DEMYSTIFYING GRAPH NEURAL NETWORKS IN RECOMMENDER SYSTEMS

BY

HOUXIANG JI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Recommender systems have become indispensable tools for many applications with the explosive growth of online information. Recommender systems learn user's interest based on their profile and historical behavior and then recommend the item with the highest predicted ratings. Graph Neural Networks (GNNs) is a powerful graph representation learning method and have shown their unprecedented performance on many scenarios including natural language processing and computer vision. Most data required in the recommender systems is naturally and essentially represented with graph structure, for example, user-item interactions can be represented as a bipartite graph. With the superiority in graph learning, GNNs are gaining more and more attention in the field of recommender systems.

This thesis presents our study of GNNs, which are employed in the recommender systems, to characterize their runtime behavior from various levels. To this end, we carefully profile GNN models on training and identify the most expensive operations which are worthy of attention for overhead reduction. We observe that the memory-intensive aggregation phase in the GNNs dominate the overall runtime different from the conventional deep neural network models. Further, we investigate the GNN's behavior at the micro-architectural level, which have received little attention so far, and summarize several key observations. Based on these insightful observations, we propose and discuss software and possible hardware mechanisms to optimize GNN-based recommender systems.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Given this opportunity, I want to greatly thank Professor Josep Torrellas for his great support and guidance over the past three years. We collaborated on several projects ranging from hardware security to graph neural networks. I learned different skills in these projects and got growth as a researcher. Beside all the guidance, I am impressed by Josep's serious and passionate attitude towards research. Thanks Josep for all the support. I also would like to thank all my collaborators on the projects. Thanks for all your help.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

With the rapid development of online business and social media platforms, recommender systems are deployed and utilized for many scenarios, like product suggestions on online e-commerce websites, recommending potential new friends on the social media and generating playlists that users may like for video and music services. Recommender systems alleviate the information overload problem for users and predict their preference from a vast volume of information. Recommender systems learn the users' preferences from but not limited to user-item historical interactions, user demographic features and social information.

Representation learning based methods [1, 2] which encode the user and items as continuous vectors or embeddings, have gradually replaced the neighborhood method [3, 4] in the recommender systems because of superior accuracy. Various models have been proposed to learn representations of users and items for better preference estimation, from matrix factorization [1, 5] to current prevailing deep learning models [2, 6]. Among all the deep learning models, graph neural network (GNN) is the most attractive technique because of its expressive power in representation learning from graph-structured data which are the main format of data used in recommender system. For example, user can be modeled as nodes and their friend connection is represented by the links in the social network. In addition to the graph structure, the structural external information can also be integrated to the graph-structure data easily. In the social relationship graph example, external information can be attached on each node as a feature vector. GNN explores the graph structure and external information at the same time and provides a unified perspective to model the abundant heterogeneous data in recommender system.GNN-based models outperform previous techniques and achieve state-of-the-art accuracy in both academic research and industry work. For different recommendation tasks, plenty of variants are designed and proved to be useful. PinSage [7] is a variant of graph convolution network, which is developed and deployed in web-scale recommender systems in Pinterest.

GNN typically consists of two major phases: aggregation and update. The information from neighbors are aggregated to the central node and the aggregated information is processed by functions to update the feature vector of the node. These two key designs enable GNN to capture the graph structure information and external information attached on each node.Understanding the underlying behavior of these two phases or whole training process is vital to optimizing the GNN model and accelerating the GNN process. Most prior works from recommendation community focus on the accuracy improvement by designing new network structure but ignore the efficiency and runtime overhead for the models. Moreover,

prior analysis on GNNs is limited to algorithm level and GPU platform, which falls short of details required to accelerate the GNN process on CPUs. For example, aggregation phase is identified as the most time-consuming phase in several accelerator works [8, 9] but their profiling only measures the execution time without detailed characterization. In this thesis, we present a more detailed characterization of representative GNN models in recommender systems from microarchitectural level, making following key contributions:

- With careful characterization of GNN training, we identify the key operations and phases which are waiting to be optimized for acceleration.
- Different from conventional deep learning models, memory-intensive operations make up the majority of the overall runtime and we investigate the system to find the hardware bottleneck like limited memory bandwidth.
- We also study the scalability of GNN models on multiple CPUs and workload imbalance problem emerges due to the inherent graph property and dependencies in the GNN model.
- Finally, we discuss the potential optimization design from both software and hardware level based on the key observations we found in the characterization.

CHAPTER 2: BACKGROUND

This chapter provides an overview of the recommender systems and the prevailing models deployed in these systems. Basic terminology and concepts regarding the graph neural network are introduced. We also discuss the motivation of utilizing graph neural networks in recommender systems and give a brief introduction to existing GNN-based models.

2.1 RECOMMENDER SYSTEMS

Recommender systems estimate users' preference on a set of items and recommend the items with highest ratings to the users. [10, 11]. The estimation of ratings is based on the related information about the items, the user and the interactions between items and users.[12]. Recommender systems has been a popular research area since the mide-1990s and constitutes a problem-rich research area. The abundance of practical applications also makes the area attractive. Recommender systems are usually classified into three categories based on how the recommendations are made: collaborative filtering, content-based recommendations and hybrid approaches. [10]. Collaborative filtering recommends the items that people with similar tastes and preferences liked in the past by learning from the user-item explicit (e.g. ratings) or implicit (e.g. browsing history) interactions. In content-based recommendations, users will be recommended items similar to the ones the users preferred in the past. Demographic features and social information of the users are also considered in the latest models. Hybrid approaches integrate two or more types of recommendation strategies to give the final recommendation.

The aforementioned models like collaborative filtering, assume the uses have static preferences and build the user profile based on the historical interactions ignoring the sequential information [11]. Sequential recommendation is another mainstream research direction in the recommender systems, which assumes the users' preference is dynamic and evolving and seeks to recommend the successive items by exploring the sequential patterns in the user-item interaction history [13]. Sequential recommendation can be further divided into sequential recommendation and session-based recommendation based on the anonymity of users and whether the behaviors are segmented into sessions. The main challenge of sequential recommendation is to learn an efficient sequential representation of user's varying preference. Markov Chain was adopted in early work. Owing to advantage of Recurrent Neural Network (RNN) in sequence modeling, some works employ the RNN to capture the sequence representations. Graph Neural Network (GNN) [14] also becomes popular as it can capture the

complex transition patterns as well. We give a more detailed introduction about GNN in the following sections and explain the reasons why GNN are popular in recommender systems.

2.2 GRAPH NEURAL NETWORKS

Graph neural networks(GNNs) is a deep learning based model which aims at the non-Euclidean data, more specifically graph domain. Graphs are a kind of data structure in which nodes may represent the objects and the relationships between objects are modeled as the edges. Graphs have shown their great expressive power in many tasks such as node classification, link prediction, and clustering. The non-Euclidean structure hinders conventional machine learning techniques like convolution neural network (CNN).

2.2.1 General Formulation of GNNs

The main idea of GNNs is to aggregate the information from neighbors and update the central node representation based on the aggregated information [13]. The aggregated information is processed by functions defined by the GNN model and final representations in the nodes after GNN execution are utilized to solve the problems, like predicting the functionality of a protein. Here we present the general formulation of GNNs. Table 2.1 lists the notations used in GNN.

	Description		Description
$\mathcal{G}$	graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$	$\mathcal{V}$	vertices of G
$\mathcal{E}$	edges of $\mathcal{G}$	D_v	degree of vertex v
$\mathcal{N}(v)$	all neighbors of vertex v	$\mathcal{S}(v)$	sampled subset of $N(v)$
ψ	feature processing function	$\mathbf{A}$	adjacency matrix
K	number of layers	H	vertex feature vector length
$\mathbf{h}$	feature matrix	$\mathbf{h}_v$	feature vector of vertex v
$\mathbf{a}$	aggregation feature matrix	$\mathbf{a}_v$	aggregation feature vector of vertex v
$\mathbf{W}$	update weight matrix	$\mathbf{b}$	update bias vector

Table 2.1: GNN symbols

From the perspective of network architecture, GNN typically stack K propagation layers, which consists of aggregation and update phases. Simply, in the aggregation phase of $k-th$ layer, **Aggregation** function aggregates multiple feature vectors from neighbors $\mathcal{N}(v)$ and itself from layer $k-1$ to each vertex in the layer $\mathbf{a}_v^k$. Common aggregation functions include average function, max pooling and addition.

The aggregation feature vector $\mathbf{a}_v^k$ is passed to the update function and generate the feature vector $\mathbf{h}_v^k$ for this layer. The **update** function could be a shallow neural network or any

differentiable function. The formulation for two phases are shown in equation 2.1 and 2.2 respectively.

$$\mathbf{a}_v^k = \text{AGGREGATE}(\mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \cup \{v\}) \tag{2.1}$$

$$\mathbf{h}_v^k = \text{UPDATE}(\mathbf{a}_v^k) \tag{2.2}$$

After K layers, each vertex's feature vector contains the information from the neighbors which are up to K hops away. There is a final **Readout** function applied on the feature vectors after K iteration and generate reasonable predictions. The overhead of readout function is always negligible compared with the whole training process. In this thesis, we may use vertex and node interchangeably. Readout function on the whole graph can be formulated as:

$$\mathbf{h}_G = \text{READOUT}(\mathbf{h}_v^k \mid \forall v \in G) \tag{2.3}$$

Readout function can work on single node as well:

$$\mathbf{h}_v = \text{READOUT}(\mathbf{h}_v^k \mid v \in G) \tag{2.4}$$

In order to reduce both the computational complexity and the memory footprint, the **Sample** function is usually applied before aggregation function. In sampling, we randomly select up to some pre-determined number of neighbors for each vertex shown in equation 2.5. Sometimes, the Pool function follows the update function to transform the original graph into a smaller graph [15].

$$\mathcal{S}(v) = \text{SAMPLE}^k(\mathcal{N}(v)) \tag{2.5}$$

The sampling step is essential for executing GNNs with large input graphs on memory-limited devices such as GPUs and accelerators [16, 7]. And the sampling step makes aggregation and update phase quicker with less data to process. The size of selected neighbors are predefined when constructing the GNN. Generally speaking, the larger sample size, the better the performance as the nodes get information from more neighbors. People may worry about the performance of the GNN, like accuracy on prediciton tasks. Fortunately, based on [16], for some graphs, there is no additional benefit to aggregate information from more neighbors after a threshold. Sampling is not enough to fit the graph in memory when the graphs become extremely larger like ogbn-papers100M dataset in Open Graph Benchmark which has more than 100 million nodes and 1 billion edges [17]. A common processing method, mini-batch training, is then combined with sampling together to fit the subgraph into memory. Search algorithms like breadth-first search (BFS) are employed to find the K-hop neighborhood of each vertex in a mini-batch for a K-layer GNN.

In summary, GNNs involve several typical operations: Sampling, Aggregation, Update and Readout. Except the update function, the other operations are graph structure-dependent. Sampling can be done during preprocessing [18] or with random selection at runtime [16]. In this work, we ignore the Readout phase as it is only performed once and the overhead is negligible. Multiple GCN models will be given as examples in next section to explain the above operations in detail.

2.2.2 Typical GNN Models

GNN models are classified by aggregation and update functions. We provide several popular GNN models as examples to show how GNNs work. Graph Convolutional Network (GCN) [19] is one of the most widely used GNN, which bridges the gap between spectral-based and spatial-based convolution. It is formulated as:

$$\mathbf{a}_v^k = \sum \frac{1}{\sqrt{D_v \cdot D_u}} \mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \cup \{v\} \tag{2.6}$$

$$\mathbf{h}_v^k = ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k) \tag{2.7}$$

GraphSage [16] adopts the uniform sampling to alleviate the receptive field expansion. Sampling not only saves the memory footprint but also accelerates the inference phase (aggregation and update phases). It is formulated as:

$$\mathbf{a}_v^k = \sum \frac{1}{D_v + 1} \mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{S}(v) \cup \{v\} \tag{2.8}$$

$$\mathbf{h}_v^k = ReLU(\mathbf{W}^k a_v^k + \mathbf{b}^k) \tag{2.9}$$

$\mathcal{S}(v)$ is defined in equation 2.5, indicating the selected neighbor set.

Graph Attention Network (GAT) [20] leverages the attention mechanism to differentiates the contributions of neighbors and updates the feature vector of each node by attending over its neighbors.

$$\alpha_{uv} = \frac{exp(LeakReLU(p^T[W^{k-1}\mathbf{h}_v^{(k-1)} \bigoplus W^{k-1}\mathbf{h}_u^{(k-1)}])}{\sum_{k \in \mathcal{N}(v)} exp(LeakReLU(p^T[W^{k-1}\mathbf{h}_v^{(k-1)} \bigoplus W^{k-1}\mathbf{h}_k^{(k-1)}])} \mid u \in \mathcal{N}(v) \tag{2.10}$$

$$\mathbf{h}_v^k = ReLU(\sum_{u \in \mathcal{N}(v)} \alpha_{uv} \mathbf{W}^{k-1} \mathbf{h}_u^{(k-1)} \tag{2.11}$$

p is learnable parameter across iterations like W. This breaks the assumption that central

node sends/receives the identical influence to/from its neighbors.The formulations are shown in equation 2.10 and equation2.11.

Gated Graph Neural Networks (GGNN) is a typical RecGNN methods, which adopts a gated recurrent unit (GRU) for update. GGNN is mainly for sequential predictions.It is formulated as:

$$\mathbf{a}_v^k = \frac{1}{|\mathcal{N}(v)|} \sum \mathbf{h}_u^{(k-1)} \mid \forall u \in \mathcal{N}(v) \tag{2.12}$$

$$\mathbf{h}_v^k = GRU(\mathbf{h}_v^{(k-1)}, \mathbf{a}_v^k) \tag{2.13}$$

From the four GNN models, we can see that update function is really simple, a fully connected (FC) layer activated by the rectified linear unit (ReLU) in GCN, GraphSage and GAT. GGNN uses special GRU as it focuses on the sequential patterns. The aggregation function of four models gather each vertex's neighbors' feature vectors and reduce the gathered feature vector to a simpler one. GraphSage is different from the others with $\mathcal{S}(v)$ instead of all neighbors $\mathcal{N}(v)$. Generally speaking, all these models can adopt sampling by replacing $\mathcal{N}(v)$ with $\mathcal{S}(v)$ in aggregation.

2.3 GRAPH NEURAL NETWORKS FOR RECOMMENDER SYSTEMS

Before introducing the GNN models which are adopted in the field of recommendation, we talk about the motivations of applying GNN to recommender systems. The most intuitive reason is that GNN techniques have shown its expressive power in representation learning for graph data in various domains [14] and most data required in recommender systems has essentially a graph structure. For example, user-item interactions can be treat as bipartite graphs between user nodes and item nodes, where the edge represents the interactions.Social relationship between users is easy to be modelled as a graph. Knowledge graph and citation graph [17]. A sequence of items can be transformed into the sequence graph, where each item can be connected with one or more subsequent items.

Due to the various characteristics of different types of data, a variety of models have been proposed to estimate the users' preferences [21] and each of the models is limited to the target data type. A model which is good at exploring the useful patterns in social network may fail to recommend suitable items based on the sequential tasks. A unified GNN framework can address all these tasks considering the inherent graph structure in the information. Both node property and graph property can be learned through GNN. Besides, it is convenient and flexible to include additional information if available in the future in the graph by editing

the feature vector on nodes or changing the edges.

Collaborative filtering (CF), one of the widely used idea in recommender systems, is explicitly encoded in the GNN through propagation process and the CF signals is utilized for a better representation learning and rating prediction. In addition, multi-hop connectivity among user-item interactions can be modeled by GNN more conveniently. Dedicated GNN frameworks like PyTorch Geometric [22] and Deep Graph Library [23] are developed on top of the general machine learning frameworks such as Tensorflow [24] and PyTorch [25]. With the help of these frameworks, researchers can develop new GNN models and verify them more quickly on the tasks.

Many of GNN models used in recommender systems can be seen as the variants of the previous GCN, GraphSage and GAT. LR-GCCF [26] is a variant of GCN without activation. IG-MC [27] is a variant of GraphSage with sum updater. MCCF[28], Gemini[29] and GraphRec[30] are variants of GAT.More Graph neural network based recommendation models can be check in [13].

CHAPTER 3: METHODOLOGY

In this chapter, we first describe the platform and the tools we used to study GNN in recommender systems. We provide a detailed description of GNN models and conventional recommendation model, following which we provide the statistics for the datasets.

3.1 EXPERIMENTAL SETUP

The platform for the evaluation is a 22-core Intel Skylake server CPU with AVX-512 vector extensions. Each core has a 32KB L1 data cache, a 1MB private L2 cache, and a 1.375MB slice of a non-inclusive shared L3 cache. The CPU is clocked at 2.1GHz without dynamic frequency scaling. The maximum DRAM bandwidth is 54GB/s. These 22 cores are from the same socket to avoid inter-socket communication. We uses Intel OpenMP instead of GOMP. By default, we disable the SMT on the cores and run 22 threads in total.

3.2 PROFILING TOOLS

To characterize GNNs in recommender systems, we mainly use two tools for assistance: Pytorch profiler [31] and Intel Vtune from Intel OneAPI Base Toolkit [32]. Pytorch profiler is an open-source tool that enables accurate and efficient performance analysis.There were standard performance debugging tools that provide CPU or GPU hardware-level information but missed Pytorch-specific context of operations. Pytorch profiler allows one to check which operators were called during the execution of a code range wrapped with a profiler context manager. We mainly use Pytroch profiler to analyze the execution time of the models on dataset.

Intel Vtune Profiler is a performance analysis tool for serial and multi-threaded applications. Vtune Profiler has many important analysis types including Hotspots analysis, HPC performance characterization analysis, Microarchitecture Exploration, Memory Access analysis and so on. As we already have Pytorch profiler to get the high-level information, we mainly use the Microarchitecture exploration with ITT API analysis to learn about the microarchitectural information on the execution. During Microarchitecture Exploration analysis, the VTune Profiler collects a complete list of events for analyzing an application. Microarchitecture Exploration analysis is based on the Top-Down Microarchitecture Analysis Method, which is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application. Processors are conceptually divided

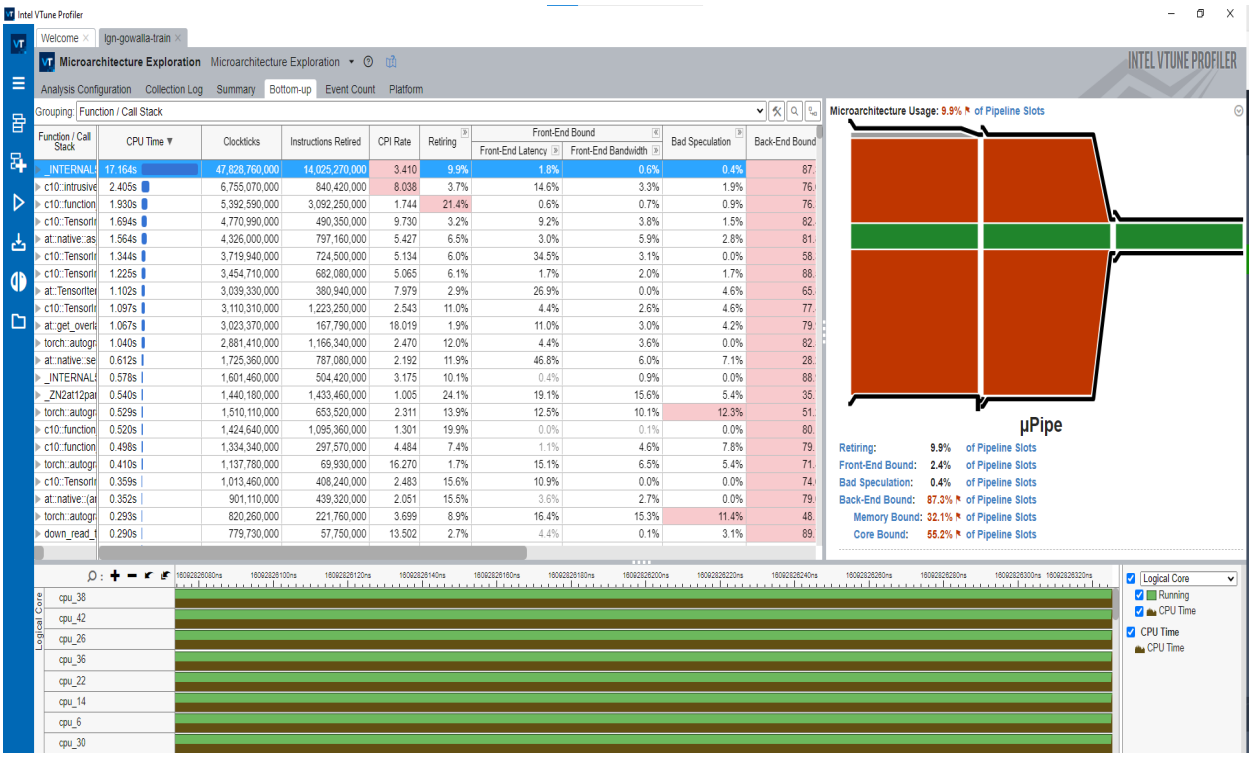

Figure 3.1: Vtune Profiler Example Output

into front-end, where fetch and decode the instructions, and the back-end, where the required computation is finished in the analysis.The operations are placed by front-end into pipeline slots and moved through the back-end. At each cycle, Vtune classify all the pipeline slots into four categories: retired, bad speculation, front-end bound and back-end bound. Retired pipeline slots contains useful work that get issued and retired. Bad speculation slots contains useful work that issued but cancelled. Front/Back-end bound slots could not be filled with useful work due to problems in the front/back-end. Front-end bound execution may be due to a large code working set, poor code layout, or microcode assists. Back-end bound execution may be due to long-latency operations or other contention for execution resources. Bad speculation is most frequently due to branch misprediction. Front/back-end bound pipeline slots are further identified as latency or bandwidth bounded. Back-end bound pipeline slots are further divided into L1 bound, L2 bound, L3 bound, DRAM bound, Store bound and Core bound slots. An example output from Vtune Profiler is shown in Fig 3.1. Based on the microarchitecture exploration results, we find the bottlenecks during the GNN model execution and corresponding functions. Vtune only knows about hardware and only the low-level functions are shown like tensor computation function. There is no information about where this function is called in the GNN architecture, so we still need to figure them

out with the expertise in both algorithm and microarchitecture.

3.3 DATASET DESCRIPTION

To characterize the GNN models for recommendation, we conduct experiments on three representative benchmark datasets in recommendation tasks: Gowalla, Yelp2018, Amazon-Book and Last.fm. To explore more about models, the benchmarks vary in terms of domain, size and sparsity. The statistics of datasets is shown in Table 3.1.

Dataset	*User#*	*Item#*	*Interaction#*	*Density*
Gowalla [33]	29,858	40,981	1,027,370	0.00084
Yelp2018 [34]	31,668	38,048	1,561,406	0.00130
Amazon-book [35]	52,643	91,599	2,984,108	0.00062

Table 3.1: Statistics of the experimented dataset

Gowalla: contains user-venue check-ins from a location-based social network. To ensure the quality of the dataset, we use 10-core setting [36] which means that after users and venues have a minimum of 10 check-ins, the data is valid. **Yelp2018**: is adopted from the 2018 edition of Yelp challenge. The local business are viewed as the items and people give their rates on them. **Amazon-book**: is one from the Amazon-review collection [35], a widely used dataset for product recommendation. Simiarly, 10-core setting is used to ensure the each user and item have at least ten interactions.

3.4 RECOMMENDATION MODEL DESCRIPTION

To demonstrate the different characterization of models in recommender systems, we not only select the GNN-based recommendation model: Neural Graph Collaborative Filtering (NGCF) [37] and LightGCN [38], but also include one conventional recommendation model: Matrix Factorization (MF) [39].

Different from typical GNN models present in the section 2.2, GNN models in the recommender systems require the clarification between user and item nodes in the graph. The additional notations used in the recommender systems are listed in the table 3.2

MF algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. MF became widely known for its effectiveness and performance on Netflix prize challenge [1].Given $I \in R^{m \times n}$, MF learns a user embedding matrix $U \in R^{m \times d}$ and item embedding matrix $V \in R^{n \times d}$ by decomposing A. d is length of

	Description		Description
M	Number of users	N	Number of items
e_u	Embedding of user u	e_i	Embedding of item i
$\mathcal{N}(u)$	items interacting with user u	$\mathcal{N}(i)$	users interacting with item i
e_u^k	refined user embedding after k layer	e_i^k	refined item embedding after k layer

Table 3.2: Additional symbols in recommender systems

latent factors. Embedding in recommender system is the same with feature vector in GNN. We use them interchangably in the following sections.The predicted ratings are calculated by $\tilde{I} = U \times V^T$. One objective function is to minimize the distance between I and $\tilde{I}$. By MF, the empty slots in the A are filled by the predictions in $\tilde{A}$ which are also the predicted ratings we want.

NGCF largely follows the standard GCN [19] model which is shown in section 2.2. NCGF explicitly encodes the collaborative signal, which is latent in user-item interactions, in the form of high-order connectivities in user-item bipartite graph by performing embedding propagation.In contrast, methods like MF lack the ability to encode the collaborative signal in embeddings by only mapping from pre-existing features such as user ID and attributes. For a connected user-item (u,i), the message embedding (aggregation feature vector) from i to u is:

$$\mathbf{a}_{ui}^k = \frac{1}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(i)|}}(W_1^{k-1}\mathbf{e}_i^{k-1} + W_2^{k-1}(\mathbf{e}_i^{k-1} \odot \mathbf{e}_u^{k-1}) \tag{3.1}$$

After receiving the message from connected items, user u update its embedding by:

$$\mathbf{e}_u^k = LeakyReLU(W_1^{k-1}\mathbf{e}_u^{k-1} + \sum_{i \in \mathcal{N}(u)} \mathbf{a}_{ui}^k) \tag{3.2}$$

W_1^k and W_2^k are layer-wise trainable weight matrix to perform feature transformation at layer k. NGCF performs the same operations from item node to user node. We skip the equations for items. After propagation through K layers, NGCF concatenates $K+1$ embeddings from each layer to obtain the final user embedding $e_u = (e_u^0, e_u^1, ..., e_u^K)$ and item embedding $e_i = (e_i^0, e_i^1, ..., e_i^K)$. The prediction score for each user-item pair is generated by the inner product between these two final embedding.

$$\mathbf{a}_{ui}^k = \frac{1}{\sqrt{\mathcal{N}(u)\mathcal{N}(i)}}\mathbf{e}_i^{k-1} \tag{3.3}$$

LightGCN is built upon the NGCF but simplifies the NGCF model greatly. The authors of LightGCN performed the ablation studies on NCGF to evaluate the usefulness of each operation in the NGCF and cut the useless or even negative designs in the NCGF. Two

common designs, feature transformation and nonlinear activation, are proven to be harmful to CF effect and thus eliminated in LightGCN design. The aggregation from i to u on user-item pair(u, i) in LightGCN is formulated in equation 3.3, and the update phase is then simplified to:

$$\mathbf{e}_u^k = \sum_{i \in \mathcal{N}(u)} \mathbf{a}_{ui}^k \tag{3.4}$$

Final user and item embedding become: $e_u = \sum_K^{k=0} \alpha_k \mathbf{e}_u^k$ and $e_i = \sum_K^{k=0} \alpha_k \mathbf{e}_i^k$. α_k is equal or larger than zero, denoting the importance of the k-th layer embedding in constituting the final embedding. In the layer combination, LightGCN uses sum instead of concatenation and then perform the inner product on the embeddings. The sum operation reduces the final feature vector length and saves the time.

3.4.1 Why we investigate NGCF and LightGCN

We choose NGCF and LightGCN in our study for several reasons. At first, both NGCF and LightGCN are the state-of-the-art GCN models for recommendation with competitive accuracy. Second, NGCF is a variant of GCN which is the most widely used model in the whole GNN community and also the most influential one. Insights on the NGCF can be applied to the GCN and provides guidance for further model design or GCN accelerator design. Third, LightGCN simplifies the NGCF by including only the most essential components in GCN for recommendation and shows substantial improvements over NGCF. LightGCN is the kernel model of the GCN to some extent. Fundamental operations are present in LightGCN, including aggregation, update, and readout. We also integrate the sampling into LightGCN implementation to cover all the important operations. Based on these reasons, we believe LightGCN is a good representatives of GNNs in recommender systems to profile.

3.4.2 Parameter Settings

All the models are implemented in PyTorch [25]. Both LightGCN and NGCF are intialized with the Xavier method [40] and the embedding size is fixed to 64. MF is initialized with normal distribution $\mathcal{N}(0, 1)$ and the embedding size is also fixed to 64. We optimize NGCF and LightGCN with Adam [41] and use the default learning rate of $1e^{-3}$. We discard the mini-batch in the original papers [37, 38] and use full-batch training instead as our platform has enough memory space with 22 cores.

To achieve the accuracy reported in the paper [38], we fine-tuned the LightGCN's hyper-parameters including L2 regularization coefficient λ, layer combination coefficient α_k and

number of layers K. After searching the setting space, the optimal value λ is $1e^{-5}$ and $\alpha_k = \frac{2}{1+K}$. The number of layers is set to 3 to achieve the best performance. NGCF share the same settings with LightGCN and LeakyReLU slope is set to 0.2 by default in it. The early stopping and validation strategies are the same with the paper[37]. LightGCN, NGCF and MF employ the *Bayesian Personalized Ranking* (BPR) loss [39].

CHAPTER 4: GNN CHARACTERISTICS IN RECOMMENDER SYSTEM

In this chapter, we first provide high-level and phase-to-phase runtime breakdown of the selected GNN Models (NGCF and LightGCN) and MF on three datasets. After analysis on the runtime, we conduct the microarchitectural analysis on LightGCN model via Intel Vtune [32] to observe the GNN behavior and identify the bottlenecks on a lower level. These observations are valuable for designers to accelerate the GNN process.

4.1 EXECUTION TIME BREAKDOWN

We have seen several GNN accelerator papers talking about the runtime breakdown [42, 43, 44]. Based on the breakdown, people propose methods to accelerate the most time-consuming part. However, they only focused on the inference phase including aggregation and update but ignore the sampling phase and backward propagation phase. Instead, we measure time spent on all the phases in an end-to-end training epoch of GNN.

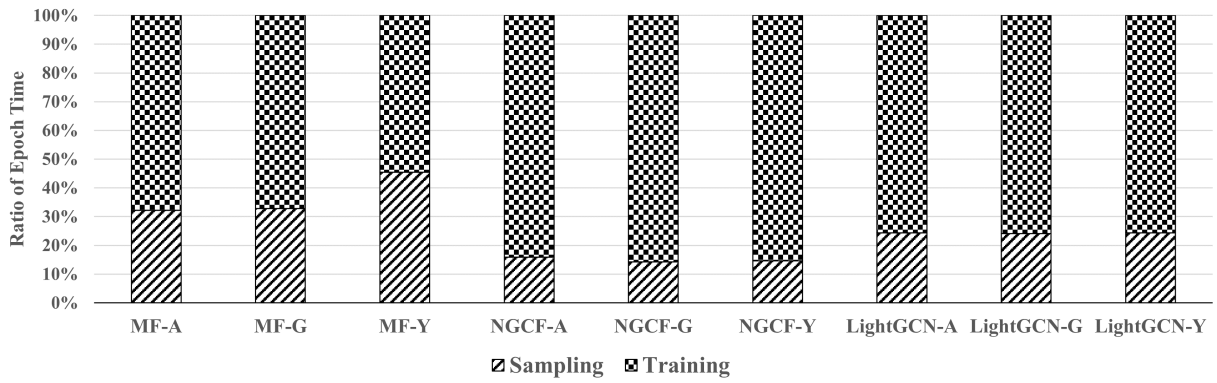

Figure 4.1: Breakdown of Epoch Time

We first present the runtime breakdown between training and sampling in Fig 4.1. *Training* only includes inference and backward propagation not the whole epoch. Figure 4.1 shows the breakdown of three models on three datasets: MF/NGCF/LightGCN-G/Y/A, where G represents the Gowalla dataset, Y represents the Yelp2018, and A represents the Amazon-Book. The runtime is normalized to the whole epoch time and we get the number by averaging 500 epochs after skipping the first 50 epochs. In MF, sampling phase takes up more than 30% of the epoch and can be up to 45.5% on Yelp2018. NGCF spends on average 14% of the time on sampling and it is about 24% for LightGCN. Absolute time spent on the sampling is the same among different models and the breakdown difference comes from the

difference in real training part. MF has much simpler operations than NGCF and LightGCN which consumes less time and leads to the sampling part increase. As we explained in the section 3.4, LightGCN cuts off several operations from the NGCF and that's why portion of sampling time becomes higher. The sampling time on each dataset increases with the graph size. Sampling time on amazon-book is almost three times the time on Gowalla which has similar density. The sparsity in the graph also influences the sampling time as irregular memory access would take more time than regular ones.

To validate our observation, we also did the runtime analysis with GraphSage [16] from Deep Learning Graph (DGL) [23] on a larger graph: opbn-products (approximate 2 million nodes and 61 million edges)from Open Graph Benchmark (OGB) [17]. Ogbn-products is similar to amazon-book dataset but includes more kind of items and more users. We give the detailed formulation for GraphSage in 2.2. The sampling time takes more than 60% in the whole epoch.

Key Takeaway 1: Sampling phase's contribution to the execution time needs more attention and the need for optimization on sampling becomes more severe with larger graph dataset.

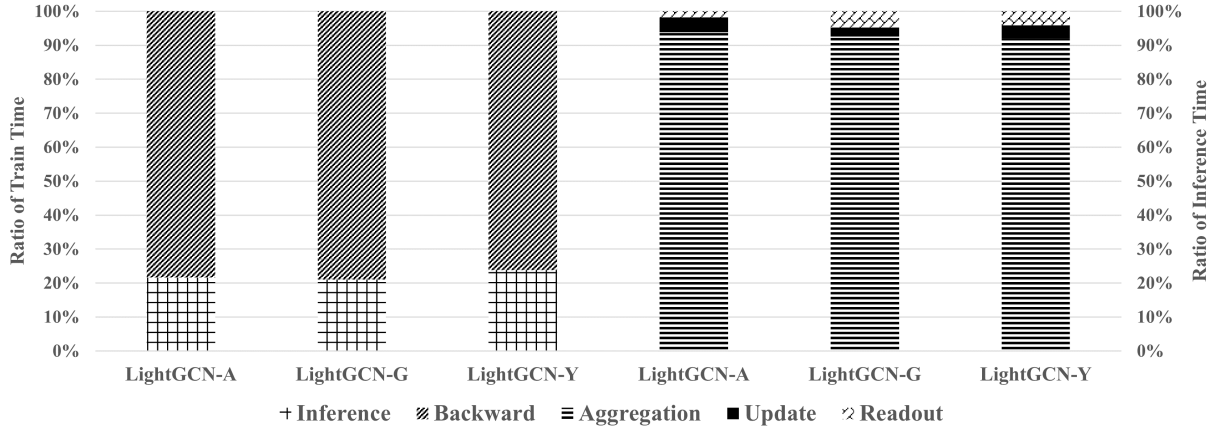

Figure 4.2: Breakdown of Training and Inference Time

We then look into the *Training* part of LightGCN and show the time breakdown among main phases in the Fig 4.2. The left three columns in 4.2 shows the time breakdown between inference and backward propagation and are normalized to the total training time. Still the *Training* excludes the sampling phase. The right three columns show the time breakdown of the aggregation, update and readout phase and they are normalized to the inference time shown in the right axis.

The inference contributes to $21.05\%, 23.81\%$ and 21.68% of the total training time and backward propagation occupies around 80%. The ratio between inference and backward

propagation in LightGCN is slightly lower than it in the deep neural networks (DNNs), shown in the SparseTrain [45]. In SparseTrain, authors divided the backward propagation to backward propagation by input(BWI) and backward propagation by weights (BWW). The sum of BWW and BWI contributes to 75% of execution time on VGG16 [46], 62% on ResNet[47] and 65% on Fixup [48]. Due to the LightGCN's removal of feature transformation from NGCF, the only trainable parameters of LightGCN are only the embeddings of the $0-th$ layer. The backward propagation in LightGCN needs to calculate and pass the gradients back to each node, which is chosen randomly at the sampling time, and update the embedding. It is not hard to imagine that with more trainable parameters, backward propagation would occupy more training time. We measure the backward propagation time and inference time of NGCF on Gowalla. The results shows backward propagation occupies 88.79% of the total training time while the number in LightGCN is 78.98%.

Key Takeaway 2: Similar to DNNs, backward propagation GNNs in recommender systems dominates the execution time in training, while most of the accelerator work is only for inference. Speedup on backward propagation will speedup the whole training significantly.

We have analyzed the execution time breakdown from two levels: sampling and training, inference and backward. In this part, we dive into the inference part which consists of aggregation, update and readout phases. The three columns on the right hand of Fig 4.2 show the execution time breakdown of inference phase of LightGCN on datasets.

Aggregation phase constitutes $92.6\%, 92.1\%, 93.9\%$ of the inference time on Gowalla, Yelp2018 and Amazon-Book dataset respectively. Update phases only occupy $2.7\%, 3.9\%$ and 4.3% in three experiments and remained part are for readout function. Since the aggregation performs a simple reduction for each vertex after gathering its neighbors' feature vectors, it is memory-intensive.

Different from aggregation phase, update phase and readout phase are supposed to be computation-intensive. However, in the LightGCN, the portion of the computation is reduced significantly compared with NGCF. The reduction comes from the LightGCN's simplification. LightGCN only sums the feature vectors from its neighbors in update phase while NGCF performs feature transformation and activation. Similarly, readout function takes up less time due to the smaller final embedding output and computation is reduced.Readout function's contribution to execution time is less than 5% in LightGCN. But even in NGCF, readout phase only takes 7.8% of inference time on Amazon-Book dataset.It is because readout function only executes once per iteration. With larger graphs, overhead from readout function is more negligible if the graph size increases.

Key Takeaway 3: Aggregation dominates the GNN's inference in recommendation and leave room for further improvement. Optimizations on update phase could contribute to

limited speedup. And unfortunately, readout phase is not a good optimization target.

4.2 MICROARCHITECTURE ANALYSIS

In this part, we use Intel Vtune for microarchitectural analysis on the GNN models and locate several main bottlenecks based on the Intel Vtune results. In particular, we emphasize on analysis of the LightGCN model which has the core functions in most GNNs. More detailed reasons are given in section 3.4.1.

4.2.1 Memory-Bounded

Intel Vtune classifies the pipeline slots into four classes: retired, front-end bound, bad speculation and back-end bound. Front-end bound are divided into front-end latency bound and bandwidth bound.Back-end bound slots are divided into memory bound and core bound. Figure 4.3 is breakdown of the pipeline slots either doing useful work or stalled due to different bottlenecks during a full-batch training of LightGCN on datasets.

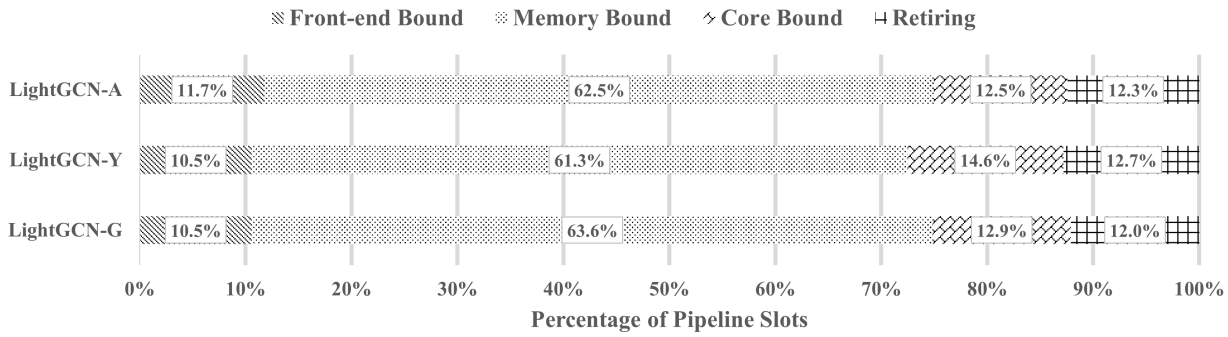

Figure 4.3: Breakdown of Pipeline Slots in end-to-end Training

The breakdown shows that only 12.3% of the pipeline slots attribute to useful work on three datasets on average. 11% of pipeline slots are identified as front-end bound.Front-End bound represents a slots fraction where the processor's Front-End undersupplies its Back-End. Front-end is responsible for fetching instructions and decode them. Front-End bound denotes unutilized issue-slots when there is no Back-End stall.Instruction-cache misses would be one possible reason for front-end bound.

About 13% pf pipeline slots are core bound. It may indicate the machine ran out of an Out-of-Order resources like shared ALU, certain execution units are overloaded or dependencies in program's dataflow are limiting the performance. In the context of LightGCN,

the core bound slots are most likely due to the unresolved dependencies. In the aggregation phase, every central node cannot update its feature vector until it receives the information from all the selected neighbors. The nature of aggregation builds up an explicit dataflow dependencies.

The memory sub-system is the most severe bottleneck. More than 60% of the pipeline slots are stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation. The profiling result is consistent with our runtime breakdown. Aggregation phase, which takes more than 80% of execution time, gathers the feature vectors from the neighbors to the central nodes. Different from the DNN inputs data like image and video, graph data is arranged in memory more irregularly and sparsity is higher, above 99% in our benchmarks. The irregular memory access is more expensive than the regular ones and more pipeline slots could be stalled and wait for the memory response. We further investigate the memory bound slots to see the reason why the GNN is severely memory-bounded.

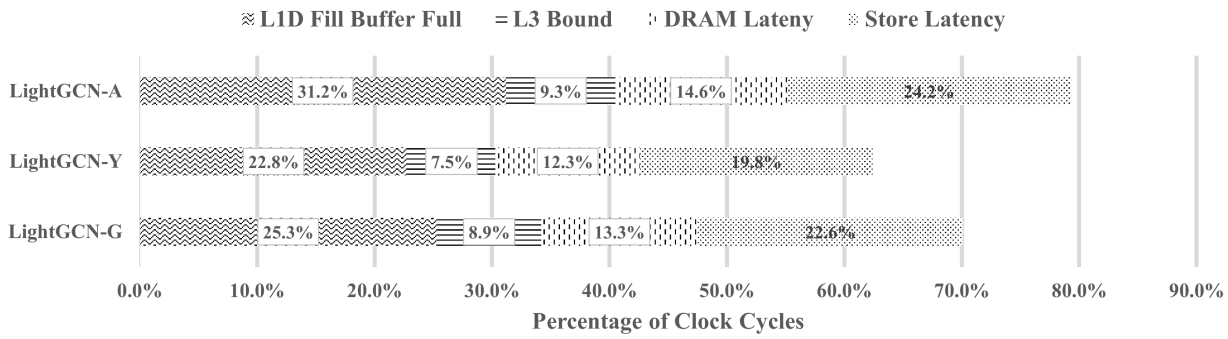

Figure 4.4: Percentage of Clock Cycles stalled due to four reasons

We select clock cycles as metrics instead of pipeline slots for the following analysis. Analysis on clock cycles and analysis on pipeline slots are slightly different.On each cycle, there may be stalls due to different reasons, which means that metrics measured in clock cycles may overlap. For example, two slots of five are wasted on each cycle, which means 40% waste in terms of pipeline slots but it is 100% in terms of clock cycles as there are wasted bubbles on every cycle. The clock cycle analysis is less precise, but it is still useful for identifying the dominant performance bottleneck in the code. [49]. We select four metrics which wastes the most clock cycles within memory bound slots for further analysis: L1D Fill Buffer Full, L3 Bound, DRAM Latency and Store Latency. Their contribution to stalled cycles are shown in the Fig 4.4.

L1D Fill Buffer Full indicates how often the L1 data cache fill-buffer is full. Unavailability

of the L1D fill buffer limits additional L1D miss memory access requests to proceed. Fill-buffers hold the lines coming from the lower level memories and release them to RAMs either when the fill-buffer is full or when there is an opportunity where the cache is not receiving requests from the core pipeline. In Skylake, there are 10 entries for fill-buffer [50]. The higher the metric value, the deeper the memory hierarchy level the misses are satisfied from. In LightGCN, unavailable L1 data cache fill-buffer is one the biggest reasons leading to the cycle stalls. 31.2% cycles are stall due to the full fill buffer in LightGCN on Amazon-Book. The higher the metric value, the deeper the memory hierarchy level the misses are satisfied from. In this case, it hints that L1 misses are often satisfied from deep in the memory hierarchy, such as from DRAM. Meanwhile, it also hits on approaching bandwidth limits to lower level of memory like DRAM. Therefore, reducing DRAM bandwidth pressure is good direction to optimize GNN workloads.

L3 Bound metric shows how often CPU was stalled on L3 cache. Processor would fetch the cache lines from DRAM to L3 and extract useful data/instruction. However, due to the high sparsity in graph and irregular memory location of nodes, only one node in a cache line is required in most cases which waste the cache space a lot. Besides, the node-wise sampling easily leads to neighbor explosion and it hurts the cache performance. In three experiments, about 10% of stalled cycles are due to L3 cache miss.

DRAM latency metric represents a fraction of cycles during which programs are stalled due to the latency of the main memory (DRAM). 14.6%, 12.3% and 13.3% of cycles are stalled due to DRAM latency for LightGCN on Amazon-book, Yelp and Gowalla dataset. This observation aligns with the full fill-buffers. As the fill-buffer is always full, more request is sent to DRAM and more and more of them cannot be satisfied in time with the original bandwidth. DRAM latency can be alleviated by optimizing data layout, increasing the data locality and reducing DRAM bandwidth requirement.

Key Takeaway 4: GNNs are heavily memory-bounded in recommender systems due to the memory-intensive aggregation phase. GNNs approach the bandwidth limits to DRAM and therefore, reducing the DRAM bandwidth requirement is a good direction to accelerate the GNN process.

Store latency is another biggest reason for stalled cycles contributing to 24.2%,19.8% and 22.6% on experiments. Typically, store access do not stall the pipeline on Out-of-Order CPUs with advanced design. Store latency metric represents cycles fraction the CPU spent handling long-latency store misses (missing 2nd level cache). Stores are mainly in the update phase and aggregation phase always sends more reading request than the writing request from update. The long store latency could rise from two places: full-buffer and DRAM. In case of the data cache, the write-policies followed are write-back and write-allocate. The fill-buffers

in the data-cache also hold the lines on which stores need to be performed. Unavailable fill-buffers hurt the store performance. DRAM suffers from the huge amount of request for reading data with limited bandwidth. The store request will wait for a long time until DRAM is free.

In addition to these four metrics, contested accesses also contribute 2.5% of stalled cycles in LightGCN runs. Contested accesses occur when data written by one thread is read by another thread on a different core. This metric is a ratio of cycles generated while the caching system was handling contested accesses to all cycles. Common examples of contested accesses include synchronizations, true data sharing and false sharing. Alternative aggregation and update phase in GNN imply the implicit synchronizations. For example, during the aggregation phase, the central node cannot do any computation until all the neighbors have sent the information to it. This intra-layer dependencies imply the need for synchronization. Fortunately, we can remove the synchronization if the update function's result are independent of the order neighbor's feature vectors arrive at the central node. Sum function is a good example. But if the update function requires the matrix-matrix multiplication then we have to wait for all neighbors. Similarly, inter-layer dependencies require the synchronization.For example, after aggregation through one layer, the nodes need to update its feature vector and then it can send the new feature vector to its neighbors for the next layer. The inter-layer dependencies guarantee the result correctness and next layer always receives the latest data from previous layer. This insight is also verified by another detailed Vtune report about time consumption on functions.KMP_{wait} function from Intel OpenMP takes a significant portion of CPU time which indicates the threads spend a lot of time waiting for each other.

Key Takeaway 5: Intra-layer and Inter-layer dependencies in GNNs greatly limit the parallelism we can explore. Relaxing the dependencies by changing the model training algorithm may speedup the process.

4.2.2 Workload Imbalance

We notice the workload imbalance problem when checking the core utilization metric on the datasets which are 50.59%, 66.9% and 44.4% on Amazon-book, Gowalla and Yelp2018. Load imbalance is one of the most possible reasons for pool physical CPU utilization. We collect the CPU utilization information for LightGCN on one epoch and generate the utilization histograms shown in Fig 4.5, 4.6 and 4.7.

The effective CPU utilization histograms display a percentage of the wall time the specific number of CPUs were running simultaneously. For example, in Fig 4.5 when LightGCN

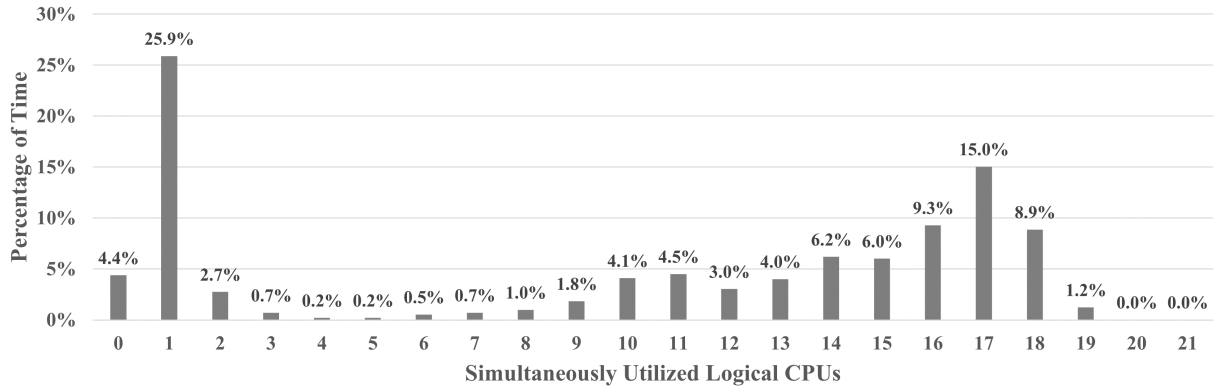

Figure 4.5: Effective CPU Utilization Histogram for LightGCN on Amazon-Book dataset

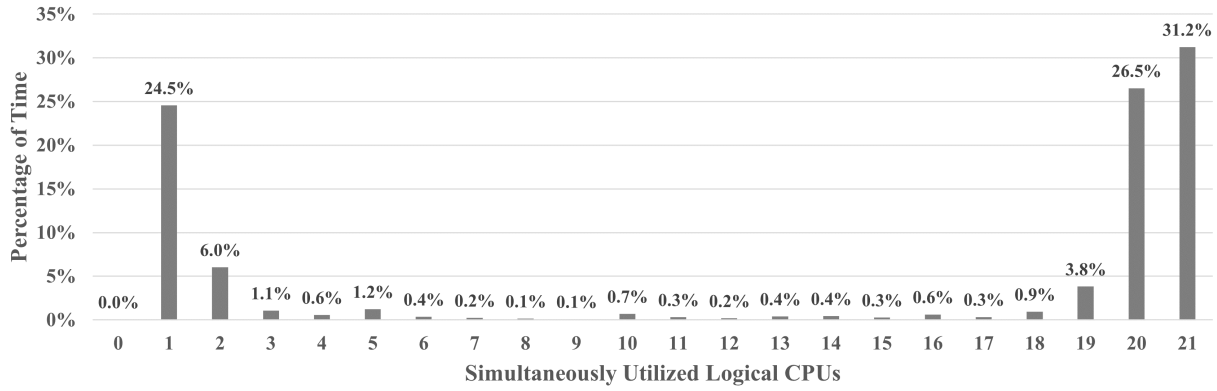

Figure 4.6: Effective CPU Utilization Histogram for LightGCN on Gowalla dataset

runs on Amazon-Book dataset, there are only one cores running for 25.9% of time and there are 1.2% of total execution time when 19 cores are running in parallel. Although, we have 22 cores in total but there is no time period when all the 22 CPUs run together.

LightGCN shows different utilization histograms on three datasets.When running on the Gowalla dataset, there are either 1 cores or more than 20 cores running at the most time. And there is no stall during the whole epoch. While in Yelp and Amazon-Book, we see there is no core running for time to time. In Yelp, the cores are idle for around 10% of time. The pattern on Gowalla is easier to understand. The CPU which is responsible to master thread is always running and when the aggregation and update phase come, all the CPUs start working on the job. We got a snapshot for the thread working status for LightGCN on Gowalla in Fig 4.8. The left column shows the thread number and the one highlighted with blue box is the master thread. The green bar means the CPU is running and brown bar means the CPU is working on jobs. Master thread always have the brown bar and the other threads can wait for an interval and then run for jobs. Because we profile the whole

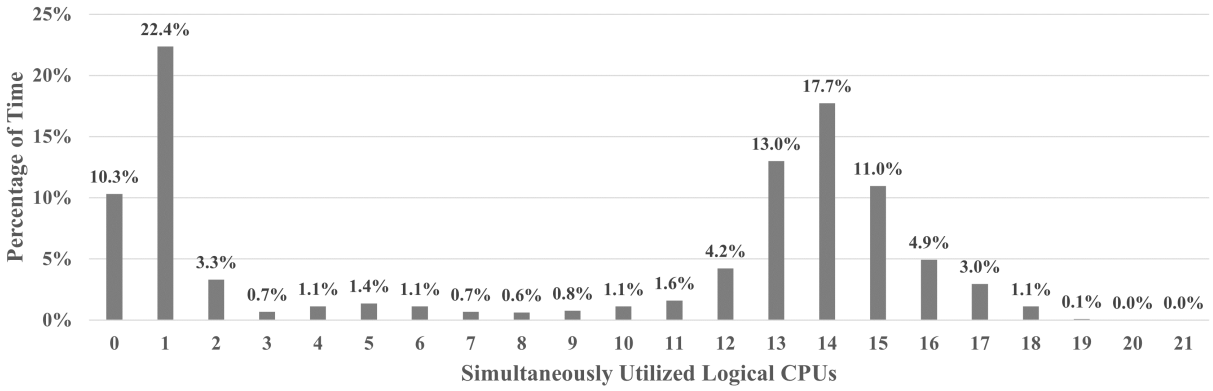

Figure 4.7: Effective CPU Utilization Histogram for LightGCN on Yelp dataset

epoch instead of the training phase only, there could be only one or two CPUs running to handle the operations outside GNN main program. This is associated with the specific implementation.

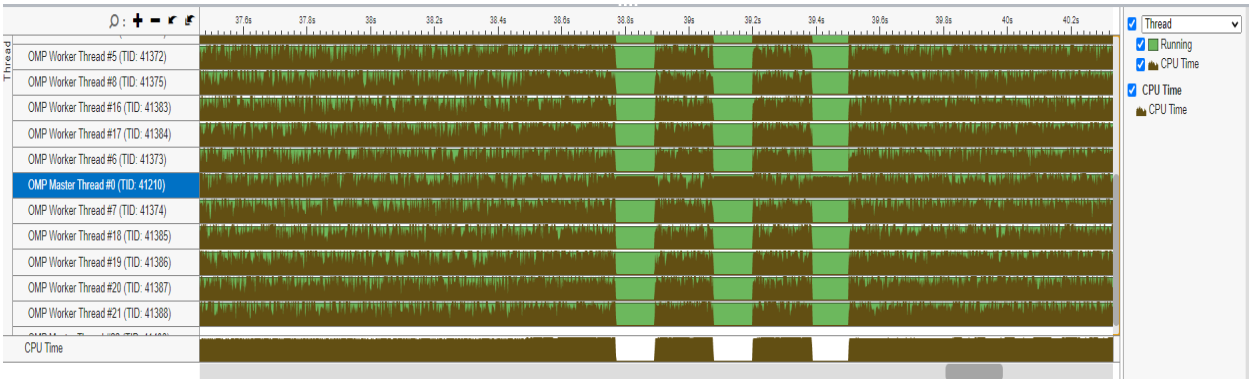

Figure 4.8: Thread Working Status at Runtime

LightGCN has lower CPU utilization when running on Amazon-Book and Yelp datasets compared with the Gowalla. Bi-modal patterns are shown in both histograms. One modal is at the one CPUs while the modal is at 17 and 14 CPUs respectively. The shift of modal indicates that some CPUs are idle while its 'colleagues' are busy with GNN tasks. For example, the modal on 14 CPUs can be interpreted as 14 CPUs are busy with work and the left 8 CPUs have done their job and wait or even there is no job for them to work on.

Graphs following power-law distribution is one of the reasons for workload imbalance [9]. Power-law distribution states that the number of nodes y of a given degree x is proportional to $x^{-\beta}$for a constant $\beta > 0$. When we present the graph by the adjacency matrix, non-zeros can be clustered or appear in only a few rows. Besides, based on the table 3.1, we know the sparsity of the three datasets is extremely high ($> 99.9\%$). The non-zero elements are highly

scattered, making it challenging to access enough non-zero data to feed the CPUs, therefore only part of the CPUs will work at the same time. We checked whether the Amazon-Book, Gowalla and Yelp2018 follow the graph law by counting the non-zero elements in each row. We find that 1% of rows already have more than 97% of non-zero elements in the adjacency matrix which hints the power-law distribution.

Key Takeaway 6: Workload imbalance issue emerges in the GNN models due to power-law distribution in the graph. The imbalance limits the parallelism and underutilizes the CPUs. Intelligent workload scheduling is in demand.

4.2.3 Effects of Environment Configuration on GNN

In this part, we changes several configurations in the platform to see their influence on the GNN's performance.

Varying the number of threads: By default, the SMT is disabled and we enable SMT to try different number of threads among 1 and 44. We limit the thread number to 44 because we only have 22 cores in one socket with 2 way SMT. Further increase will involve the inter-socket communication. We only tried LightGCN on Gowalla and measure the epoch time. The ratios of epoch time with different number of threads over epoch time with one thread are shown in Fig 4.9.

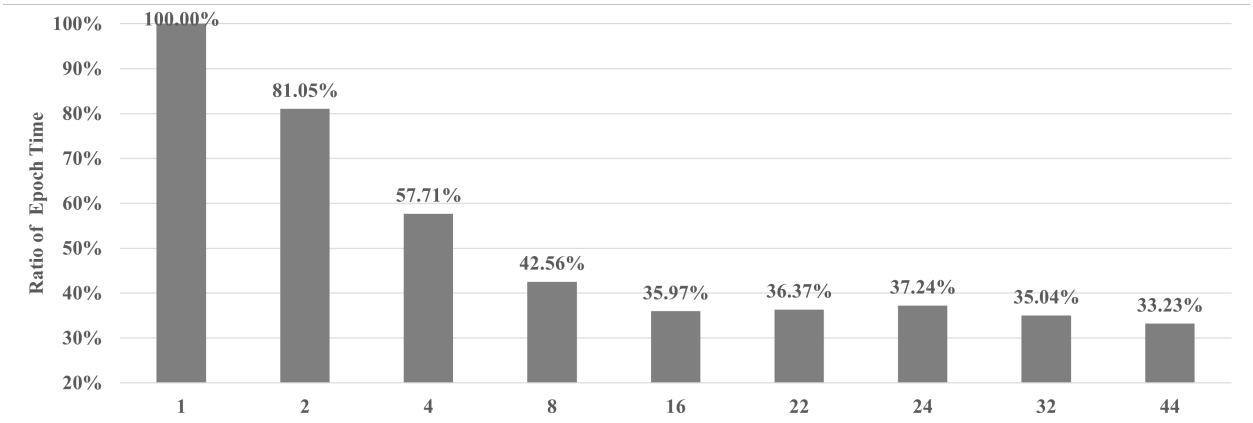

Figure 4.9: Epoch Time with different number of threads

The lower, the faster GNN finishes. We run the LightGCN for 500 epochs and get the average number. The execution time decreases notably when the thread number increases from 1 to 16. And after more than 16 threads, the increase in threads cannot bring much speedup and the diminishing marginal returns imply the parallelism in GNN are already explored and is not bottlenecked by computation power anymore. It is consistent with our

observation that GNN is memory-bounded.

Varying Intel OpenMP scheduling Policy: By default, OpenMP statically assigns loop iterations to threads. When the *parallel for* block is entered, it assigns each thread the set of loop iterations it is to execute. We find the loops in the negative sampling code and tried dynamic and guided scheduling. However, the time changes are quite small as we only change several loops manually and didn't touch the Pytorch kernels.

Using Intel OpenMP Thread Affinity for Pinning: The Intel Compiler's OpenMP runtime library has ability to bind OpenMP threads to physical CPUs. Processor affinity takes advantage of the fact that remnants of a process that was run on a given processor may remain in that processor's state like data in cache. We select affinity and try to improve data locality. The epoch time decreased by 26.3% running LightGCN on Gowalla and decreased by 19.6% on Amazon-Books. The speedup indicates that locality optimization is a promising way to accelerate GNNs.

CHAPTER 5: POTENTIAL OPTIMIZATIONS FOR GNN

Based on the key takeaways that we have summarized during the analysis, we propose several ideas to explore for the future. Reducing the DRAM bandwidth requirement, improving the data locality and break the dependencies are three main start points.

5.1 HARDWARE-FRIENDLY SAMPLER

Training a GNN model for large-scale graphs requires high computation and memory costs. Motivated by the urgent need in terms of efficiency and scalability, sampling methods are utlized and achieve a significant effect. Two major types of sampling are node-wise sampling (used in LightGCN) and layer-wise sampling. Generally, the common ground between node-wise sampling methods is that they perform the sampling process on each node and sample neighbors based on specific probability. Most layer-wise sampling methods are proposed to alleviate neighbor explosion caused by recursive neighbor sampling. Layer-wise sampling method samples a certain number of nodes together in one sampling step. Time consumption of the layer-wise sampling method is significantly reduced by avoiding the exponential extension of neighbors. We focus on the node-wise sampling for now.

A few works have been proposed to accelerate GNN based on the specific hardware design like [42, 9]. However, most of them only accelerates the inference process leveraging hardware characteristics ignoring that the sampling phase gradually becomes a time-consuming process with the drastic extension of graph data. Our profiling also shows the high overhead on the sampling process. Based on these insights, we propose to accelerate the sampling phase utilizing the hardware characteristics.

To accelerate the sampling phase, we can increase the cache hit ratio and utilize the existing prefetch techniques to reduce the expensive memory accesses. For example, the vertex v_0 choose its neighbor n_1 during sampling and n_1 is fetched to cache. Ideally, it saves the memory access if the next vertex v_1 also chooses n_1 to sample assuming v_0 and v_1 have the mutual neighbor n_1. In reality, two vertices which execute and do the sampling consecutively may not have the mutual neighbors. The sampling algorithm would reorder the sampling on vertices, choose two vertices who share a larger set of neighbors and run sampling for them consecutively. The goal is to maximize the overlap on neighbor sets of two consecutive vertices in the whole sampling phase with the knowledge of adjacency matrix. The order change on sampling phase may also influence aggregation and update phase. For example, assuming we a graph with four nodes v_1, v_2, v_3, v_4 and run a GNN on the small

graph. Previously, we sample neighbors for them in the v_1, v_2, v_3, v_4 order and execute the aggregation/update in the same order.But now as we know v_1 and v_4 have more mutual neighbors than v_1 and v_2, we will sample neighbors for them in the v_1, v_4, v_3, v_2 order and execute the aggregation/update in the corresponding sequence. A good side-effect is that this intelligent sampling also help the training because the v_1, v_4 works on the same neighbors and aggregate information from them. In high-level, this hardware-friendly sampling algorithm utilizes the knowledge about adjacency matrix which is fixed after a graph is given to improve the data locality at runtime.

5.2 RUNTIME WORKLOAD BALANCE

Workload imbalance hinders the further parallelism exploration in GNNs. AWB-GCN [9] proposed three hardware-based autotuning techniques to alleviate the workload imbalance: dynamic distribution smoothing, remote switching, and row mapping. However, these techniques are operated at row-level and AWB-GCN aims at the ASIC instead of CPUs. Instead of row-level, thread-level scheduling is much easier and the monitor cost would be lower. The idea is that creating a thread runtime status monitor to check the status and move the current work or dispatch new GNN work to it. However, the thread migration has cost and one possible way to reduce the cost is the migration is done between two cores which read in the similar data. For example, we can trace which nodes' feature vector has been read by the threads and the CPU. Two threads which have similar trace can move between the cores.

5.3 ASYNCHRONOUS GNN

Inter-layer and Intra-layer dependencies creates the implicit synchronization for GNNs which takes a lot of time. To break the inter-layer dependencies, one possible idea is to relax the dependencies and allow the head thread to proceed with old data. The central node can update its features in layer $k+1$ with the old feature vector (from layer $k-1$ instead of k) from its neighbor. But this asynchronous way could lead to incorrect results or failed convergence of the GNN model. To guarantee the correctness, neighbor may send out the changes (δ) in its feature vector to the central node and central node update its value again. The potential benefit is that before δ is sent, fast thread can switch to another task. Another idea is to utilize graph clustering [51] to generate subgraphs. The GNN runs on each subgraph and communicate the weight update information with others. As long as

the subgraphs are disconnected, the inter-layer dependencies only occurs within the cluster. And each GNN on each cluster can run at their own speed.Running on a single subgraph also benefits the data reuse and improve performance. This idea is similar to GNN under a huge distributed system [52].

CHAPTER 6: RELATED WORK

Recommender systems have become indispensable tools of many business like product suggestion on e-commerce website. Deep learning based recommendation models are broadly used throughout industry and academia [21]. Among all the deep learning based models, GNNs become more and more attractive because of its superior ability in learning on graph-structured data, which is fundamental for recommender systems [7, 53].

Due to its growing popularity and broad application cases, GNNs have also attracted attention from computer architecture community. Multiple customized systems are designed for GNN workloads. HyGCN [8] is built based on the insight that GNN's two alternating phases show significantly different computation needs and thus uses separate engines for the aggregation and update stages. Additionally, HyGCN manages the pipelined execution of aggregation and update with an inter-phase coordinator. The update engine utilizes a conventional systolic array to accommodate the huge computation demand, and the aggregation engine has an architecture to handle the irregular accesses with window sliding and shrinking. HyGCN is limited to GCN, and is not generalized to other types of GNN. EnGN [54], inspired by CNN accelerators, treats a GNN as a concatenated matrix multiplication of feature vectors, adjacency matrices, and weights. With a single dataflow, EnGN is generalizable to many GNN variants. AWB-GCN[9] is motivated by the power-law distribution of most graphs, which means that some parts of the computation are dense and some are extremely sparse, which leads to the imbalance among process engines (PEs). AWB-GCN alleviates the workload imbalance via three balancing algorithms: distribution smoothing, remote switching and evil row remapping. The balancing algorithm is chosen at run-time based on the sparsity and PE's status. GRIP [55] leverages the abstraction of GReTA [56] to develop a general accelerator for any GNN variant. The hardware implementation in GRIP is similar to HyGCN, with specific hardware for vertex-centric and edge-centric computation. These accelerators focused on the inference, ignoring the training process and they did the simple work on characterizing GNN inference and aims at the accelerator design.

Yan et al.[57] characterized the several GCN models on popular benchmarks with NVIDIA GPU V100, to understand the computation and memory accessing pattern of GCNs. They profiled and analyzed the inference stage only. Zhang et al.[58] have also characterized the inference performance of GNNs. They constructed a GNN benchmark based on the extensive model review and a general GNN description framework which decomposes GNN inference execution into a Scatter-ApplyEdge-Gather-ApplyVertex (SAGA) pipeline and then analyzes the behavior of each phase. This work is also for GPU. In contrast, our work characterize

the state-of-the-art GNN models in the recommender systems and full training process is profiled. Compared with the simple characterization in the accelerator work, we did detailed microarchitectural analysis and figure the bottlenecks more accurately.

CHAPTER 7: CONCLUSION

This thesis presents our study on characterizing the graph neural networks (GNNs) in the recommender systems. We utilize profiling tools to conduct both high-level (execution time) and microarchitectural analysis on GNN models with representative recommendation workloads. Our work is the first one focusing on the CPU platform and covering the whole training process of the GNN. Based on the detailed and valid characterization, we summarized several valuable insights which are good guidance to develop GNN models or hardware systems for GNNs.

REFERENCES

[1] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[2] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[3] R. M. Bell and Y. Koren, "Scalable collaborative filtering with jointly derived neighborhood interpolation weights," in *Seventh IEEE international conference on data mining (ICDM 2007)*. IEEE, 2007, pp. 43–52.

[4] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An algorithmic framework for performing collaborative filtering," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, 1999, pp. 230–237.

[5] Y. Koren and R. Bell, "Advances in collaborative filtering," *Recommender systems handbook*, pp. 77–118, 2015.

[6] C. Shi and S. Y. Philip, *Heterogeneous information network analysis and applications*. Springer, 2017.

[7] 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*, 2018, pp. 974–983.

[8] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 15–29.

[9] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt et al., "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[10] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE transactions on knowledge and data engineering*, vol. 17, no. 6, pp. 734–749, 2005.

[11] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-based systems*, vol. 46, pp. 109–132, 2013.

[12] J. Lu, D. Wu, M. Mao, W. Wang, and G. Zhang, "Recommender system application developments: a survey," *Decision Support Systems*, vol. 74, pp. 12–32, 2015.

[13] S. Wu, F. Sun, W. Zhang, and B. Cui, "Graph neural networks in recommender systems: a survey," *arXiv preprint arXiv:2011.02260*, 2020.

[14] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[15] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng et al., "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[16] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[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," *arXiv preprint arXiv:2005.00687*, 2020.

[18] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," *arXiv preprint arXiv:1809.05343*, 2018.

[19] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[21] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[22] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[23] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma et al., "Deep graph library: Towards efficient and scalable deep learning on graphs." 2019.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[26] L. Chen, L. Wu, R. Hong, K. Zhang, and M. Wang, "Revisiting graph based collaborative filtering: A linear residual graph convolutional network approach," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 27–34.

[27] M. Zhang and Y. Chen, "Inductive matrix completion based on graph neural networks," *arXiv preprint arXiv:1904.12058*, 2019.

[28] X. Wang, R. Wang, C. Shi, G. Song, and Q. Li, "Multi-component graph convolutional collaborative filtering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 6267–6274.

[29] J. Xu, Z. Zhu, J. Zhao, X. Liu, M. Shan, and J. Guo, "Gemini: A novel and universal heterogeneous graph information fusing framework for online recommendations," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3356–3365.

[30] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, 2019, pp. 417–426.

[31] "Introducing pytorch profiler - the new and improved performance tool," 2021. [Online]. Available: https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/

[32] "Intel vtune profiler: Quickly find and fix performance bottlenecks and realize all the value of your hardware," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[33] D. Liang, L. Charlin, J. McInerney, and D. M. Blei, "Modeling user exposure in recommendation," in *Proceedings of the 25th international conference on World Wide Web*, 2016, pp. 951–961.

[34] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua, "Neural graph collaborative filtering," in *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*, 2019, pp. 165–174.

[35] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *proceedings of the 25th international conference on world wide web*, 2016, pp. 507–517.

[36] R. He and J. McAuley, "Vbpr: visual bayesian personalized ranking from implicit feedback," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[37] X. Wang, X. He, M. Wang, F. Feng, and T. Chua, "Neural graph collaborative filtering," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2019, Paris, France, July 21-25, 2019.*, 2019, pp. 165–174.

[38] 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*, 2020, pp. 639–648.

[39] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," *arXiv preprint arXiv:1205.2618*, 2012.

[40] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[41] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[42] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[43] R. Garg, E. Qin, F. M. Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam et al., "A taxonomy for classification and comparison of dataflows for gnn accelerators," *arXiv preprint arXiv:2103.07977*, 2021.

[44] 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*, 2020, pp. 401–415.

[45] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Sparsetrain: Leveraging dynamic sparsity in software for training dnns on general-purpose simd processors," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 279–292.

[46] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[48] H. Zhang, Y. N. Dauphin, and T. Ma, "Fixup initialization: Residual learning without normalization," *arXiv preprint arXiv:1901.09321*, 2019.

[49] "Intel vtune profiler user guide," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html

[50] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[51] S. E. Schaeffer, "Graph clustering," *Computer science review*, vol. 1, no. 1, pp. 27–64, 2007.

[52] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: Scalable distributed training for large-scale graph neural networks," *arXiv preprint arXiv:2104.06700*, 2021.

[53] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.

[54] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, 2020.

[55] K. Kiningham, C. Re, and P. Levis, "Grip: a graph neural network accelerator architecture," *arXiv preprint arXiv:2007.13828*, 2020.

[56] K. Kiningham, P. Levis, and C. Ré, "Greta: Hardware optimized graph processing for gnns," in *Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*, 2020.

[57] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.

[58] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer architecture letters*, vol. 19, no. 1, pp. 59–62, 2020.