Master’s Thesis of Engineering

Ginex++: Acceleration Techniques for Training Billion-scale Graph Neural Networks on a Single Machine

Ginex++: 단일 머신에서 수십 억 규모의 그래프 신경망
학습을 가속화하기 위한 기법

February 2023

Graduate School of Engineering
Seoul National University
Computer Science and Engineering Major

Sunhong Min

Master's Thesis of Engineering

Ginex++: Acceleration Techniques for Training Billion-scale Graph Neural Networks on a Single Machine

Ginex++: 단일 머신에서 수십 억 규모의 그래프 신경망 학습을 가속화하기 위한 기법

February 2023

Graduate School of Engineering
Seoul National University
Computer Science and Engineering Major

Sunhong Min

Ginex++: Acceleration Techniques for Training Billion-scale Graph Neural Networks on a Single Machine

Advisor Jae W. Lee

Submitting a master's thesis of Engineering

November 2022

Graduate School of Engineering
Seoul National University
Computer Science and Engineering Major

Sunhong Min

Confirming the master's thesis written by
Sunhong Min

January 2023

Abstract

Recently, lots of efforts have been made to seek meaningful inspirations from graph structured datasets using Graph Neural Networks (GNNs). The size of real-world graph datasets grows over time, and there appear cases where the dataset is too big to fit in a single machine's main memory. Lately, several approaches have been made to leverage high-performance storage devices such as NVMe SSDs to scale-up a single machine for GNN training. As opposed to distributed training systems, which scale-out multiple machines for GNN training, disk-based GNN training systems promise equivalent training quality in a more cost-effective manner with an endurable training time increase. Ginex [35] is the state-of-the-art SSD-based GNN training system targeted on billion-scale graph datasets on a single machine. Ginex restructures the conventional GNN training pipeline by separating *sample* and *gather* steps, and thereby realizes a provably optimal cache policy known as *Belady's algorithm*. Although it does a decent job by minimizing the latency arose from *gather*, the newly introduced overhead called *inspect* becomes non-negligible. We apply two acceleration techniques directly purposed to decrease *inspect* overhead on top of Ginex, and name it Ginex++. Two techniques are called neighbor cache compression and *k*-hop approximation for changeset precomputation. By evaluating these techniques on four billion-scale graph datasets, Ginex++ achieves 1.28× higher training throughput on average (1.51× at maximum) than Ginex.

Keyword: Graph Neural Network Training, Neighbor Cache Compression, *k*-hop Approximation

Student Number: 2021-29333

Contents

List of Tables

List of Figures

Chapter 1

Introduction

While conventional application fields of Deep Neural Network (DNN) include images and texts, it has recently extended its application range to graphs as well. Graph Neural Network (GNN), a new class of DNN, is well known as a powerful tool [41, 48, 54] in lots of inference tasks on graph-structured datasets. Examples include node classification [21], recommendation [34, 44], and link prediction [47]. GNN effectively captures plenty of relational information that are embedded in input nodes, and exploits them to retrieve valuable information that are not explicitly exposed.

The primary difference of GNN and conventional DNNs is that nodes (i.e., data samples) in a graph are closely related to each other; data samples are independent in conventional DNNs. Therefore, in order to handle one mini-batch in GNN training, not only the feature vectors of the nodes in the mini-batch but also those of the neighbor nodes of the nodes in the mini-batch are needed [13, 21]. Here comes the two most fundamental steps in the GNN training process: `sample` and `gather`. During `sample` step, nodes themselves in the mini-batch and their neighbor nodes must be retrieved by traversing through the graph dataset. Then during `gather` step, feature vectors of those nodes, that are usually sparsely scattered, must be collected into a contiguous buffer to be prepared for subsequent DNN processes, such as forward and backward passing. These two steps intrinsically require huge number of data accesses.

Existing GNN frameworks [14, 40] thus choose to keep the entire graph dataset in the main memory to avoid excessive overhead from lots of data accesses. For performance grounds, disk-based GNN training has seldom been studied [23]. However, as the size of the real-world graph datasets grows rapidly, the need to scale-up or scale-out the GNN training system has been steadily emphasized. The size may reach hundreds of GBs or even a few TBs, and may exceed the main memory capacity [43, 45]. Several approaches share common ideas of utilizing more than one machine to address this scalability issue of conventional in-memory GNN training [15, 45, 49, 51, 53, 55]. Nonetheless, it is difficult to say that above approaches are the best choices, since they scale the whole hardware components by the same factor, despite the fact that some of them are certainly underutilized.

Ginex [35], as an alternative, takes advantage of high-performance storage devices such as NVMe SSDs to present a more cost-effective method. It effectively reduces the amount of I/O requests, which successfully overcomes serious issues related to NVME SSDs such as low bandwidth compared to DRAM and lack of byte-addressability. Specifically, Ginex realizes the optimal cache policy to create the feature cache in the main memory. This is possible due to the reogranized GNN training pipeline, which consists of `sample` step followed by `gather` step. In traditional GNN training pipeline, `sample` and `gather` are performed in parallel. However, Ginex boldly gives up this parallelism and suggests a new serialized training pipeline. The first phase, which is called `inspect` in Ginex, performs `sample` for multiple batches, or a *superbatch*. The second phase performs `gather` of feature vectors of the previously sampled nodes, by dynamically managing the feature cache according to the optimal policy precomputed during the first phase.

Serialization of above two steps enables the realization of the optimal cache policy. Nonetheless, it is also true that a new overhead is introduced in Ginex due to the solitary execution of `inspect` stage. This stage consists of a parallel execution of superbatch sampling and changeset precomputation, and its time is dependent on the one that takes more time. According to various settings, such as types of datasets, sampling sizes, and batch sizes, `inspect` overhead may increase greatly, threatening the benefit earned from Ginex's original training pipeline. It is therefore a reasonable and necessary attempt to lessen the overhead of `inspect`. Two possible acceleration techniques for `inspect` include neighbor cache compression and k-hop approximation for changeset precomputation. Chapter 3 and

Chapter 4 each elaborates on above techniques in detail, respectively.

We evaluate our new techniques, named Ginex++, compared to Ginex on a server with an 8-core Intel Xeon CPU with 64GB memory and an NVIDIA V100 GPU with 16GB memory. We use four billion-scale graph datasets whose total size ranges from 373GB to 569GB. According to the evaluation results, Ginex++ reduces the training time by 1.28$\times$ on average (1.51$\times$ at maximum) compared to Ginex. The followings summarize our major contributions:

- We profile Ginex according to various sampling sizes, batch sizes, and datasets, and show their impact on `inspect` overhead.
- We introduce two acceleration techniques, neighbor cache compression and *k*-hop approximation for changeset precomputation, to reduce `inspect` overhead in Ginex.
- We implement above techniques on top of Ginex and call it Ginex++, and show the effectiveness of it by evaluating on four billion-scale datasets that do not fit in memory.

Chapter 2

Background

2.1 Graph Neural Networks

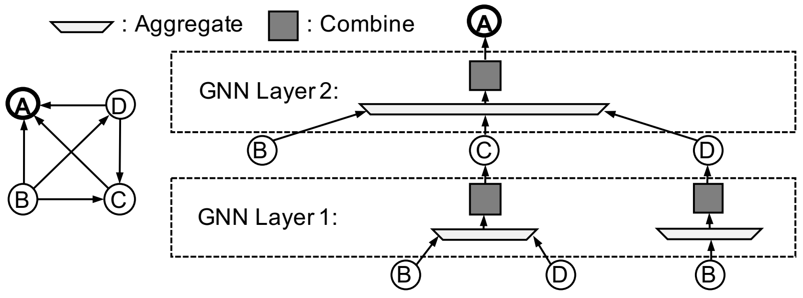


Figure 2.1: 2-layer GNN training on Node A (reproduced from [35])

2.1.1 GNN Training

In a graph-structured dataset, the nodes are connected to each other and these connections are called the edges. Also, each node has its unique feature vector, representing various information about the node. GNNs operate on these graph-structured datasets and their purpose is to create a new embedding for each node. When GNNs operate well, this new embedding wisely captures rich relational information of the node. As a result, these embeddings can be used for various downstream tasks such as node classification and link prediction. On top of retrieving feature vectors of the target node, or the seed node, GNNs must also retrieve vectors from the target node's *k*-hop in-neighbors as inputs. Some node's in-neighbors are nodes that point to that node, while out-neighbors are nodes that are pointed by that node.

Each layer in GNN synthesizes the information of the nodes at each hop. Therefore, it is said that *k*-layer GNN can reflect up to *k*-hop in-neighbors [13, 21].

There are two major steps in each layer of GNN: `Aggregate` and `Combine`. We denote h_v^i as the embedding of node v after the ith layer. The computation is done as follows:

$$h_v^i = \texttt{Combine}(\texttt{Aggregate}(\{h_u^{i-1} \mid u \in N(v)\})) \tag{2.1}$$

$N(v)$ represents the set of neighbor nodes of node v. The features of the incoming nodes are aggregated into a single vector in `Aggregate` step. Lots of functions including mean, max, and sum, can be used for aggregate functions. More complex options are also frequently used in recent works [38]. In `Combine` step, the aggregated feature passes a fully connected layer with a non-linear function. Figure 2.1 shows this whole process with an example of a 2-layer GNN training on Node A.

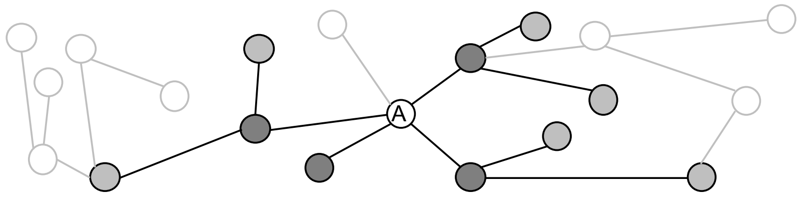


Figure 2.2: Sampling for a 2-layer GNN (sampling size = (4,2), batch size = 1) (reproduced from [35])

2.1.2 Neighborhood Sampling

The major difference of GNN compared to conventional DNN training is that data samples in training dataset are closely connected to each other. As a result, even when the batch size is small, *k*-hop in-neighbors of the nodes in the batch must be also collected, leading to a high training cost. Neighborhood sampling comes for rescue to manage the above-mentioned *neighborhood explosion* problem. The gist is that only a subset of *k*-hop in-neighbors of the seed nodes are sampled, instead of whole of them. GraphSAGE [16], one of the famous works that takes advantage of this approach, randomly samples only a given number of in-neighbors during each aggregation step. Figure 2.2 is an example of a 2-hop (i.e., 2-layer) GNN sampling on Node A. Note that the sampling size is given as (4, 2). It means that the model chooses (at most) four among the neighbor nodes directly connected to the seed node (i.e., Node A) and (at most) two neighbor nodes are chosen for each of the previously chosen nodes. There are lots of variants of the approach, which mainly differ in the design of sampling function such as its sampling size or the method of choosing among multiple candidates [5, 6, 10, 44]. No more than three layers are chosen in practice. Specifically, popular options for the sampling size for GraphSAGE are (25, 10), (10, 10, 10), and (15, 10, 5) [31].

2.2 Disk-based GNN Training

2.2.1 Merits

Real-world graph datasets increase by their size over time, easily exceeding over billions of nodes and tens of billions of edges [43, 45]. The size of a graph dataset is determined by the summation of the size of node feature vectors and the adjacency matrix. It often exceeds the main memory capacity of a single machine, reaching up to several hundreds of GBs or even a few TBs. Scaling-out approach, or distributed training, partitions the dataset into several portions and allocates them to multiple machines in the cluster. While this is indeed a popular solution, it is hard to avoid the criticism that the approach is too cost-intensive [50]. Disk-based GNN training, on the other hand, can wisely exploit high-performance storage devices like NVMe SSDs as memory extensions to provide a much cost-effective solution [23]. NVMe SSDs are known for their plentiful capacity enough to hold the entire dataset, as well as much faster read performance than previously released commodities.

2.2.2 Challenges and Solutions

Still, there exist several challenges when performing GNN training based on SSD. SSD is basically a block device where data is transferred in a 4KB chunk. However, `sample` and `gather` steps in GNN training mostly involve fine-grained random accesses of only tens to hundreds of bytes. Naturally, a naïve usage of SSD on GNN training inevitably results in a huge I/O penalty due to coarse access granularity and low bandwidth (compared to DRAM).

This fact introduces two major challenges. First, OS page cache, which simply keeps recently accessed pages, is an inferior choice for such fine-grained random data accesses. Several approaches instead utilize application-specific in-memory cache as an alternative. PaGraph [28] statically caches features vectors of nodes with the criteria of descending order of out-neighbors' counts. Ginex [35] manages the feature cache dynamically, thereby successfully achieving the optimal cache policy known as Belady's cache replacement algorithm [3]. The algorithm is proven to be optimal by always evicting the data that are not to be needed for the longest time in the future steps.

The second challenge is that the conventional GNN training pipeline is sub-optimal. The two data preparation operations of GNN, i.e., `sample` and `gather`, run in parallel in the conventional GNN training system. However, since they are both I/O intensive operations, the parallel execution results in a resource contention. This is the reason why the time spent on data preparation is even longer than the lengthy one among `sample` and `gather`. Ginex resolutely discards this parallelism and adopts serial execution. In order to lessen the overhead of this new design, Ginex samples multiple batches, or a superbatch, at a time, and designs the optimal feature cache using the information obtained from previously sampled superbatch. Detailed explanation on Ginex's design follows in Section 2.3.

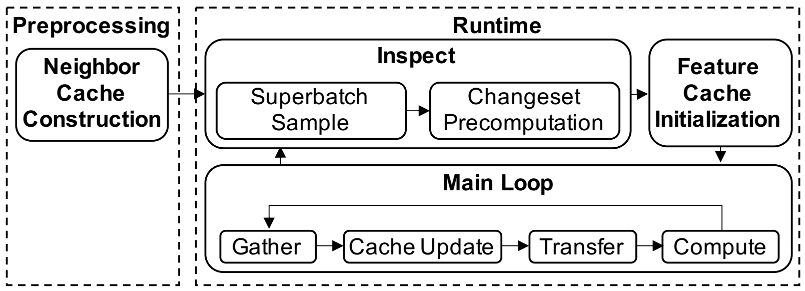


Figure 2.3: Ginex training pipeline overview (reproduced from [35])

2.3 Ginex

2.3.1 Overview

Figure 2.3 illustrates an overview of Ginex training pipeline. First, during the offline preprocessing, neighbor cache is constructed with a predefined size. Then, Ginex enters the runtime and starts training by iterating the following three stages: `inspect`, `feature cache initialization`, and `main loop`. `inspect` stage consists of two steps: `superbatch sample` and `changeset precomputation`. In `superbatch sample`, Ginex samples multiple number of batches, called a *superbatch*, at once. These sampled results are used to set up the feature cache policy for the subsequent `gather` step. Changeset is merely a sequence of cache updates, i.e., what to evict from and insert into the cache for each iteration, and during `changeset precomputation`, changesets for all iterations are computed beforehand. During the step, it also determines which feature vectors to prefetch into the feature cache at initialization, which is in turn used in the next stage called `feature cache initialization`. Finally, in `main loop`, Ginex performs i) `gather` of designated feature vectors, ii) `cache update` for next iteration, iii) `transfer` of gathered feature vectors to GPU, and iv) `compute` on GPU.

2.3.2 Inspector-Executor Execution Model

The core idea of Ginex can be said in short as the application of the inspector-executor execution model [32, 37]. Fundamentally, the inspector procedure runs at the beginning, preparing necessary information for the subsequent executor procedure to properly operate. Ginex adapts the idea of the inspector-executor execution model to enable efficient application-specific in-memory caching for GNN training. To be specific, `inspect` stage corresponds to the *inspector*, and `main loop` stage corresponds to the *executor*. By performing `superbatch sample` beforehand in `inspect` stage, Ginex is able to collect necessary information about the nodes that are to be accessed later in the `gather` step. This separation is the key to meet the optimality of the feature cache.

2.4 Newly Introduced Overhead: Inspect

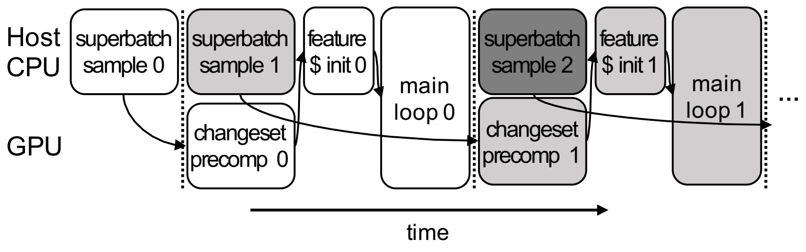


Figure 2.4: Superbatch-level pipeline of Ginex (adopted from [35])

2.4.1 Superbatch-level Pipeline

As shown in Figure 2.3, Ginex reorganizes the conventional GNN training pipeline to realize the optimal feature cache policy. That is, Ginex separates `sample` from `gather` and serializes them, which introduces a new overhead called `inspect` that consists of `superbatch sample` and `changeset precomputation`. `superbatch sample` utilizes the preconstructed neighbor cache, and it runs in parallel with `changeset precomputation`. This superbatch-level pipeline of Ginex is depicted in Figure 2.4. Of course, the runtime stages for the same superbatch should be run in sequence, but they can be pipelined for different superbatches. Thus in Ginex, `changeset precomputation` of the ith superbatch runs in parallel with the next (the $(i+1)$th) `superbatch sample`. Since `changeset precomputation` mainly runs on GPU and `superbatch sample` on CPU, they are suitable candidates for such parallel execution.

2.4.2 Neighbor Cache Construction

As can be seen in Figure 2.3, Ginex constructs a static neighbor cache before the runtime training process begins. Specifically, Ginex creates and saves the neighbor cache in the SSD during the preprocessing procedure. Ginex has a criterion for the *importance* of each node and attempts to put as many in-neighbors of *important nodes* as possible in the neighbor cache. To sort nodes by their importance, Ginex refers to a metric introduced in Aligraph [43] that assesses the trade-off between the cost and the benefit of caching neighbors of each node. The metric, to be specific, is defined as the ratio between the number of out-neighbors and in-neighbors. The grounds are that access frequency (benefit) can be assumed to be proportional to the number of out-neighbors, and that the cache space overhead (cost) can be assumed to be proportional to the number of in-neighbors. The neighbor cache, which is saved in the SSD after creation, is loaded at the beginning of `inspect` stage, providing aid for `superbatch sample`. Note that almost all the memory space excluding the working buffer for sampling processes in `superbatch sample` can be used to hold the neighbor cache.

2.4.3 Superbatch Sample

At the beginning of this step, Ginex loads the preconstructed neighbor cache from the SSD. Next, multiple sub-processes are launched and they perform sampling for as many batches as the superbatch size, S. During the sampling process, information about the target node's neighbor nodes must be collected. In order to collect them, Ginex first looks up the neighbor cache and retrieves necessary information from it. It only retrieves the information from the SSD when the desired information does not exist in the cache. The results of superbatch sampling are saved in the SSD. They consist of two types of data, ids and adj. ids is an 1-D array of the sampled nodes' IDs. adj contains the connectivity information about the sampled nodes. These two types of data are saved in the SSD as separate files annotated with the corresponding batch index. That is, in total $2 \times S$ files ($ids\_0, ids\_1, ...$ $, ids\_(S-1), adj\_0, adj\_1, ... , adj\_(S-1)$) are saved. The size of each file varies according to the characteristics of the dataset, the sampling size, and the batch size, but typically ranges from several hundred KBs to just a few MBs.

2.4.4 Changeset Precomputation

Changeset is merely a set of cache update information about which feature vectors to insert into and which to evict from the feature cache. This can be computed every time when `gather` is performed, but Ginex chooses to precompute all the changesets beforehand. This can be done by utilizing the previously saved ids files. The merit of the precomputation is that the computation can be speedily done in batch on GPU. Each ids file is first loaded on the CPU memory, and is then streamed to GPU when needed. After necessary computations done on GPU, the results of the changeset precomputation are sent back to CPU and are stored in the SSD. The reason for such streaming process is because the total size of ids files may exceed the GPU memory capacity. Along with changesets, the information (i.e., which features to keep) about the initial cache state is also derived at this step. As a result, ($S + 1$) files are created and saved in the SSD, namely one file for the cache initialization ($init$) and S files for the cache update information for every iterations ($update\_0, update\_1, ...$ $, update\_(S - 1)$).

2.4.5 Inspect Overhead

The portion of `inspect` time out of the entire training time may vary according to various settings: types of datasets, sampling sizes, batch sizes, and so forth. Since we seek to reduce the time spent on `inspect` stage, there is more room for improvement when the `inspect`

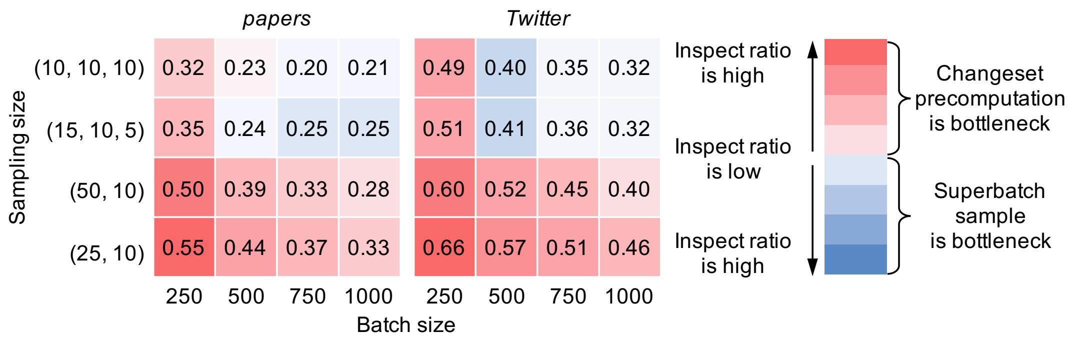


Figure 2.5: Inspect ratio over training time for varying sampling sizes and batch sizes

ratio is high. We profile the `inspect` ratio over training time for sampling sizes of (10, 10, 10), (15, 10, 5), (50, 10), and (25, 10), and for batch sizes of 250, 500, 750, 1000. We use two billion-scale datasets, extended from the existing datasets, *ogbn-papers100M* [17] (*papers*) and *twitter-2010* [27] (*Twitter*). Details on how the extension is made are described in Section 5.1.

First of all, we can observe that the `inspect` ratio gets higher as the batch size gets smaller. In Figure 2.5, within each row, it tends to show thicker color as it goes from right to left. Generally, smaller batch sizes take more time for training than larger batch sizes, but there is a lower chance of overfitting than larger batch sizes [18]. Therefore users may choose adequate batch size according to their needs. In many works [28,39,55], various batch sizes of few tens to few thousands are used for GNN training. Since there exist computation and memory requirements barriers when increasing the batch size in GNN training [11], considerations on small batch sizes must also be made.

Secondly, it can be seen that the `inspect` ratio becomes higher as the total product of the sampling size becomes smaller. By multiplying the size in each layer of the sampling size, we can estimate the approximate overhead of the sampling size. For example, since the sampling size of (10, 10, 10) means that 3-layers are used and at most ten nodes are sampled in each layer, we can conjecture that at most $10\times10\times10$=1000 nodes will be sampled per each target node in such setting. Various sampling sizes such as (10, 10, 10) and (15, 10,

5) are used in many GNN works [31, 35], and especially, the sampling size of (25, 10) is recommended for GraphSAGE [16]. Likewise, it is important to cover various sampling sizes in GNN training.

Lastly, according to various settings, either `changeset precomputation` or `superbatch sample` can be the bottleneck. Specifically, there is a tendency that `superbatch sample` becomes the bottleneck when the total product of the sampling size and the batch size gets bigger. Conversely, `changeset precomputation` tends to become the bottleneck when the total product of the sampling size and the batch size gets smaller. It is obvious that GNN training can be done on various settings, as users' needs may vary. Therefore, it is necessary to aim to reduce both `changeset precomputation` time and `superbatch sample` time in order to effectively reduce the `inspect` overhead of the system.

Chapter 3

Neighbor Cache Compression

3.1 Analysis on Ginex Neighbor Cache

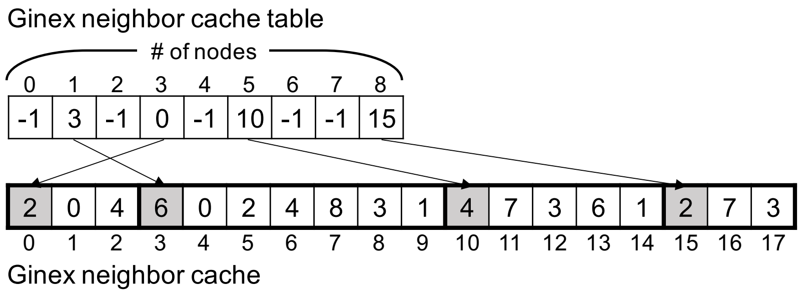


Figure 3.1: Ginex neighbor cache structure

Figure 3.1 shows the structure of Ginex's neighbor cache table and neighbor cache, and how they interact. The two major roles of the cache table and the cache are to i) tell if the given node's neighbors are present in the cache or not, and to ii) correctly return the list of the given node's neighbor nodes if they exist in the cache. To do so, Ginex uses direct addressing method. The neighbor cache table is an 1-D array having as many elements as the number of nodes. Each element of the table represents the index to refer to in the cache if its value is larger than -1. If its value is equal to -1, it indicates that the corresponding node's neighbor nodes are not present in the cache, and must be retrieved from the SSD. In Figure 3.1, the value at the 0th, 2nd, 4th, 6th, and 7th element of the cache table is -1. This indicates that the neighbor nodes of Node 0, 2, 4, 6, and 7 are not retrievable from the cache and must be retrieved from the SSD. This successfully accomplishes the first role.

For the cache table element whose value is larger than -1, it is treated as the index to refer to within the cache. The elements of the cache pointed by the cache table involve the number of neighbors for the corresponding node, and the IDs of the neighbors listed from the next element. For example, in Figure 3.1, the value at the 5th element of the cache table is 10, which means that the information about the neighbor nodes of Node 5 starts from the 10th element of the cache. Since the value of the 10th index of the neighbor cache is 4, it means that the number of neighbors of Node 5 is four. Therefore, we can assume that the actual IDs of the neighbors are listed in the next four consecutive entries, i.e., from the 11th index to the 14th index of the cache. This is how Ginex performs its second role.

It is obvious that when there is a cache miss, the system suffers from a massive I/O penalty by traveling down to the SSD. The cost of such cache miss is severe, so it is effective to pack as many information about the neighbor nodes as possible in the cache. We profile Ginex's neighbor cache hit ratio on four billion-scale datasets, extended from the existing datasets, *ogbn-papers100M* [17] (*papers*), *ogbn-products* [17] (*products*), *com-friendster* [27] (*Friendster*), and *twitter-2010* [27] (*Twitter*). Detailed explanations on the datasets are presented in Section 5.1. As a result, Ginex's neighbor cache hit ratio records 0.73, 0.43, 0.80, and 0.89 on *papers*, *products*, *Friendster*, and *Twitter* datasets, respectively. It can be seen that especially for particularly dense (i.e., having relatively lots of edges per node) datasets like *products*, the neighbor cache hit ratio is quite low. One simple yet powerful way to pack as many information in the limited size of the cache is to compress its contents. We therefore explore several candidates for the compression scheme.

3.2 Compression Schemes

3.2.1 FastPFOR

FastPFOR [25] is a famous C++ library for integer compression schemes. The library finds chances to utilize SIMD instructions whenever possible. Also, it can decompress at a rate of 15GB/s, which is considerably faster than generic compression schemes. However, FastPFOR's application is limited to the compression of arrays of 32-bit integers, and is specially designed for the case where most integers are small. This is not the case for Ginex, where the elements of the neighbor cache table and the neighbor cache are 64-bit integers whose values may be arbitrarily big.

First of all, it is essential to use 64-bit integers for the elements in the neighbor cache table. The maximum possible value of the elements of the cache table is the number of entries of the cache (minus one to be precise). Since we set the size of the neighbor cache as big as several tens of GBs to handle billion-scale datasets, the number of entries of the cache easily exceeds the maximum value that can be represented by a 32-bit integer (i.e., 2^{31}-1).

Secondly, the maximum possible value of the elements of the cache is the larger one among the two: the number of the nodes, and the largest of the number of neighbors of each node. In practical settings, it is nearly impossible for the latter to exceed the maximum value that can be represented by a 32-bit integer. The problem is the former, since we target billion-scale datasets whose total size may easily exceed hundreds of GBs or even a few TBs. Although the largest number of nodes among the datasets we use to evaluate is about 444.24M, it is unsafe to claim that this is always the case. When the size of the dataset gets bigger, the number of the nodes may exceed the maximum value that can be represented by a 32-bit integer. For these reasons, we conclude that FastPFOR is not appropriate for our case.

3.2.2 LZ4

LZ4 [7] is a generic compression algorithm that is scalable with multi-core CPU. It is known for its extremely fast decompression speed of multiple GB/s per core. Specifically, according to the benchmark test introduced in [9], LZ4 (v1.9.3) records a compression ratio of 2.10, a compression speed of 740MB/s, and a decompression speed of 4500MB/s. At first, we regard this scheme appropriate for our case since two important features were high compression ratio and fast decompression speed. The compression speed is not critical for our case as the neighbor cache construction is done before the runtime training process begins. However, when we apply this scheme on *products* dataset, the compression ratio on the neighbor cache records 1.30 (the neighbor cache of 59GB compresses to 45GB). This discrepancy in compression ratio stems from the fact that the scheme is a generic approach, which does not take into consideration that the target bytes are actually a sequence of integers. The solo superbatch sample time records 1.11× improvement after the compression scheme is applied. We regard such speed-up quite deficient, and seek for other compression schemes with higher compression ratio.

3.2.3 ZSTD

ZSTD [9], a short for Zstandard, is a fast lossless compression algorithm created by Meta (former Facebook). It is known to be supported by a fast entropy stage introduced in Huff0 and FSE library [8]. One outstanding aspect of the scheme is a notably high compression ratio. According to the benchmark test introduced in [9], ZSTD (v1.5.1) records a compression ratio of 2.89, a compression speed of 530MB/s, and a decompression speed of 1700MB/s. Note that the test was done on the same benchmark of LZ4 (v1.9.3) introduced above. Compared to LZ4, its compression speed and decompression speed are bit slower, but the compression ratio is much better. We consider ZSTD as a decent candidate for our case for its high compression ratio. The decompression speed is inferior than that of LZ4, but we plan to decompress multiple byte-blocks in parallel by taking advantage of multi-processing. As a result, when we apply the scheme on the same *products* dataset, the compression ratio on the neighbor cache records 1.92 (the neighbor cache of 86GB compresses to 45GB). As in LZ4, it shows an inferior compression ratio than that of the benchmark, since ZSTD is also not specifically targeted for compressing integers. Yet, we believe that ZSTD approach is more competitive than other aforementioned approaches. The superbatch sample time alone records 1.65× improvement after the compression scheme is applied.

3.3 New Neighbor Cache Structure

Figure 3.2 shows Ginex++'s neighbor cache table and the compressed neighbor cache, and how they successfully deliver the two major roles. As a reminder, the two roles of the cache

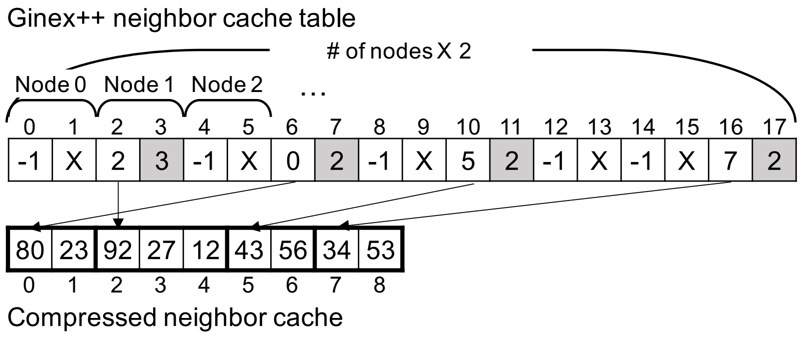


Figure 3.2: Ginex++ neighbor cache structure

table and the cache are to i) notify if the target node's neighbors exist in the cache or not, and to ii) precisely return the list of the target node's neighbor nodes if they are present in the cache. We determine to apply the compression scheme per single list of neighbor nodes in the neighbor cache. Ginex's neighbor cache is nothing but a sequence of information about corresponding nodes' neighbor nodes. The information consists of the number of the neighbor nodes, and the actual IDs of the neighbor nodes. We apply the compression scheme for each sequence of information. After applying the scheme multiple times, we simply concatenate the compressed results into an 1-D array. We call this concatenation of the compressed results the compressed neighbor cache.

We propose a novel neighbor cache table structure, whose size doubled compared to Ginex's cache table. In Figure 3.2, there are two entries per each node in the cache table. The first entry among them indicates the index to refer to in the cache if its value is larger than -1. If its value is -1, it means that the target node's neighbor nodes are not present in the cache, and should be found from the SSD. The role of the first entry is same as Ginex's neighbor cache table. The second entry, which is newly added, indicates the compressed size of the corresponding node's neighbor nodes within the compressed neighbor cache. If the target node's neighbor nodes are not present in the cache, this newly added entry may contain any value, i.e., a *don't care* value. For example, Node 1's neighbor nodes' information starts from the 2nd index within the compressed neighbor cache, and the length of the compressed

```
1  # Below for loop runs on multi−process using num_workers of workers in PyTorch DataLoader
2  n_id_to_neighbor_nodes = {}
3  for n_id in batch:
4     start_offset = neighbor_cache_tbl[n_id * 2]
5     if start_offset != −1: # indicates cache hit
6        compressed_size_in_bytes = neighbor_cache_tbl[n_id * 2 + 1]
7        decompressed_neighbor_nodes = ZSTD_decompress(neighbor_cache + start_offset, compressed_size_in_bytes)
8        n_id_to_neighbor_nodes[n_id] = decompressed_neighbor_nodes
9     else: # indicates cache miss
10       neighbor_nodes = Read_from_SSD(n_id)
11       n_id_to_neighbor_nodes[n_id] = neighbor_nodes
```

Figure 3.3: Pseudocode for retrieving neighbor nodes using compressed neighbor cache

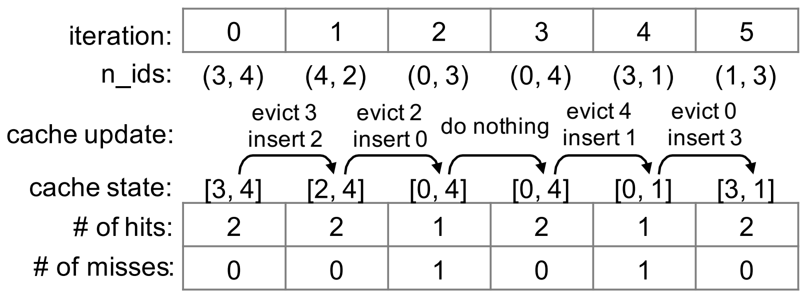
information is 3-bytes. We can know that Node 2's neighbor nodes' information is not present in the cache, since the value of the cache table at the 4th index is -1. In this case, the next element (i.e., the value at the 5th index) may contain any value. This successfully delivers the first role.

To discuss how Ginex++ fulfills the second role, please refer to Figure 3.3. It shows how the information about the target node's neighbor nodes are correctly retrieved. For example, for Node 5 in Figure 3.2, since the `start_offset` is 5, we know that there is a cache hit. Then, we retrieve the compressed size of the information of the neighbor nodes by reading the next entry in the cache table, which is 2 (at the 11th index). Therefore, we read two consecutive bytes beginning from the 5th index of the compressed neighbor cache, i.e., the 5th and 6th index. Then, as presented in line 7 of Figure 3.3, decompression method of the chosen scheme is used to return a list of the neighbor nodes. Note that the previously earned compressed size is used as an argument. This is how Ginex++ performs the second role.

Chapter 4

k-hop Approximation for Changeset Precomputation

4.1 Analysis on Ginex Changeset Precomputation



iteration:	0	1	2	3	4	5
n_ids:	(3, 4)	(4, 2)	(0, 3)	(0, 4)	(3, 1)	(1, 3)
cache update:	evict 3 insert 2	evict 2 insert 0	do nothing	evict 4 insert 1	evict 0 insert 3	
cache state:	[3, 4]	[2, 4]	[0, 4]	[0, 4]	[0, 1]	[3, 1]
# of hits:	2	2	1	2	1	2
# of misses:	0	0	1	0	1	0

Figure 4.1: Ginex changeset precomputation

Along with superbatch sample, changeset precomputation is also an important step that takes part in inspect stage. The purpose of this step is to calculate the cache update information for future iterations, and save them in the SSD. Ginex accomplishes the optimal cache policy when calculating the changeset, since it knows which node IDs, or `n_ids`, are accessed in which iterations in advance. As can be seen in Figure 2.4, such information is available in advance, since Ginex constructs the training pipeline to guarantee that the ith superbatch sample is completed before the ith changeset precomputation begins. Therefore, at the point of changeset precomputation, information about the previously sampled nodes for S batches is ready.

Figure 4.1 shows the inputs and the results of Ginex's changeset precomputation. The inputs are given as `n_ids` for each iteration. The results are the cache update information and the initial cache state information. Detailed algorithm about how Ginex successfully outputs such optimal cache update information is introduced in the original paper [35]. Also, our new approach does not alter the original algorithm, so we don't spare space for the explanation on it.

In Figure 4.1, we assume that the superbatch size, or S, is equal to 6, and at most one entry can be evicted from and inserted into the cache at each iteration. Also, we assume that the cache has two entries. Ginex successfully delivers the provably optimal Belady's algorithm [3] by evicting the data that will not be used for the longest distance, and filling them with the data that are accessed in earlier iterations within the current superbatch. As a result, Ginex records ten cache hits and two cache misses in the example, which is a cache hit ratio of 0.83. Ginex computes the changesets by simulating the cache state (i.e., which node IDs are present in the cache) of every iteration; it compares the difference between the two consecutive cache states. This guarantees that the optimal cache policy is met, but involves quite a lot of computations. Although Ginex successfully solves this problem in $O(S)$ complexity using its unique approach, it still requires three passes over the S access traces. This computation overhead could be severe when S becomes bigger.

4.2 Detailed Explanation on the Approach

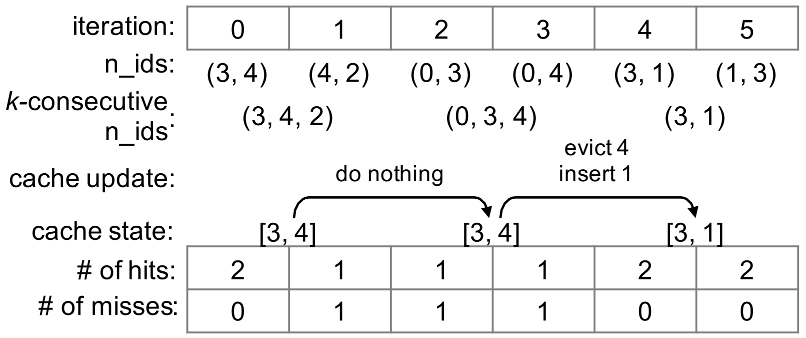

iteration: 0 1 2 3 4 5
n_ids: (3, 4) (4, 2) (0, 3) (0, 4) (3, 1) (1, 3)
k-consecutive n_ids: (3, 4, 2) (0, 3, 4) (3, 1)
cache update: do nothing; evict 4 insert 1
cache state: [3, 4] [3, 4] [3, 1]
# of hits: 2 1 1 1 2 2
# of misses: 0 1 1 1 0 0

Figure 4.2: Ginex++ changeset precomputation (when *k* = 2)

As mentioned above, Ginex realizes the optimal cache policy by comparing the two consecutive cache states. This approach guarantees to realize the optimal cache policy, but it takes lots of time for the computation. We thus propose to approximate the calculation of the changeset to alleviate the computation overhead. To be specific, we combine *k* consecutive `n_ids`s, and use them to simulate the cache state instead. As we combine more `n_ids`s (i.e., increase the value of *k*), it means that the cache update information will be calculated more sparsely, thereby gradually reducing the time needed for the calculation. However, the quality of the cache will be negatively affected more as *k* increases, so there is a trade-off between the two. We seek to come up with an adequate *k*, that effectively reduces the time spent on changeset precomputation, while not much harming the feature cache quality. We denote this approach by *k*-hop approximation for changeset precomputation.

Figure 4.2 shows an example of how Ginex++ works with *k*=2, when receiving the same `n_ids`s as inputs as in Figure 4.1. First, *k* consecutive `n_ids`s are combined to create $\lceil S/k \rceil$ new `n_ids`s. Then, we simply treat these new `n_ids`s as if they are original `n_ids`s, and perform further steps identically. Specifically, we use Ginex's original algorithm to return the cache update information in three passes of the access traces. Now, however, the length of the access traces becomes $\lceil S/k \rceil$ instead of S, so the calculation overhead is approximately reduced by $1/k$. Ginex++ records nine cache hits and three cache misses, which is a cache hit ratio of 0.75. Previously when Ginex calculates the optimal cache update information in Figure 4.1, the cache hit ratio records 0.83. Likewise, by setting *k* bigger than 1, there may be some negative impact on the cache quality. Note that Ginex's original approach is identical to Ginex++ with *k* equal to 1. The negative impact on the feature cache may lead to some delay in the future `gather` step, which uses the feature cache to retrieve feature vectors of the demanded nodes. In order to take full advantage of the new approach, we must attempt to i) increase *k* to reduce the time spent for the computation and at the same time, ii) keep *k* small enough to promise decent feature cache quality.

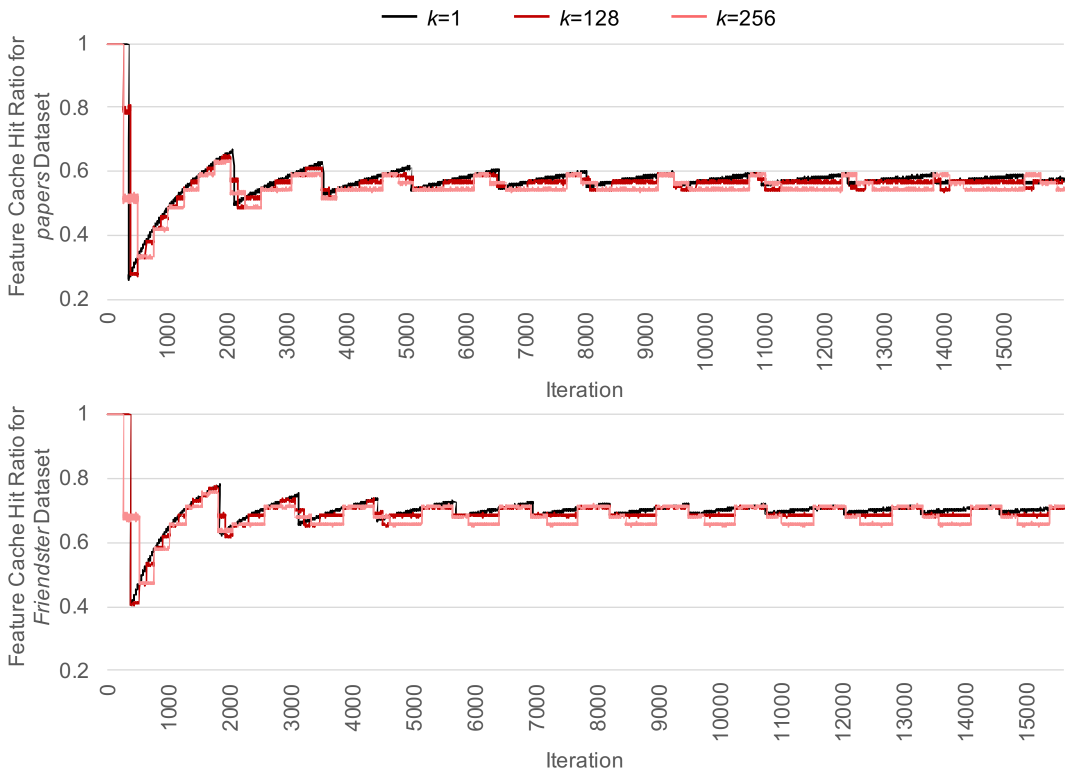


Figure 4.3: Feature cache hit ratios of the approach with various *k*s

4.3 Impact on Quality of Feature Cache

We measure the feature cache hit ratio on *papers* and *Friendster* dataset with various *k*s. Figure 4.3 shows the feature cache hit ratio over iterations for various *k*s. For the clearness of the figure, we only depict the results of *k* with value of 1, 128, and 256. When *k* is set to 1, the feature cache is updated every iteration, following the optimal cache policy. Thus, its feature cache hit ratio can be regarded as the highest possible hit ratio reachable. As *k* gets bigger, the interval between each cache update is also elongated. Therefore, the cache quality becomes sub-optimal to some degree. As can be seen in Figure 4.3, as *k* grows, the feature cache hit ratio expresses a stair-like tendency. Its update cycle is prolonged, and the cache shows slightly more cache misses during the prolonged periods.

However, the absolute cache hit ratio degradation itself is not that severe, as can be seen in the figure; even for the result when *k* is as big as 256, the hit ratio curve is not that far away from that of when *k* is equal to 1. Rather, some other practical conditions such as memory capacity are more crucial than the cache hit ratio when determining the adequate value of *k*. As *k* increases, Ginex++ must make unions of more `n_ids`s, and keep the unioned results on GPU memory to perform changeset precomputation. Also, Ginex++ must keep *k* consecutive batch inputs in CPU memory for each cache update (which happens every *k* iterations) to be made. Therefore, it is impossible to keep increasing *k* due to such GPU and CPU memory capacity limitations. In addition, keeping many batch inputs in CPU memory can result in excessive use of swap space, which causes severe and negative impact on the performance. Hence, in order to take all the above into considerations, running a short profiling for one small superbatch for various *k*s may be helpful. Details on how an adequate *k* can be selected are discussed in Section 5.2.2.

Chapter 5

Evaluation

5.1 Methodology

Table 5.1: System configurations

CPU	Intel Xeon Gold 6244 CPU 8-core @ 3.60 GHz
GPU	NVIDIA Tesla V100 16GB PCIe
Memory	Samsung DDR4-2666 64GB (32GB $\times$ 2)
Storage	Samsung PM1725b 8TB PCIe Gen3 8-lane
S/W	Ubuntu 18.04.5 & CUDA 11.4 & Python 3.6.9 & PyTorch 1.9

5.1.1 System Configurations

Table 5.1 summarizes the system configurations. We evaluate Ginex and the new techniques on a Gigabyte R281-3C2 server with an 8-core CPU (16 logical cores with hyper-threading), an NVIDIA V100 GPU, and a Samsung PM1725B NVMe SSD.

5.1.2 Model and Dataset

We adopt a widely used 2-layer GraphSAGE [16] for the evaluation. Specifically, the sampling size is set to (25,10) and the model has a hidden dimension of 256. The batch size is by default set to 1000. We extend the following four real-world datasets for the evaluation: *ogbn-papers100M* (*papers*) [17], *ogbn-products* (*products*) [17], *com-friendster* (*Friendster*) [27], and *twitter-2010* (*Twitter*) [27]. We follow the methodology in [23] to extend the datasets. To be specific, we use a graph expansion technique [4] which applies Kronecker graph

Table 5.2: Graph datasets

	Original		Extended		
Dataset	nodes	edges	nodes	edges	size
ogbn-papers100M	111.06M	1.62B	444.24M	14.24B	569GB
ogbn-products	2.45M	61.86M	220.41M	20.24B	388GB
com-friendster	65.61M	1.81B	262.43M	15.48B	393GB
twitter-2010	41.65M	1.47B	249.91M	14.63B	373GB

theory [26] to preserve the intrinsic characteristics of the original graph. Table 5.2 shows the details about the original and the extended datasets. For the training set, we randomly choose 10% of the nodes. The working set, however, may involve the entire feature vectors since in GNN training, the feature vectors of not only the target nodes in the training set but also those of their k-hop neighbors are needed. The feature dimension of all datasets is set to 256 by default. Also, the entire datasets are stored in the SSD during the training, except the pointer array in a CSC-formatted adjacency matrix, which is kept in the main memory. The size of the pointer array is only about a few GBs and at the same time is very frequently accessed, so we choose to follow the common design.

5.1.3 Comparison Baseline

We choose Ginex [35] as our baseline. Ginex is compared to Ginex+NCC (Ginex+Neighbor Cache Compression), Ginex+k-hop (Ginex+k-hop Approximation for Changeset Precomputation), and Ginex++. Ginex+NCC is Ginex equipped with the neighbor cache compression technique described in Chapter 3. Ginex+k-hop represents Ginex equipped with the k-hop approximation for changeset precomputation, as described in Chapter 4. Finally, Ginex++ is the mixture of above two techniques, the neighbor cache compression and the k-hop approximation for changeset precomputation, on top of Ginex. We show the effect of each technique by comparing all four of them in Section 5.3. After that, we directly compare Ginex with Ginex++ in the rest of the evaluations to demonstrate the effect of the mixture of the two acceleration techniques introduced in this work. We follow Ginex's guideline to decide the superbatch size: guaranteeing that the total size of runtime files falls within 3% of the target size (100GB). Specifically, the superbatch size is set to 16000, 8800, 15600, and 21600 for *papers*, *products*, *Friendster*, and *Twitter*, respectively.

5.2 Impact of Two Techniques

We first evaluate each of the two techniques to quantify their impact. Especially for Ginex+k-hop, we provide grounds for choosing an adequate k for practical usages.

5.2.1 Impact of Neighbor Cache Compression

Table 5.3: Comparison of neighbor cache hit ratio and superbatch sample time with or without compression. The size of the neighbor cache saved in the SSD is equally 45GB.

Dataset		*papers*	*products*	*Friendster*	*Twitter*
Non-compressed	Neighbor cache hit ratio	0.73	0.43	0.80	0.89
	Superbatch sample time (s)	331.93	690.51	250.06	251.18
ZSTD-compressed	Neighbor cache hit ratio	0.81	0.67	0.93	0.97
	Superbatch sample time (s)	267.44	419.21	216.69	248.61

In order to confirm the effect of the neighbor cache compression, we measure neighbor cache hit ratio and solo superbatch sample time with and without compression. The results are organized in Table 5.3. We only run a single superbatch to report the time. For all the four datasets, superbatch sample time reduces after the compression scheme is applied. Especially for *products* dataset, the speed-up of superbatch sample time records the highest ($1.65\times$). The reason is because neighbor cache hit ratio improves the most, from 0.43 to 0.67. *products* dataset has the biggest edges-per-node value among the four datasets, meaning that it is relatively more difficult for the neighbor cache to hold lots of information about target nodes' neighbor nodes. For this reason, its neighbor cache hit ratio is quite low before compression.

5.2.2 Impact of *k*-hop Approximation for Changeset Precomputation

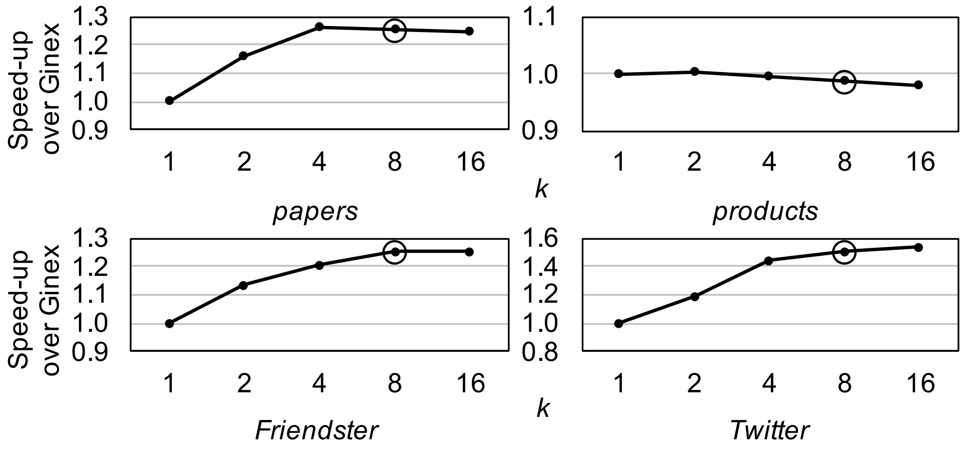


Figure 5.1: Speed-up of Ginex+*k*-hop over Ginex with varying *k*s

We evaluate our second technique by varying the value of *k*, and further provide a guideline for choosing an adequate *k*. Figure 5.1 shows speed-up of Ginex+*k*-hop over Ginex with *k*s varying from 1, 2, 4, 8, and 16. Note that Ginex+*k*-hop with *k* equal to 1 is same as the original Ginex. There is a gain in speed-up as *k* increases up to some point. For Ginex+*k*-hop and Ginex++, it is essential to choose proper *k* for applying *k*-hop approximation technique for changeset precomputation. We suggest to pick *k* as *8*, since the gain from increasing *k* quickly diminishes after such point. Therefore we use *k* as 8 for Ginex+*k*-hop and Ginex++ in all of the following experiments.

5.3 Overall Performance

We first provide normalized training time breakdown of Ginex, Ginex+NCC, Ginex+*k*-hop, and Ginex++ for the four datasets. Figure 5.2 demonstrates the results. There are six components in the training time: inspect, switch, data preparation, cache, transfer, and compute. Inspect time is the time spent for the parallel execution of superbatch sample and changeset precomputation. Switch time is the time corresponding to the feature cache

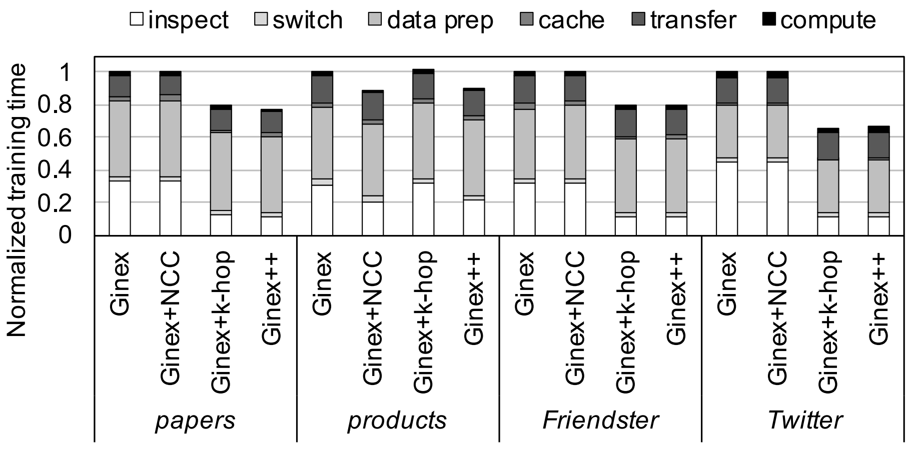


Figure 5.2: Normalized training time breakdown of Ginex, Ginex+NCC, Ginex+*k*-hop, and Ginex++. Smaller is better.

initialization. Data preparation time is the time for gather step and runtime file loading. Cache time refers to the time needed for cache update, and transfer time is the time needed for data communication between CPU and GPU. Lastly, compute time refers to the conventional GNN training computation for forward pass and backward pass.

For all the four datasets, Ginex++ shows superior performance. Ginex++ records the speed-ups of 1.29×, 1.11×, 1.25×, and 1.51× over Ginex for *papers*, *products*, *Friendster*, and *Twitter*, respectively. These performance gains stem from two techniques, neighbor cache compression and *k*-hop approximation for changeset precomputation.

For *papers*, *Friendster*, and *Twitter* datasets, major portions of the performance gains come from *k*-hop approximation. The reason for this is because changeset precomputation step is the bottleneck in inspect stage; it takes more time than superbatch sample. Therefore, by effectively shortening the time spent for changeset precomputation via *k*-hop approximation, inspect time is greatly reduced. To be specific, Ginex+*k*-hop shows 2.60×, 2.77×, and 4.14× speed-up in inspect time for *papers*, *Friendster*, and *Twitter* datasets, respectively. As a result, the portions of inspect time of the entire training time decrease from 0.33, 0.32, and 0.45 to 0.15, 0.15, and 0.17, for *papers*, *Friendster*, and *Twitter* datasets, respectively.

For *products* dataset, on the other hand, most of the performance gains come from neighbor cache compression. For this case, superbatch sample is the bottleneck in inspect stage. Thus, by applying neighbor cache compression, Ginex+NCC makes inspect faster than Ginex by 1.50$\times$. This reduces the portion of inspect time of the entire training time from 0.31 to 0.24.

It is not clearly shown in the figure, but a synergetic effect can exist between two techniques. If one technique shortens one part of inspect, and if the other part becomes the bottleneck in consequence, then applying the other technique can further improve the performance. For instance, when superbatch sample is the bottleneck, applying neighbor cache construction can alleviate the time spent on it. As a result of such application, it is fully possible that changeset precomputation becomes the new bottleneck. Then, applying k-hop approximation will very likely reduce the time spent on changeset precomputation, eventually resulting in further reduction in inspect time. For such scenarios, applying both techniques will result in synergetic effect.

5.4 Sensitivity Study

We perform several sensitivity studies on following three parameters that can significantly affect the performance: superbatch size, feature dimension, and batch size.

5.4.1 Impact of Superbatch size

We evaluate Ginex++ with different superbatch sizes in Figure 5.3. Specifically, we make adjustments to change the target runtime file size from 25GB to 100GB with the gap of 25GB. The training time is normalized to that of the default setting (i.e., when the target runtime file size is 100GB). Overall, it shows a consistent trend that larger superbatch size leads to better performance. The main reason for this is, as mentioned in the original paper [35], because the switch time, which is constant, is amortized. Nonetheless, it can also be seen that increasing the superbatch size leads to diminishing returns at certain point. This trend is

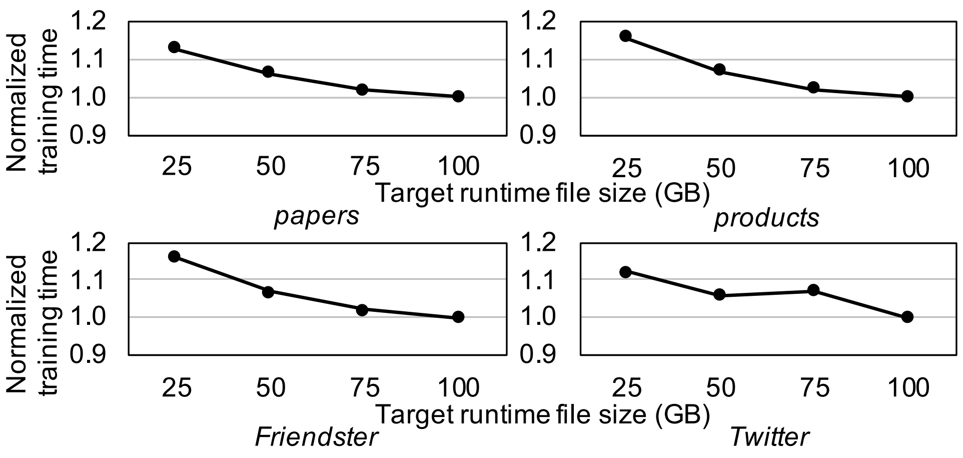


Figure 5.3: Training time of Ginex++ normalized to the default superbatch size on varying superbatch sizes

also similar to that of Ginex, and therefore is safe to use the same default setting as in Ginex.

5.4.2 Impact of Feature Dimension

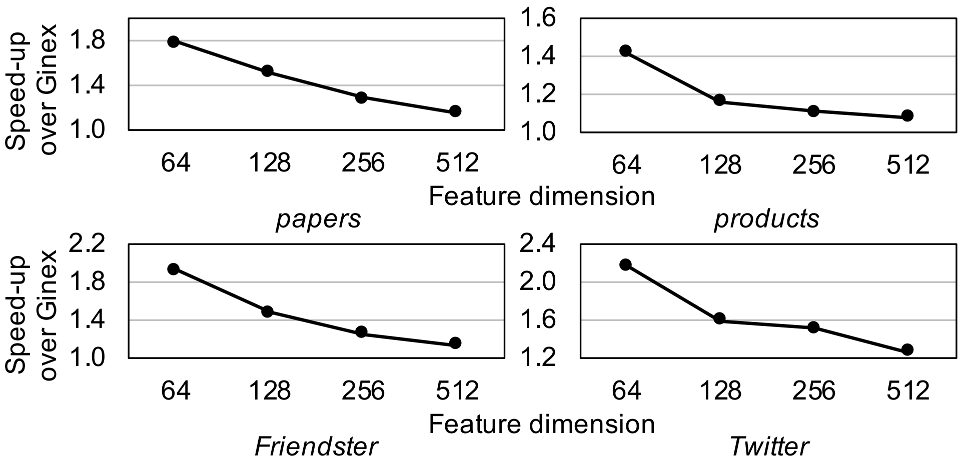


Figure 5.4: Speed-up of Ginex++ over Ginex with varying feature dimensions

Figure 5.4 shows the speed-up of Ginex++ over Ginex with varying feature dimensions. We run experiments by varying feature dimensions to $\times 1/4$, $\times 1/2$ and $\times 2$ of the default setting (256). For all the four datasets, Ginex++ ourperforms Ginex. There is also a common trend, that the speed-up gets greater as the feature dimension becomes smaller. This can be explained by the inspect ratio over the entire training time. As the feature dimension becomes bigger, the time consumed for gathering feature vectors is increased almost linearly. On the other hand, the time spent on inspect is shortened as the feature dimension becomes bigger; superbatch sample time is not affected by the feature dimension, but changeset precomputation time is reduced. This is because there is lesser chance for the feature cache to hold more feature vectors as the feature dimension grows. For example, when the feature dimension is 512, only about 5% of the whole feature data can be kept in Ginex's feature cache [35]. Thus, the overhead of changeset precomputation is alleviated as the feature dimension gets bigger. Combined with the increase of gather time and the decrease of changeset precomputation time, the effect of making the insepct ratio small is magnified as the feature dimension lessens. As mentioned throughout the work, if the inspect ratio is high, there is more chance for Ginex++ to reduce the training time using its two techniques. This is why Ginex++ shows even better performance for the cases where the feature dimensions are smaller.

5.4.3 Impact of Batch Size

Figure 5.5 demonstrates the impact of batch size. Ginex++ is faster than Ginex in all the cases. Not only that, it shows a trend that for smaller batch sizes, Ginex++ records much better performance. The major reason for this is because of the high inspect ratio when the batch size is small. As stated in the original paper [35], as the batch size decreases, changeset precomputation overhead can be the new bottleneck. The time for changeset precomputation reduces way slower than other stages, that scale down almost proportionally. Therefore, *k*-hop approximation is very effective for such case. For example, when batch size is 250, the inspect overhead ratio is reduced by 2.05×, 1.76×, 2.15×, and 2.17× in Ginex++ than Ginex for *papers*, *products*, *Friendster*, and *Twitter* datasets, respectively.

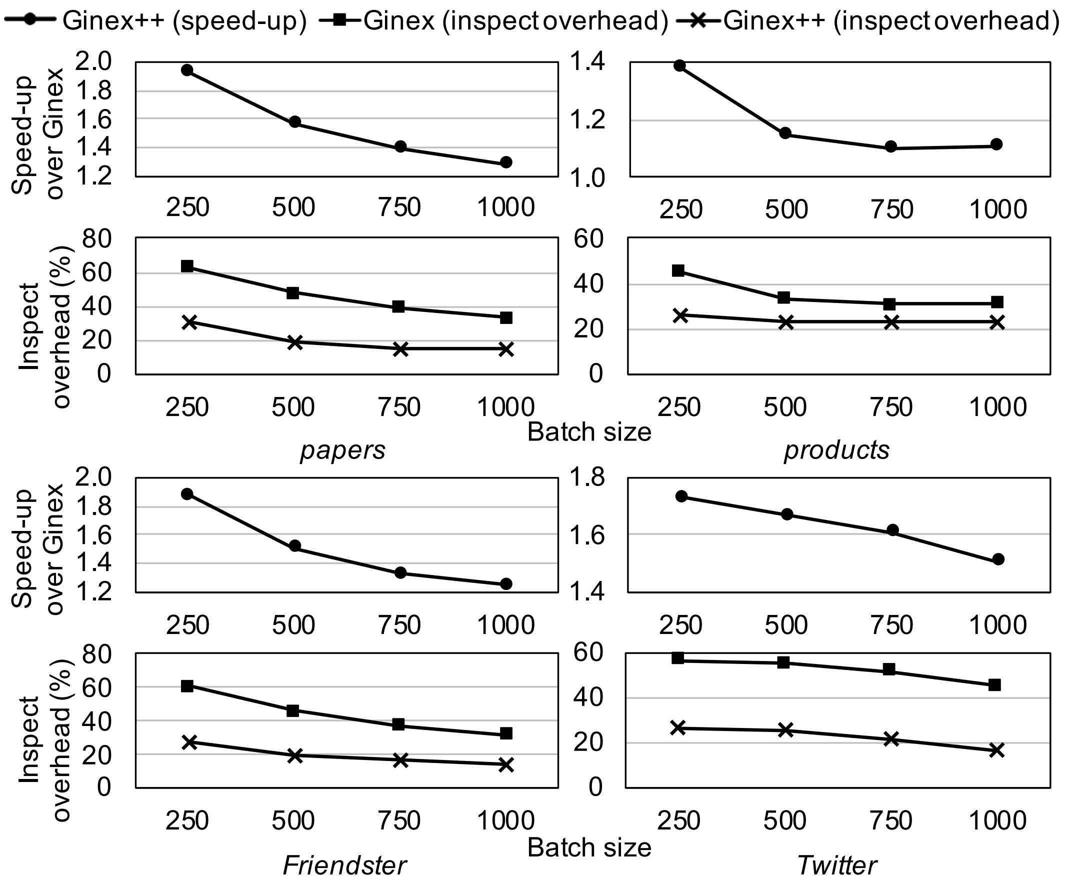


Figure 5.5: Speed-up of Ginex++ over Ginex, and inspect overhead of Ginex++ and Ginex with varying batch sizes

Chapter 6

Related Works

Scaling-up of Other Graph Workloads. Lots of proposals have been made to target the disk-based graph processing on a single machine on large datasets [19, 22, 29, 33, 36, 52, 56]. GraphChi [22] utilizes a technique called parallel sliding windows to enable graph workloads operate on a PC. X-stream [36] achieves to reduce the number of disk accesses by using an edge-centric method. FlashGraph [52] and MOSAIC [29] introduce their own graph data structure. Marius [33] targets graph embedding learning by partition caching and buffer-aware data orderings. These approaches take advantage of storage devices to substitute cluster-based approach. However, their focus is not on GNN workloads, rather on other graph processing workloads.

Scaling DNN Training with SSD. There have been proposals to exploit SSD to scale large-scale DNN training, but not GNN [2, 20, 30]. Dragon [30] and FlashNeuron [2] treat direct storage access as a secondary support for GPU memory. Behemoth [20] replaces HBM DRAM with flash memory and thereby proposes a DNN training accelerator. These works attempt to overcome GPU memory capacity wall in large-scale DNN training. This work, however, focuses on CPU memory capacity wall in GNN training for large-scale graph datasets.

Integer Compression Schemes. There are many compression schemes for integers, introduced to alleviate the storage cost [1, 12, 42, 57]. Binary packing [24] first calculates the minimum number of bits needed to represent the largest value in a block. Then, it is used to represent all the remaining values in the block. VariableByte [12] uses multiples of 7 bits to represent the actual value itself, and one bit for every 7 bits to indicate the end of every integer.

PForDelta [57] finds the smallest possible bits to represent the majority (e.g., 80%) of a block of multiple deltas (gaps). Simple16 [46] and Simple8b [1] both try to store as many integers as possible inside a given size of the array. They each uses an unique table of combinations of the values when storing the integers. Although Ginex++ uses the state-of-the-art generic compression scheme for neighbor cache compression, further performance enhancement can be made if a compression scheme specially targeted for integers is applied. Yet, since above schemes are designed for 32-bit integers, some modifications must be made to handle 64-bit integers. We leave this as a future work.

Chapter 7

Conclusion

Real-world graph datasets can scale up to several hundreds of GBs or even a few TBs, easily exceeding the main memory capacity. Ginex proposes a state-of-the-art SSD-enabled GNN training system on a single machine, in contrast with popular yet costly distributed training system. However, Ginex inevitably introduces a new overhead called `inspect`, in order to achieve the optimal in-memory feature caching. This overhead can be a potential and significant burden to the system according to various sampling sizes, batch sizes, types of datasets, and so forth. We thus propose Ginex++, which applies two acceleration techniques on top of Ginex. The two techniques are neighbor cache compression and *k*-hop approximation for changeset precomputation, directly targeted to reduce `inspect` overhead. As a result, Ginex++ successfully achieves 1.28$\times$ higher training throughput on average (1.51$\times$ at maximum) than Ginex, without any change in the training quality.

Bibliography

[1] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Software: Practice and Experience*, 2010.

[2] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "Flashneuron: Ssd-enabled large-batch training of very deep neural networks," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 387–401. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/bae

[3] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[4] F. Belletti, K. Lakshmanan, W. Krichene, Y.-F. Chen, and J. Anderson, "Scalable realistic recommendation datasets through fractal expansions," *arXiv preprint arXiv:1901.08910*, 2019.

[5] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *International Conference on Machine Learning*, 2018, pp. 941–949.

[6] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018.

[7] Y. Collet, "Lz4 - extremely fast compression," 2011.

[8] Y. Collet, "New generation entropy coders," 2013.

[9] Y. Collet, "Zstandard-fast real-time compression algorithm," 2018.

[10] W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi, "Minimal variance sampling with provable guarantees for fast training of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1393–1403.

[11] W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi, "Minimal variance sampling with provable guarantees for fast training of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1393–1403.

[12] D. Cutting and J. Pedersen, "Optimizations for dynamic inverted index maintenance," in *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1989.

[13] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 3844–3852.

[14] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[15] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Jul. 2021, pp. 551–568.

[16] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., 2017.

[17] 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," 2021.

[18] I. Kandel and M. Castelli, "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset," *ICT express*, vol. 6, no. 4, pp. 312–315, 2020.

[19] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, 2016, p. 447–461.

[20] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, "Behemoth: A flash-centric training accelerator for extreme-scale DNNs," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 371–385. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/kim

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[22] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Oct. 2012, pp. 31–46.

[23] Y. Lee, Y. Kwon, and M. Rhu, "Understanding the implication of non-volatile memory for large-scale graph neural network training," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 118–121, 2021.

[24] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, 2015.

[25] D. Lemire, L. Boytsov, O. Kaser, M. Caron, L. Dionne, M. Lemay, E. Kruus, A. Bedini, M. Petri, R. B. Araujo *et al.*, "The fastpfor c++ library: Fast integer compression," 2019.

[26] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[27] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[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*, ser. SoCC '20, 2020.

[29] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*. Association for Computing Machinery, 2017, p. 527–543.

[30] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: Breaking gpu memory capacity limits with direct nvm access," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 414–426.

[31] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," *Proc. VLDB Endow.*, 2021.

[32] R. Mirchandaney, J. Saltz, and R. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 05, pp. 603–612, may 1991.

[33] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association, Jul. 2021, pp. 533–549.

[34] A. Pal, C. Eksombatchai, Y. Zhou, B. Zhao, C. Rosenberg, and J. Leskovec, *PinnerSage: Multi-Modal User Embedding Framework for Recommendations at Pinterest.* New York, NY, USA: Association for Computing Machinery, 2020, p. 2311–2320. [Online]. Available: https://doi.org/10.1145/3394486.3403280

[35] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," *arXiv preprint arXiv:2208.09151*, 2022.

[36] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 472–488. [Online]. Available: https://doi.org/10.1145/2517349.2522740

[37] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018.

[38] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018.

[39] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[40] 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.

[41] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, p. 4–24, Jan 2021.

[42] H. Yan, S. Ding, and T. Suel, “Inverted index compression and query processing with optimized document ordering,” in *Proceedings of the 18th International Conference on World Wide Web*, 2009.

[43] H. Yang, “Aligraph: A comprehensive graph neural network platform,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3292500.3340404

[44] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’18, 2018.

[45] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, “Agl: A scalable system for industrial-purpose graph machine learning,” *Proc. VLDB Endow.*, 2020.

[46] J. Zhang, X. Long, and T. Suel, “Performance of compressed inverted list caching in search engines,” in *Proceedings of the 17th international conference on World Wide Web*, 2008.

[47] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, 2018.

[48] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge & Data Engineering*, vol. 34, no. 01, pp. 249–270, jan 2022.

[49] G. Zhao, T. Zhou, and L. Gao, "Cm-gcn: A distributed framework for graph convolutional networks using cohesive mini-batches," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 153–163.

[50] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, oct 2015. [Online]. Available: https://doi.org/10.1145/2808233

[51] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," 2021.

[52] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, Feb. 2015, pp. 45–58.

[53] D. Zheng, X. Song, C. Yang, Q. Su, M. Wang, C. Ma, and G. Karypis, "Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs," 2022.

[54] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," 2021.

[55] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "AliGraph: A comprehensive graph neural network platform," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2094–2105, aug 2019.

[56] X. Zhu, W. Han, and W. Chen, “GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning,” in *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association, Jul. 2015, pp. 375–386.

[57] M. Zukowski, S. Heman, N. Nes, and P. Boncz, “Super-scalar ram-cpu cache compression,” in *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

초 록

최근 그래프 신경망(GNN)을 사용하여 그래프로 구조화된 데이터에서 의미 있는 영감을 찾으려는 많은 노력이 이루어지고있다. 현실의 그래프 데이터는 시간이 지남에 따라 그 크기가 커지고, 이에 따라 단일 머신의 메모리에 맞지 않는 경우를 쉽게 찾을 수 있다. 이에 따라 최근 NVMe SSD와 같은 고성능 저장 장치를 활용하여, GNN 학습을 위해 단일 머신을 확장하는 여러 접근 방식이 제안되었다. GNN 학습을 위해 여러 머신을 동시에 사용하는 분산 시스템과 달리, 단일 머신에서 디스크를 기반으로 한 GNN 학습은 약간의 학습 시간 증가가 있긴 하지만 동등한 학습 품질을 보장하면서도 비용 측면에서 더욱 효율적인 대안을 제공한다. 그 중 하나인 Ginex [35]는 단일 머신에서 수십 억 규모의 그래프 데이터를 대상으로 GNN을 학습시키는 SSD 기반의 최첨단 학습 시스템이다. Ginex는 *sample* 단계와 *gather* 단계를 분리하는 방식으로 기존 GNN 학습 파이프라인을 재구성함으로써 *Belady's algorithm*으로 알려진 최적의 캐시 정책을 실현한다. 이렇게 Ginex는 *gather* 단계에서 걸리는 시간을 최소화하하는데 성공하였지만, 이 때문에 새로 도입된 *inspect*라는 오버헤드는 무시할 수 없게 되었다. 이에 본 논문에서는 *inspect* 오버헤드를 줄이기 위한 두 가지 가속화 기법을 Ginex에 적용하고 이를 Ginex++ 라고 명명한다. 두 기법은 이웃 캐시 압축과 changeset의 사전 계산을 위한 *k*-hop 근사이다. 네 개의 수십 억 규모의 그래프 데이터에 이러한 기법을 적용하여 평가한 결과, Ginex++ 은 Ginex보다 평균적으로 1.28× (최대 1.51×) 더 높은 학습 처리량을 달성한다.

주요어: 그래프 신경망 학습, 이웃 캐시 압축, *k*-hop 근사

학번: 2021-29333

Acknowledgement

지난 2년은 제 인생에서 가장 많은 변화를 겪게 된 기간이라 해도 과언이 아닙니다. 팬데믹으로 인해 전반적인 생활 패턴이 바뀐 것은 물론이고, 연구라는 낯선 무언가를 본격적으로 접하게 된 기간이기 때문입니다. 2020년 여름, 두 달 동안 집중적으로 논문을 작성하는 팀에 들어가 연구실 인턴십을 성공적으로 마쳤을 때는, 몸은 힘들었지만 뿌듯한 감정과 더 연구를 경험해보고 싶다는 막연한 호기심이 생겼습니다. 그러면서 본격적으로 대학원 진학에 대한 고민을 시작했습니다.

'어떤 연구실의 대학원생이 자신의 지도 교수님에 대해 좋게 얘기한다면, 그 연구실은 믿고 들어가도 된다'는 우스갯소리가 있습니다. 제가 대학원 지원과 연구실 선정을 고민할 때, 2020년 봄 학기에 논리 설계 과목 학부생 조교를 하면서 친분을 쌓게 된 성훈이와 샘이가 지도 교수님의 칭찬을 자주 했던 것이 큰 영향을 미쳤습니다. 그리고 그로부터 약 2년이 지난 지금, 저는 당시에 내릴 수 있는 최고의 결정을 내렸다고 자부합니다.

첫 번째 이유는 교수님입니다. 이재욱 교수님은 따뜻할 땐 따뜻하게, 냉철할 땐 냉철하게 학생들을 지도해주시는 참된 지도자이십니다. 학생이 스스로 연구를 해나갈 수 있게 자율성을 존중해주시면서도 올바른 방향으로 나아갈 수 있게 적절한 조언을 해주십니다. 이런 점들을 볼 때 교수님은 중용(中庸)의 대가이신 것 같습니다. 학생들도 이런 점들을 잘 알기에 교수님을 전적으로 신뢰하고, 교수님의 요구사항에 대해 신속하고 적절한 피드백을 내놓습니다. 이런 든든한 연구실 문화를 만들어내신 존경하는 교수님께서 저와 MBTI가 ISFJ로 같다는 것을 알았을 때 내심 기분이 좋았습니다.

두 번째 이유는 연구실 동료들입니다. 저는 일을 할 때 주변 사람들이 제게 미치는 영향이 정말 중요하다고 생각하는 사람입니다. 그렇기 때문에 실력으로나 인격으로나 훌륭한 연구실 동료들이 주변에 많이 있어서 참 감사하다는 생각을 많이 했습니다. 제 졸업을 책임져 주고 해외 학회를 함께한 연홍이, 팀으로서 같이 일할 기회를 얻게 되어 진심으로 감사했던 준이 형, 예진 누나, 현지, 승렬이, 불멸의 서버 관리자 성훈이, 현승이 형, 상우, 박사 후 연구원으로서 날카로운 지도를 해주셨던 함 박사님과 종현이 형, 아쉽게도 같이 일을 하지는 못했지만 항상 응원하는 문경 누나, 성준이 형, 종성이 형, 훈이 형,

동욱이 형, 수성이, 봉근이 형, 리해, 원석이 형, 동현이, 근수, 용상이 형, 그리고 마지막으로 궂은일을 도맡아 해주시는 연구실 최고의 분위기메이커 미림 선생님. 다시 한 번 깊은 감사의 말씀 올립니다.

세 번째 이유는 풍족한 연구 환경입니다. 교수님께서는 연구를 위한 것이라면 지원을 아끼지 않으셨습니다. 덕분에 최신 사양의 서버와 클라우드를 사용할 수 있었고, 논문 집중 기간에는 맛있는 음식을 먹으며 연구에 집중할 수 있었습니다. 주변에 저희 연구실의 풍족함을 자랑하면 다들 많이 부러워했습니다.

이 외에 크고 작은 이유들을 더 나열하자면 지면이 부족할 것입니다. 연구실에 와서 후회를 하고 좋지 않은 기억을 가진 채 나가는 사람들도 많은데, 이렇게 행복한 마무리를 할 수 있는 저는 분명히 큰 복을 받았다고 할 수 있겠습니다. 연구를 할수록 아이러니하게도 '이를 이용해서 현업에서는 어떤 일에 적용할 수 있을까'라는 궁금증이 더욱 커지게 되어 스타트업에 취업을 하게 되었지만, 연구에 대한 호기심과 욕심은 지금도 마음 한 구석에 자리해있습니다. 미래에 또 어떤 결정을 내리게 될지 아직은 모르지만, 그 결정에 있어 연구실에서의 경험은 더없이 귀중한 기제가 될 것이라 믿어 의심치 않습니다.

연구실 구성원들 모두가 각자의 자리에서 순항하시길, 그리고 이후에도 꼭 다시 만나 뵐 수 있기를 진심으로 기원합니다. 모두에게 정말 감사합니다.

2023년 1월,

민선홍 올림.