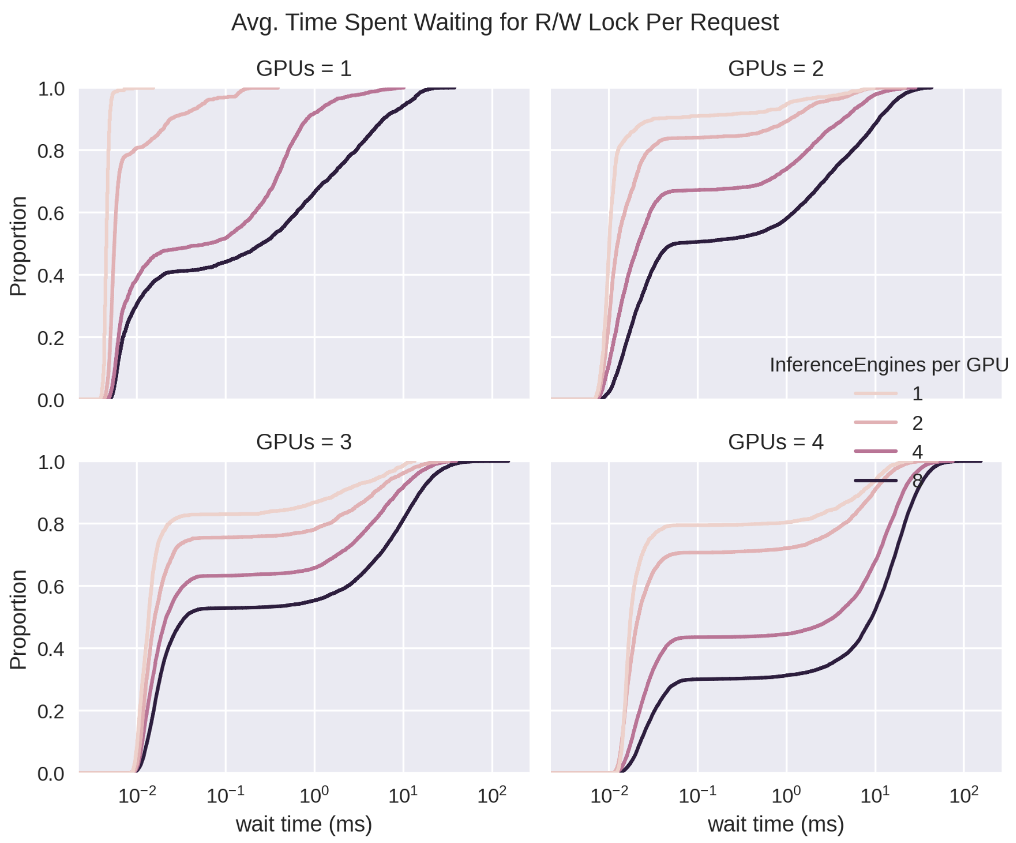


Figure 5.7: Lock contention microbenchmark.

ers (updaters) at a given time increases. Thus, the likelihood of a cache read being blocked by a cache update increases. Naturally, this is also proportional to the number of data loaders (InferenceEngines) present in the system.

Given that the average response times across various datasets is on the order of tens of milliseconds, any wait times greater than a few milliseconds will already become noticeable. In the 4 GPU case with 8 InferenceEngines, we already see that 65% of requests had to wait longer than 1 millisecond. This result points to our lock-free, masked cache approach becoming increasingly important in a large-scale system with many GPUs.

6 Conclusion

In this work we demonstrate how dynamic feature caching can effectively be applied to GNN inference systems. We show that a frequency-based admission and eviction policy can outperform existing solutions, such as caching by vertex degree or using LFU. We then propose appropriate mechanisms to implement this cache policy without impacting latency, by using a *cache candidate* abstraction and performing update operations in a separate host thread. Our system further supports sharing a single logical cache across multiple GPUs, and we propose a lock-free algorithm for performing cache updates to avoid lock contention between cache readers and writers.

6.1 Future Work

There are still several avenues to improve GNN inference by reducing data loading overheads. We list a few here:

1. **Feature compression for PCIe transfers:** To further reduce PCIe bottlenecks, node features can be compressed either offline or on-the-fly, and then copied to GPU memory in a compressed format. Decompression can take place on the GPU, which is extremely efficient [34].

2. **Efficient out-of-core inference using GPUDirect Storage:** While our system assumes that graph structure and features fit in a single host's memory, this may not be the case. Single machine inference has the nice property of avoiding network transfers, and we can still support larger scale graphs by storing graphs on disk. GPUDirect Storage [35] enables direct movement of data from NVMe SSDs to GPU memory, avoiding a redundant bounce in host memory.

3. **Implement more efficient pipelining:** While our InferenceEngine abstraction enables concurrent handling of inference requests by spawning more processes that sequentially perform sampling, data loading, and model execution, a different architecture with finer grain parallelism between these stages may yield higher throughput. A key challenge in implementing this efficiently is managing Python's Global Interpreter Lock and avoiding redundant copies between stages.

6.2 Acknowledgements

I would like to thank my advisor, Dr. Aditya Akella, as well as members of the UTNS lab, including Dr. Daehyeok Kim, Geon-Woo Kim, and Jeongyoon Moon, among many others, for their gracious guidance and assistance with my thesis. I would also like thank Dr. Chris Rossbach and Dr. Devangi Parikh for serving on my honors thesis committee. Finally, I would also like to thank my friends and family, who have always provided me invaluable support throughout the years.

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. Boost. *Boost C++ Libraries*. http://www.boost.org/. Last accessed 2023-04-15. 2023.
4. 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.
5. G. Cormode and S. Muthukrishnan. “An Improved Data Stream Summary: The Count-Min Sketch and Its Applications”. *J. Algorithms* 55:1, 2005, pp. 58–75. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.001. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1016/j.jalgor.2003.12.001.
6. *CUDA MPS*. https://docs.nvidia.com/deploy/mps/index.html.
7. 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.
8. 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.
9. 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].
10. 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.
11. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”. *IEEE/ACM Trans. Netw.* 8:3, 2000. ISSN: 1063-6692. DOI: 10.1109/90.851975. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1109/90.851975.
12. 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.

13. 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.

14. 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.

15. *GitHub, HashiCorp Golang LRU Cache Implementation*. https://github.com/hashicorp/golang-lru.

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

17. 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.

18. 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.

19. W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. "Open Graph Benchmark: Datasets for Machine Learning on Graphs". *CoRR* abs/2005.00687, 2020. arXiv: 2005.00687. URL: https://arxiv.org/abs/2005.00687.

20. W. Jakob, J. Rhinelander, and D. Moldovan. *pybind11 — Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2016.

21. 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.

22. 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.

23. G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". *SIAM J. Sci. Comput.* 20, 1998, pp. 359–392.

24. 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.

25. 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.

26. 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.

27. 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.

28. 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.

29. 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.

30. 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.

31. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

32. 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.

33. S. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W. W. Hwu. "PyTorch-Direct: Enabling GPU Centric Data Access for Very Large Graph Neural Network Training with Irregular Accesses". *CoRR* abs/2101.07956, 2021. arXiv: 2101.07956. URL: https://arxiv.org/abs/2101.07956.

34. *nvcomp*. https://developer.nvidia.com/nvcomp.

35. *NVIDIA GPUDirect Storage*. https://docs.nvidia.com/gpudirect-storage/index.html.

36. 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.

37. 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.

38. 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.

39. 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.

40. 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.

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

42. 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.

43. 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.

44. 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.

45. 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.

46. 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.

47. H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna. “GraphSAINT: Graph Sampling Based Inductive Learning Method”. *CoRR* abs/1907.04931, 2019. arXiv: 1907.04931. URL: http://arxiv.org/abs/1907.04931.

48. M. Zhang and Y. Chen. “Link Prediction Based on Graph Neural Networks”. *CoRR* abs/1802.09691, 2018. arXiv: 1802.09691. URL: http://arxiv.org/abs/1802.09691.

49. 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.