Scaling Graph Neural Networks with Persistent Memory

Bingyu Chen

Master of Science
School of Informatics
University of Edinburgh
2022

Abstract

There has been an explosion in the number of graph neural networks (GNNs) in recent years. However, applications frequently require to load all of the graph data into DRAM all at once, which restricts the amount of graph data they are able to manage. The new byte-addressed persistent memory (PM), which is similar to DRAM except that the data is stored for an extended period of time, will eventually become cheaper than DRAM for the same capacity but will be somewhat slower than DRAM. In order to provide an answer to the question ”Can PM devices be alternative of DRAM to process large dataset/model in GNNs ?”this study will measure the scalability benefits and performance losses from employing PM in GNN training and inference by working on datasets of different sizes (mainly pull-scale datasets).

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Bingyu Chen*)

Acknowledgements

I would like to thank my supervisors, Honda Michio and Shangmin Guo for their input of time, advice and help. Their feedback and insights were indispensable and without them I would not have been able to complete my project successfully. I would also like to thank my family and girlfriend for their support and love during this period.

Table of Contents

Chapter 1

Introduction

With the development of deep learning, tasks in machine learning have shifted from image classification and video processing to speech recognition and natural language interpretation. Typically, the data for these activities is represented in Euclidean space. However, the real world is complex, such as social networks, where the number of one's friends is not fixed and difficult to put in order. Such complex non-Euclidean data (non-Euclidean), with no top and bottom, no order, no coordinate reference points, and difficult to represent in square (grid-like) matrices/tensor, have been done before to fit irregular feet (non-Euclidean data) into standard shoes (neural networks), with not very good results, so the question becomes: can a new shoe be designed so that it can fit irregular feet? [1, 2]

Therefore, as technology advances, there are an increasing number of applications in which data is created from non-Euclidean domains and represented as graphs with complicated interactions and interdependencies among objects. And the graph neural network was born. Graph neural networks (GNNs) are neural networks for graph data created to solve the problem that traditional machine learning can only process data in Euclidean space. Recurrent GNNs, convolutional GNNs, graph autoencoders, and spatial-temporal GNNs are the four subcategories of currently available GNNs[2].

GNNs are currently used in a variety of tasks and fields with great success. Machine vision, natural language processing, recommendation systems, and other applications are examples. However, large-scale GCN training remains a difficult problem. The computer requires a large amount of memory to store the entire graph data. The graph data that will be used in GNN models, including the structure of the graph, information about the nodes, and so on[3]. This kind of initial data is primarily stored on the hard-drive. Storing this large data is not too much of a problem for the storage density of the hard-drive. However, the current technical challenge is that before each GNNs training, the computer must load it from the hard-drive into memory; the capacity of which is too small in comparison to the large amount of graph data. This presents a challenge because the hard-drive is the primary storage location for the initial data. For the purpose of ConvGNN [4], for instance, it is frequently required to store all of the graph data as well as the intermediate states of all of the nodes in memory. Memory overflow is a particularly problematic issue because of this, especially when a graph has a large number of nodes[2, 5].

Memory overflow resulting from large-scale training has been the subject of a great deal of previous research. ClusterGCN's [3] domain search, for instance, is restricted to subgraphs and is capable of processing massive graph structures with minimal time and memory usage while utilising deeper network topologies. In addition, GraphSage [6] recommends a batch training technique for ConvGNNs. However, each has disadvantages: In their research, ClusterGCN [3] is able to cut space utilisation from 11.2GB to 2.2GB, however for very large scale training, this may not be sufficient to resolve the memory overflow issue. GraphSage [6] trades more training time for reduced space usage, and the complexity of time and memory grows exponentially with the number of network layers.

A new hardware-based solution to the issue of the large-scale training of GNNs has emerged with the growth of non-volatile memory (NVM)[7] applications. NVMs' physical characteristics could make them advantageous. When compared to conventional DRAM technology, they offer a much higher data storage density while using much less power[8]. Therefore, it is crucial to investigate whether Non-Volatile Memory(NVM) could possibly replace DRAM. Once this solution can be implemented (without suffering too much performance loss), it will offer a fundamental solution to memory overflow.

Implementing a PM(persistent memory)-based data loader and using it for the GNNs' data preparation phase is the major objective of this project, given the aforementioned technological realities (used as a medium for storing the adjacency matrix and node feature table of graph data). This type of procedure has previously been implemented by Wei-Lin Chiang et al [5]. As earlier work has provided a full introduction to the use of NVM to GNNs and some evaluations have been conducted, we are in a better position to develop a comparable data loader and test the project's performance on various GNN algorithms. Due to the bandwidth of the present hardware technology, previous research has determined that NVM still has a data transmission latency disadvantage compared to DRAM. However, we continue to have high expectations for its performance on several algorithms.

This dissertation is divided into four main sections. In Chapter 2, we summarise and introduce the multiple GNN algorithms for this project and identify the most suitable algorithm for our project to discuss and analyse. We identify and analyse several mainstream datasets for GNN training and select the most applicable dataset based on our hardware conditions. In Chapter 3, we describe the data loader we designed to access the PM storage medium and the way in which it was combined with the *Pytorch Geomatric*, which was the main work of the project. A baseline for this experiment is also established, and metrics for evaluating PM-based GNNs are also presented, which we will use and evaluate our model against these baselines. In Chapter 4 we present the results of our experiments, providing detailed analysis and comparison with existing related work. Finally, in Chapter 5, we summarise the findings of our experiments and provide guidance for future work.

Chapter 2

Background

2.1 General Understanding of Graph Neural Networks

The development and use of neural networks in recent years have greatly aided the study of pattern recognition and data mining. Target detection, machine translation, and speech recognition are just a few examples of the many machine learning tasks that have benefited greatly from the widespread adoption of end-to-end deep learning algorithms like convolutional neural networks (CNN) [9], long and short-term memory (LSTM) [10], and autoencoders [11].

The major focus of deep learning in recent years has been the extraction of latent data characteristics from Euclidean spatial data (images, text and video). As application scenarios have evolved, we have discovered a huge variety of applications in which the data is graphically displayed. In e-commerce, for instance, graph-based learning systems can mine the interactions between users and products to provide highly accurate recommendations; in social media, they can mine the relationship between users' interests and media content based on the graph of users' attention relationships to provide highly accurate recommendations. These two examples are only two examples of how data may be visually portrayed. Following this, GNN was founded.

Using the architectural concepts of deep learning RNNs, CNNs and AEs, the generalisation and definition of significant operations have advanced significantly during the past several years. Graph convolution, for instance, may be generated from 2D convolution, as seen in 2.1, where the picture can be regarded as a graph of special instances, with individual pixels linked to adjacent pixels. Similar to two-dimensional convolution, graph convolution operations may be done on a image by averaging the information of neighbouring nodes[12].

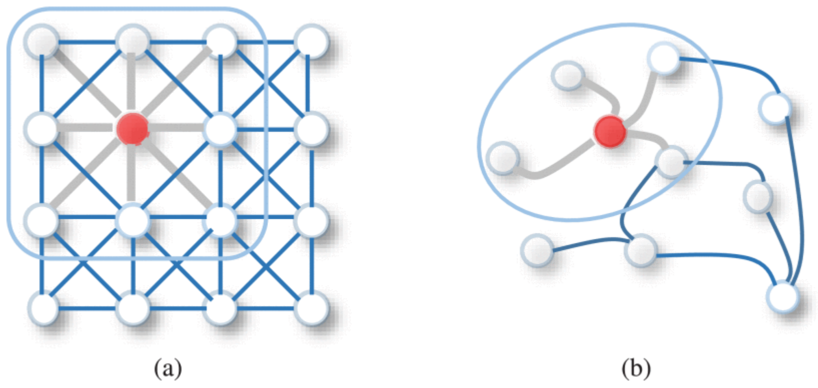

Figure 2.1: two-dimensional convolution(a) vs graph convolution(b) [12]

2.2 Features of DRAM and NVM

DRAM (Dynamic RAM) is the primary computer memory. It maintains data for a relatively little period of time, is slower than SRAM, and is far less expensive than SRAM. This is why DRAM is often used as the primary memory for computing, but SRAM is typically utilised for the CPU's first and second level caches, for instance. This project's baseline solution (using memory as the data storage medium) was implemented utilising DDR4 DRAM.

Non-volatile memory (NVM) refers to non-volatile physical storage media such as NAND flash, PCM, 3D XPoint, and others that may be made in SSD or PM form (or SCM). Flash memory, phase-change memory (PCM), and resistive random-access memory (RAM) are examples of nonvolatile memory (NVM) devices (ReRAM). The projected physical features of these NVMs will determine their potential advantages. As a result, they require very little power and have significantly greater data storage density than DRAM technology. The most important physical property is that data saved in the medium is not lost when power is turned off.[8] Although prone to ambiguity, there is still a substantial amount of contemporary literature that refers to PM or SCM-based NVMs; thus, when reading the literature, we can infer that: NVMs can refer to storage media such as Flash, PCM, and so on; NVMs can also refer to storage devices such as SCM, PM, and so on. In reality, SSDs were formerly referred to as NVMs.

The table below summarises the performance indicators of different storage technologies in a number of papers.

The table 2.1 shows that the latency of NVM read operations is much lower than that of write operations. Even the latency of read operations for PCM-based NVM

Storage	Cell size	Access granularity	Read Lat.	Write Lat.	Erase Lat.	Endurance	Standby Power
HDD	N/A	512B	5ms	5ms	N/A	$>10^{15}$	1W
SLC Flash	4-6 F^2	4KB	25μs	500μs	2ms	10^4-10^5	0
DRAM	6-10 F^2	64B	50ns	50ns	N/A	$>10^{15}$	Refresh Power
PCM	4-12 F^2	64B	50ns	500ns	N/A	10^8-10^9	0
STT-RAM	6-50 F^2	64B	10ns	50ns	N/A	$>10^{15}$	0
ReDRAM	4-10 F^2	64B	10ns	50ns	N/A	10^{11}	0

Table 2.1: Average device-level performance of different storage technologies[13][14][15]

techniques is almost the same as that of DRAM to know. As the memory consuming data structures (i.e. adjacency matrix, feature table) involve read-only operations in the GNN training process. Therefore it is of practical importance to explore the application of NVM in the GNN training process.

2.3 Related work

There have been various algorithmic optimisations for the memory overflow problem in GNNs, some of the more successful ones being GraphSage [6], ClusterGCN [3] and GraphSAINT [16]. In this section we focus on the implementation details of these three algorithms. In addition, at the end we present the latest system-level and hardware-level based solutions currently available, which are the best guide for this project.

2.3.1 GraphSage

GraphSAGE[6] is a graph neural network algorithm proposed in 2017 that addresses the limitations of GCN networks: GCN training requires the entire graph adjacency matrix, which depends on the specific graph structure, and can generally only be used in direct push learning Transductive Learning. graphSAGE uses a multi-layer aggregation function, where each layer aggregates the GraphSAGE uses a multi-layer aggregation function, where each layer aggregates the information of nodes and their neighbors to obtain the feature vectors of the next layer.

The operational flow of GraphSAGE can be divided into three steps:

1. Sampling each vertex neighboring vertices in the graph.

2. Aggregating the information contained in the neighboring vertices according to the aggregation function.

3. Obtaining a vector representation of each vertex in the graph for use in downstream tasks.

$$h^k_{N(v)} \leftarrow AGGREGATE_k(\left\{h^{k-1}_u, \forall u \in N(v)\right\}) \tag{2.1}$$

$$h^k_v \leftarrow \sigma\left(W^k \cdot CONCET(h^{k-1}_v, h^k_{N(v)})\right) \tag{2.2}$$

For the consideration of computational efficiency, a certain number of neighboring vertices are sampled for each vertex as the vertices of the information to be aggregated. The number of vertices sampled is k. If the number of vertex neighbors is less than k, then the sampling method with put-back is used until k vertices are sampled. If the number of vertex neighbors is greater than k, then sampling without put-back is used. That is, a fixed number of neighbor nodes are sampled uniformly for each node, and Batch is used for training.

2.3.2 ClusterGCN

ClusterGCN[3] is a new extension of GCN proposed by google after GraphSage. So why does ClusterGCN think Sage is not good enough? One of the main reasons is that while Sage performs a sampling operation, this method works for a small number of neighboring nodes and a low layer count, but not for a larger number of samples or a deeper layer count. For example, a simple two-layer GraphSage, with the first layer sampling 10 and the second layer sampling 10, for the training of each node (in GNN the training process of a node needs to use the data of its neighboring nodes, for example, the two-layer GCN needs to use the data of 100 nodes in the domain for the training of each node according to the sampling number of 10, 10), the training and The training and parameter update requires too much input data, and this is only for a single sample point, and the computational pressure is even greater for a 256-batch training.

The idea of ClusterGCN is very simple, violent and straightforward. Since GCN cannot handle a large graph at one time, we cut the large graph into a large number of small subgraphs, and then GCN is trained on the small subgraphs separately, which not only can batch training, but also can effectively reduce the pressure on the video memory, and the size of a subgraph is limited, so deep GCN can also run successfully.

For a graph G, we first divide all nodes V in this G into a bunch of communities consisting of a cluster of nodes:V=[$V1$, ...,Vc], with a total of c communities, where Vt

contains all nodes in the tth community. Therefore, we will get the following subgraph:

$$\overline{G} = \{G_1, ..., G_c\} = [\{V_1, E_1\}, ..., \{V_c, E_c\}] \tag{2.3}$$

We divide c clusters as above, and then each cluster can be trained as a batch. It seems that the idea of Cluster-GCN is very simple, but in fact there are some problems.

1. Directly discarding the edge between clusters, obviously the accuracy of the model will have a certain decline.
2. The clustering of nodes through the graph clustering method will have some problems

As can be seen by Table 2.2, GraphSage introduces redundant computation, trading time for space, with time and memory complexity increasing exponentially with K and r. Cluster-GCN does not have the problem of redundant computation, so the time complexity is the same as the baseline approach, and the memory complexity of Cluster-GCN is changed from the previous number of all nodes to one that depends on the batch size. Therefore Cluster-GCN has a lower integrated complexity and is more suitable for implementing deep networks.

Complexity	GCN	GraphSage	Cluster-GCN
Time	O(Kmd+Knd^2)	O(r$^K nd^2$)	O(Kmd+Knd^2)
Memory	O(Knd+Kd^2)	O(sr$^K d + Kd^2$)	O(Ksd+Kd^2)

Table 2.2: Comparison of time and memory complexity between different algorithms.[3] n is the number of nodes, m is the number of edges, K is the number of network layers, s is the batch size, r is the number of neighbours sampled for each node, and d is the dimensionality of the hidden features of the node.

2.3.3 GraphSAINT

Most of the current graph neural network models focus on solving shallow models on relatively small graphs, such as GraphSAGE, VRGCN[17], ASGCN[18], FastGCN[19], and so on. One of the main problems affecting the training of deeper graph neural networks is the ”neighbor explosion”[16]. Therefore training deep models on large graphs still requires faster methods. GraphSAGE, VRGCN, and PinSage[20] require limiting the number of neighbor samples to a small level; FastGCN and ASGCN further limit the neighbor inflation factor to 1, but also encounter challenges in scaling, accuracy, and computational complexity; ClusterGCN speeds up the training process by first clustering the ClusterGCN accelerates the training process by first clustering the graphs and then training the graph neural network on small graphs without encountering the ”neighbor explosion” problem, but introduces bias.

GraphSAINT also first draws subgraphs from the graph G through the sampler, and then constructs the GCN on the subgraphs. but two of the key innovative things are the SAMPLE sampler and the two regularization coefficient sums[16]. Together they guarantee unbiasedness and variance minimization.

Its design of Samper guarantees:

1. Nodes with large influence on each other should be sampled to the same subgraph.

2. Each edge has more than a non-negligible sampling probability.

2.3.4 PM-bases Solution

In addition to algorithmic optimization, the present effort also involves system and hardware optimization. This is the central focus of our project. Currently, [5] is the best source and guide for this project. It details the specific ideas for implementing a PM-based data loader.

During the data preparation process, SSDs are utilised as the storage medium. As the Flash-based SSD system operates as a block device and data is transferred in 4kb coarse blocks, as shown in our table 2.1, the throughput is significantly lower than DRAM. However, given the need for higher computational and memory performance for ML algorithm developers, it is essential to investigate whether NVM can be applied to GNN training.

MMAP[21] and DirectIO[22] are implemented to access the data stored on the SSD (or PMEM) in two distinct methods. The experimental findings demonstrate that DireactIO has superior performance since the DirectIO-based system reads file data directly, bypassing the OS page cache to buffer the most recently requested pages, resulting in reduced latency and quicker speed. This is an excellent guideline for our future investigations.

2.4 Task

Our task is to implement a sustainable memory-based dataset loader and combine it with a GNN model to test the performance of different GNN algorithms under that condition.

2.5 Selection of Dataset

This project chose two kinds of size datasets, one small scale and one large scale datasets. These datasets of different scales played different roles at different stages of this project.

2.5.1 Small-Scale Datasets

The small-scale datasets comes from the citation network, mainly including Cora[23], PubMed[24] and Citeseer[25]. the main reason for using the small-scale dataset is to make it easier to test the data loader developed for this project, considering our limited hardware conditions (smaller capacity DRAM).

	Cora	**Citeseer**	**Pubmed**
Nodes	2708 (1 graph)	3327 (1 graph)	19717 (1 graph)
Edges	5429	4732	44338
Features per Node	1433	3703	500
Classes	7	6	3
Training Mask	140	120	60
Validation Mask	500	500	500
Test Mask	1000	1000	1000

Table 2.3: The comparison of data scale of different datasets, Training Mask refers to the number of nodes randomly selected by Mask to participate in training(Validation Mask and Test Mask are the same).[23][24][25]

2.5.2 Large-Scale Datasets

For the large scale dataset we have used the Reddit dataset[26]. This combines considerations of current hardware conditions, as Reddit is the threshold for relatively large scale datasets. The specific indicators for this dataset are shown in table 2.4.

Dataset	**Nodes**	**Edges**	**Features**	**Classes**	**Training Mask**	**Validation Mask**	**Test Mask**
Reddut	232965 (1 graph)	11606919	602	41	153431	23831	55703

Table 2.4: Specific metrics for the Reddit dataset

Chapter 3

Methodology

3.1 Emulating Persistent Memory

3.1.1 System and Environment Preparation

The experiments for this project were carried out on a Linux system. As the local system of the physical machine for this project is Windows 10, the choice of using a local dual system solution for this project will satisfy all software requirements for this experiment. Note that choosing VMware workstation as the virtual machine to build the experimental environment is not a more achievable option. VMware ESXi[27] can support PCIe[28] direct connection with GPU, but this method only works for certain GPU models (NVIDIA GRID, NVIDIA Tesla series, etc.).

3.1.2 Persistent Memory Configuration

In this project, we used Linux to emulate 8GB Persistent Memory, details of which can be found on the official Pmem.io[29] website. Note that the Linux kernel only supports this feature after version 4.0. The specific Linux version and the procedure for emulating Persistent Memory are shown in Appendix A.

3.2 PM-based-Dataloader development

The development of the data loader is the basis for achieving the objectives of this project. As the project experiments were conducted on emulated persistent memory, rather than conventional DRAM, a new data loader needed to be built to read and write data from the PM device.

The design of the data loader for this project was based on previous work[5], and two versions of the data loader were programmed using MMAP and DirectIO respectively.

3.2.1 Dataloader with MMAP

MMAP is used to map(bind) a file into memory space. Simply put, MMAP is an image of the contents of a file inside memory. After a successful mapping, changes made by the user to this memory area can be directly reflected in kernel space, and similarly, changes made to this area in kernel space can be directly reflected in user space, just as show in figure 3.1. This is a very efficient way of transferring large amounts of data between kernel space and user space.

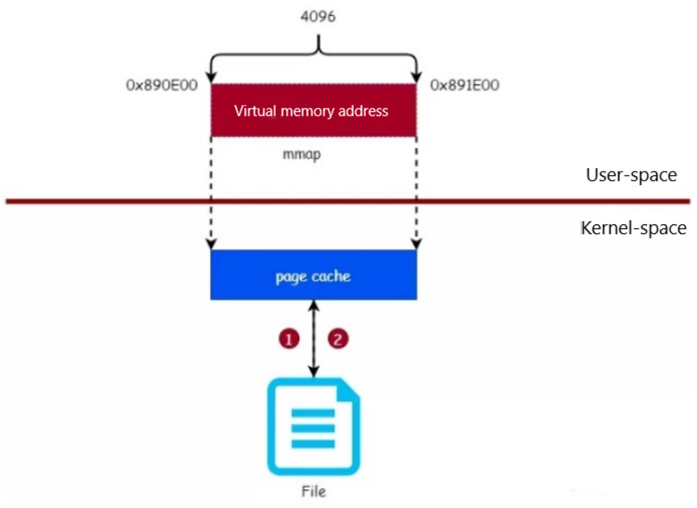

Figure 3.1: MMAP Operation

The MMAP system call allows processes to share memory with each other by mapping the same common file. Once the file is mapped to the process address space, the process can access the file as if it were normal memory, without having to call read(), write(), etc. MMAP does not allocate space, it just maps the file to the address space of the calling process (but takes up virtual memory on the Linux system).

The function deveolpment in dataloader:

1. The *read()* function uses *mmap()* to map the address of the corresponding file directly into virtual memory, and the bytes returned by *decode()* to get the data in the file.

2. In the *write()* function we can call *msync()* to explicitly synchronise the file, so that what you have written is immediately saved to the file.

3. In the *write()* function, we can call *msync()* to explicitly synchronise the file, so that what you have written can be saved to the file immediately. *closemmap()* calls *munmap()* to unmap the memory.

The code for this dataloader is written in C. As the machine learning front-end is python, we need to first compile the source file into a *.so* file via the *gcc* command, which is loaded and called in python via *cdll.LoadLibrary()*. See Appendix B.2 for details of how to use this.

3.2.2 Dataloader with DirectIO

Both MMAP and regular Buffered IO access to files go through the kernel's page cache. Data transfer: disk $->$ page cache $->$ user buffer requires two memory copies. For this reason, Linux provides a way to bypass the page cache for reading and writing files: direct I/O. The steps for data transfer are shown in Figure 3.2.

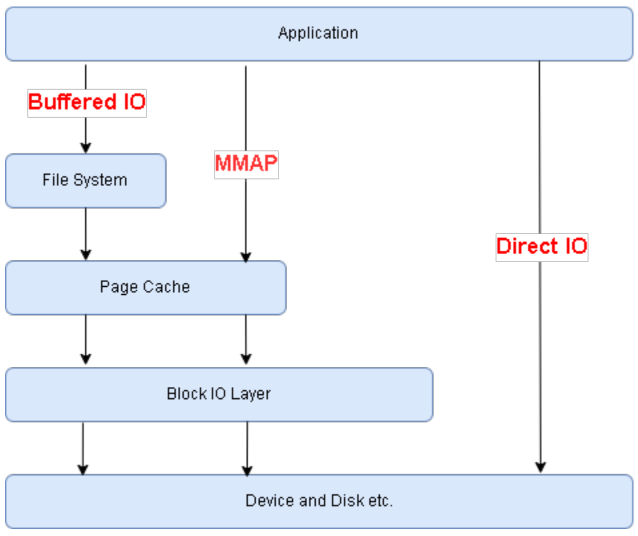

Figure 3.2: Difference between Buffered IO,MMAP and DirectIO

Using Direct IO eventually requires a call to the c language *pwrite()* and *pread()* interface and the setting of the O_DIRECT flag, which has a number of limitations:

1. The memory boundary of a buffer used to pass data must be aligned to an integer multiple of blockSize (typically 512 bytes).

2. The size of the buffer used to transfer data must be an integer multiple of block-Size.

3. The start point of the data transfer, i.e. the file and device offset, must be an integer multiple of blockSize.

This project uses the third-party Python library directio[30] directly. the file address is passed to directio via the front-end os.open (file) and the data on the SSD is manipulated directly via read() and write(). Installation and use are explained in detail in Appendix B.2.

3.3 Naive GNN Baseline

Three GNN models were selected for experimentation and evaluation in this project: GCN, GraphSage (with sampling) and ClusterGCN. The structure and parameters of the three were also tuned for different databases to determine a baseline for the optimal performance (training time and accuracy) that the three models could achieve with the current experimentation hardware.

As the main purpose of this experiment is to discuss the performance of the three models on DRAM and PM devices, the Baseline is built on DRAM-based hardware conditions.

3.3.1 GNN Model

3.3.1.1 Basic GCN

The model does not contain sampling, so large datasets like Reddit cannot be loaded onto DRAM in one go. So for this experiment the model is only applied to small datasets like Cora, Citeseer and Pubmed. For the underlying GCN model, we used a single implicit layer network structure. After testing we established 16 hidden layer neurons as shown in Table 3.1.

3.3.1.2 GraphSage (with sampling)

For GraphSage, we built our model by referring to the paper and sample code published by Will Hamilton et al [6]. Based on the physical conditions of this experiment and the two sizes of data sets, we constructed two sizes of network structures, as shown in the table 3.2. We set the number of hidden layer neurons to 16 for small scale datasets such as Cora, and 256 for large scale datasets such as Reddit, to simulate the more complex

Parameter	Value
GCNConv Layers	2
GCNConv Layer1	(Node features,16)
GCNConv Layer2	(16, Classes)
Batch Size	128
Activation	ReLU
Loss Function	NLLLoss
Optimiser:	Adam
Adam Learning Rate	0.01
Adam Weight Decay	$5e^{-4}$

Table 3.1: GCN model architecture and parameter configuration

network structures that are often required to obtain better learning results as the dataset grows larger under realistic training conditions.

Parameter	Value
SAGEConv Layers	2
SAGEConv Layer1	(Node features,256/16)
SAGEConv Layer2	(256/16, Classes)
Activation	ReLU
Loss Function	NLLLoss
Optimiser:	Adam
Adam Learning Rate	0.01
Adam Weight Decay	$5e^{-4}$

Table 3.2: GraphSage model architecture and parameter configuration

Unlike Basic GCN, GraphSage uses a sampling algorithm to accommodate training on large data sets. This uses the *NeighborSampler()* function in Pytorch Geometric. The core idea of the *NeighborSampler* is that, given the nodes of the mini-batch and the number of layers of the graph convolution, L, and the number of neighbours to be sampled at each layer, SIZE, the neighbours are sampled and a subgraph is returned for each layer in turn, from the first layer to the layer L. The following is a summary of the main logic.

For i in L:

1. Layer 1 uses the nodes of the initial minibatch to sample neighbours and returns the result of the sampling.
2. For layer i, neighbours are sampled using all the nodes involved in the previous layer's sampling, and the sampling result is returned.

We have tuned the *NeighborSampler* on the large-scale and pre-scale datasets separately. This allows it to be trained as fast as possible while still meeting the hardware performance. The specific hyperparameters are set in the table 3.3 below.

Parameter	Value(Small/Large)
Data mask	Training Mask
Size	[10, 10]/[25, 15]
Batch Size	16/256
Num of Workers	4/8

Table 3.3: Hyperparameters of Neighborsampler on small/Large scale dataset

For example, on a small data set, *Size=[10, 10]* specifies that this is a two-layer convolution. The first layer of convolution samples the number of neighbours 10, and the second layer of convolution samples the new number of neighbours 10 based on the neighbouring nodes of the previous layer. *Batch Size=16* specifies the number of nodes in the mini-batch, sampling only 16 nodes at a time.. *Num of Worker* is also an important parameter for the performance of the machine, which refers to the number of CPU cores involved in data processing during the sampling phase. When the sampling time is much longer than the training time, increasing the number of workers will improve the computational efficiency; if the training time is longer, it means that the CPU processing speed meets the GPU demand, and allocating more CPU cores will instead spend more time on thread allocation. Since on small-scale datasets, the GPU's computing time is much less than the CPU's time for sampling and transferring data, the number of Workers is set to 4 is sufficient. On the large dataset, on the contrary, we set *Num of Work = 8*.

3.3.1.3 ClusterGCN

ClusterGCN's model development is based on the paper and open source code of Wei-Lin Chiang et al. In order to unify the variables, the structure and scale of the model is consistent with the previously built GraphSage, as shown in table 3.2.

Unlike GraphSage, ClusterGCN divides the entire image into K clusters first, and then goes on to sample the neighbourhoods within the clusters. and restrict the neighbour search in that subgraph. This greatly reduces the complexity of sampling and thus removes the problem of the neighborhood explosion. This simple and effective strategy can significantly improve memory and computational efficiency, while being able to achieve comparable test accuracy to previous algorithms (in theory there may be a slight loss of accuracy, with some edge information missing due to the entire graph being cut into multiple clusters).

The clustering operation is implemented by the *ClusterData()* function of PyG. Through the study of the ClusterGCN paper we found that since subgraph nodes within clusters are highly correlated and similar, graph clustering algorithms tend to cluster similar nodes together and therefore the distribution of clusters may differ from the original dataset, resulting in a biased estimate of the full gradient when performing SGD(Stochastic gradient descent)[31] updates. Therefore, we also used PyG's *ClusterLoader()* function to implement a stochastic multi-clustering framework, where each Batch then selects multiple subgraphs for aggregation in order to recover inter-cluster connectivity and improve accuracy.

Parameter	Value(Small/Large)
Num of Parts	100/1000
Batch Size	16/32
Num of Workers	4/8

Table 3.4: Hyperparameters of Clustering on small/Large scale dataset

The parameters were selected as shown in the table 3.4. In order to meet the existing hardware conditions to successfully complete the training of large graphs, we set the *Num of Parts = 1000* and divided it into 1000 subgraphs to reduce the complexity of neighbor sampling and data volume, while setting the *Batch size = 32* in the aggregation stage, i.e. 32 subgraphs are processed per batch in training.

3.3.2 Hardware and Software

We performed all of the experiments in this project using a dual Windows and Linux based PC with the following hardware and software. Processor: Intel 12-core i7-8750H 2.2 GHz. GPU: NVIDIA GeForce RTX2070 Max-Q Design. DRAM: 16GB 2667MHZ

64bit DDR4. SSD: WDS100T3X0C-00SJG0 1TB. Development environment: Anaconda 4.8.4 for Linux running Jupyter Notebooks. Deep-Learning Library and tools: Python v3.7.13, Pytorch Geometric v2.0.4, Pytorch v1.11.0, cudatoolkit v11.3.1.

3.4 Evaluation Metrics

As the main objective of this project is to compare the performance of different models on different physical storage media, the evaluation metrics focus on the computational efficiency of the models and the physical feedback from the computer rather than on the classification accuracy of the models. We will apply our evaluation metrics to all GNN models (DRAM and SSD based) during the experiment and perform comparative analysis.

3.4.1 System-level Evaluation Metrics

The evaluation metrics are implemented using the Python extension packages *nvsmi*, *psutil* and *threading*. *nvsmi* is primarily used to monitor the physical metrics of the GPU, while *psutil* is used to monitor the memory, disk and CPU metrics. To enable runtime measurements with GNN training, we used *threading* to enable multi-threaded tasks and set the detection granularity at 1 time/s and 10 times/s to target models with different training speeds.

3.4.1.1 GPU Memory Usage

The GPU memory is used to store the data coming in from the CPU that needs to be computed. It is a good reflection of the current operating state of the GPU. Ideally, the GPU memory occupancy is high, which proves that the amount of data transferred by the CPU is sufficient to support the GPU's operations.

3.4.1.2 GPU Idle Rate

The metric = 1 - GPU runtime / total training time. As the training data is transferred in accordance with the producer-consumer model (CPU as producer, GPU as consumer), the higher the data throughput of the CPU and memory, the lower the value.

3.4.1.3 GPU Util

GPU utilisation is an indicator of how busy the various resources on the GPU are, for this project the main resource is the CUDA core.

3.4.1.4 Memory Usage

Memory usage can help us choose the right sampling hyperparameter (the more nodes sampled, the higher the memory footprint), and it can also give a side-effect of the CPU's processing power and memory transfer efficiency (when both are weak, unprocessed data will pile up in memory).

3.4.1.5 Disk IO Usage

As both the data loading phase and DirectIO read data from disk, statistics on this metric are also crucial to our analysis of the data loader's performance.A good data loader can maximise disk IO usage to complete the transfer of data as quickly as possible.

3.4.1.6 CPU IOwait

IOwait is the percentage of time that the CPU is idle and at least 1 I/O is in process. When the value of IOwait is small, this indicates that the higher the percentage of CPU time used, the higher the execution capacity of the program (throughput). Generally speaking, if io wait is greater than 25%, we need to consider if there is a bottleneck in io.

3.4.2 Model Fine-grained Evaluation Metric

We implemented fine-grained time detection for each phase of the model with a customised SageConv. The types of time detected are shown in the table 3.5.

Metric	Description
Average Linear Time	Time of linear transformation
Average Message Time	Time for information transfer between nodes in a graphical neural network
Average Aggregate Time	Mimicking the work of a convolutional neural network, the time to update the next layer of the network
Average Update Time	Time to update node information
Average Sample Time	Time of Neighbour Sampling

Table 3.5: Model Fine-grained Evaluation Metric

Chapter 4

Experimental Results and Discussion

The experiments were conducted using two physical storage media going separately, regular DRAM and simulated PM. To better observe the effect of data volume on the two different algorithms, we used the Cora (Citeseer and Pubmed are in the same scale as it) and Reddit datasets, respectively. To exclude instability in hardware performance, we repeated the training 10 times in each model configuration and hardware condition and took the average.

4.1 Independent testing of the data loader

Before the formal experiments we first tested the read performance of each of the two data loaders. The time taken to read the same size files of 7717kb for both *cora.cites* and *cora.content* was recorded. The results are shown in the table 4.1.

Dataloader	**File Size**	**Loading Time**	**Speed**
MMAP	7717kb	15-20 ms	385.9-514.2 Mb/s
DirectIO	7717kb	4-6 ms	1286.1-1929.3 Mb/s

Table 4.1: The comparison of reading speed of different data loaders

The read speed of DirectIO was as fast as we expected. As the local SSD has a maximum transfer speed of 3430Mb/s, this data loader can run 37-56% of the full SSD IO bandwidth, which is a relatively good result. However, the results of MMAP were not as good as they could have been. In response we made further explorations. We added time calculations to the C source code of this data loader and found that the mapping operation of MMAP took only 30-40 microseconds. This was 100 times worse than what we had previously measured in the Python front-end. As we explored further, we found that the time overhead occurred primarily in the *ctype* package on the Python front-end. This package is used to call the C code where we use the *restype()* function which converts the bytes into the characters the user needs, and it is the function that spends most of the time loading the data. By studying the DirectIO source code, we found that the data loader is based on the Python/C API, which is more efficient than going through third party packages. This also provided us with guidance for optimising our data loader in the future.

4.2 Experiment on Small-scale Dataset

4.2.1 Result on DRAM

First we experimented with three baseline models on DRAM devices (Basic GCN was not applied to large datasets because of the sheer volume of data that could not be loaded onto DRAM or PM in one go without sampling operations). Based on the hyperparameters we set before, we obtained the results shown in the table 4.2 and 4.3(all values are average values during training).

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
Basic GCN	3208.0Mb	0.0%	6.0%	1.69Gb	0.0297 Gb/s	0.0	1.74 s
GraphSage	1584.7Mb	48.6%	1.2%	1.35Gb	0.0373 Gb/s	0.0	75.13 s
ClusterGCN	698.0Mb	47.6%	1.3%	1.47Gb	0.0432 Gb/s	0.0	80.25 s

Table 4.2: The comparison of different DRAM-based GNN models on small dataset

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	54.288 ms	0.095 ms	3.944 ms	1.590 ms	2172.139 ms
ClusterGCN	59.222 ms	0.002ms	3.874 ms	1.281 ms	2129.835 ms

Table 4.3: Fine-grained comparison of different DRAM-based GNN models on small dataset

Looking at table 4.2 we see that GraphSage and ClusterGCN do reduce the strain on GPU memory,GPU util and DRAM to some extent. The reason for this is that the sampling algorithm transfers only a small fraction of the data to the GPU for training, at the cost of a significant increase in training time, which confirms the ”time for space” concept expressed in the GraphSage paper[6].

The most critical finding is the huge difference in GPU idle rates between the models. Table 4.2 shows that on the basic GCN, no GPU idling occurs. The main reason for this is that the neighbour sampling algorithm requires the CPU to process the data before passing it to the GPU, during which time the GPU has already computed the incoming data, so the power of the GPU drops, as can be seen in Figure 4.3 and Figure 4.4. It can also be seen from the table 4.3 that sampling takes up the most time in each training epoch.

At the same time, we can see that the value of CPU IOwait is 0 for all three models, which proves that the data transfer capability between CPU and memory can satisfy the CPU's computational power under the current database and hardware conditions.

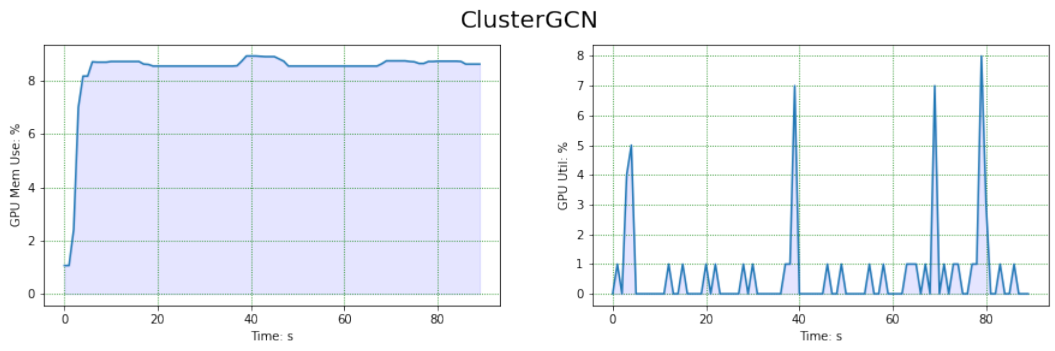

Figure 4.1: GPU Run-time performance on DRAM(ClusterGCN)

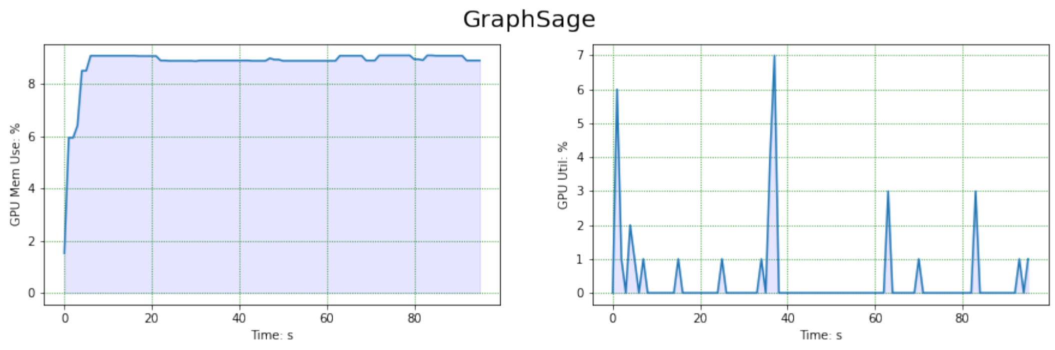

Figure 4.2: GPU Run-time performance on DRAM(GraphSage)

4.2.2 Result on PM

At this stage of the experiment we used the same hyperparameter settings as before on the PM. The results are shown in the table 4.4, 4.5, 4.6 and 4.7.

By observing and analysing the above data, we find that PM devices do not have too much impact on performance on small data sets. The sampling phase of the model is the

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
Basic GCN	3165.9Mb	0.0%	6.1%	1.70Gb	0.0312 Gb/s	0.0	2.93 s
GraphSage	1584.7Mb	56.7%	1.1%	1.33Gb	0.0322 Gb/s	0.0	97.56 s
ClusterGCN	654.1Mb	55.4%	1.1%	1.41Gb	0.0301 Gb/s	0.0	100.43 s

Table 4.4: The comparison of different PM-based GNN models(with MMAP) on small dataset

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	53.167 ms	0.100 ms	3.944 ms	1.411 ms	3264.699 ms
ClusterGCN	56.876 ms	0.002ms	3.874 ms	1.383 ms	3341.432 ms

Table 4.5: Fine-grained comparison of different PM-based GNN models(with MMAP) on small dataset

most affected part, with the use of both MMAP and DirectIO increasing the sampling time by over 20%. The MMAP-based data loader is less efficient compared to DirectIO, as we would expect, mainly due to the Python front-end calls to C taking more time by using *ctype* package. We therefore proceeded to experiment on a large dataset in the expectation that higher data processing volumes would open up the performance gap between the DRAM and PM-based data loaders and facilitate our identification of problem areas.

4.3 Experiment on Large-scale Dataset

The experiments in this section were conducted on the Reddit dataset, which is more than 100 times the size of the previous dataset. The experimental steps remain the same as those in the smaller dataset.

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
Basic GCN	3314.8Mb	0.0%	6.1%	1.70Gb	0.0294 Gb/s	0.0	2.17 s
GraphSage	1410.7Mb	56.7%	1.1%	1.33Gb	0.0311 Gb/s	0.0	86.54 s
ClusterGCN	655.9Mb	55.4%	1.1%	1.41Gb	0.424 Gb/s	0.0	92.11 s

Table 4.6: The comparison of different PM-based GNN models(with DirectIO) on small dataset

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	49.414 ms	0.088 ms	4.132 ms	1.432 ms	2765.431 ms
ClusterGCN	52.223 ms	0.001ms	3.901 ms	1.381 ms	2832.299 ms

Table 4.7: Fine-grained comparison of different PM-based GNN models(with DirectIO) on small dataset

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
GraphSage	7442.1Mb	18.6%	23.3%	6.82Gb	4.586 Gb/s	0.0	756.1 s
ClusterGCN	4986.1Mb	14.8%	23.1%	6.03Gb	5.750 Gb/s	0.0	780.0 s

Table 4.8: The comparison of different DRAM-based GNN models on large dataset

4.3.1 Result on DRAM

By looking at tables 4.8 and 4.9, we can see that the training time for the model increases considerably as the data set increases. The sampling time increases by a factor of almost 10 compared to its predecessor on the small data set, which is related to the increase in the number of nodes and edges (more neighbouring nodes need to be traversed during random sampling). At the same time, the GPU's memory footprint and utilities have increased significantly due to the increase in data volume (and correspondingly more computational power), which is the reason why the total time for training and evaluation is not as high as the increase in sampling time.

We also found by comparing between models that the sampling time and GPU memory usage of ClusterGCN is much lower than it is on GraphSage. This suggests that the subgraph division of ClusterGCN is able to significantly limit the number of neighbouring nodes as a way to reduce the complexity of sampling. We can see from Figures A and B that the GPU's computational intensity fluctuates with time. To see this more clearly, we have zoomed in on the GPU usage within 1 epoch. Figure B shows that the GPU has two peaks within an epoch, one for the training phase and one for the evaluation phase, and since the evaluation phase takes place within the full graph of nodes, the GPU memory usage is higher(the condition is same for two different models..

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	342.467 ms	3.002 ms	396.767 ms	205.719 ms	14351.452 ms
ClusterGCN	172.078 ms	0.074ms	27.230 ms	8.597 ms	9248.860 ms

Table 4.9: Fine-grained comparison of different DRAM-based GNN models on large dataset

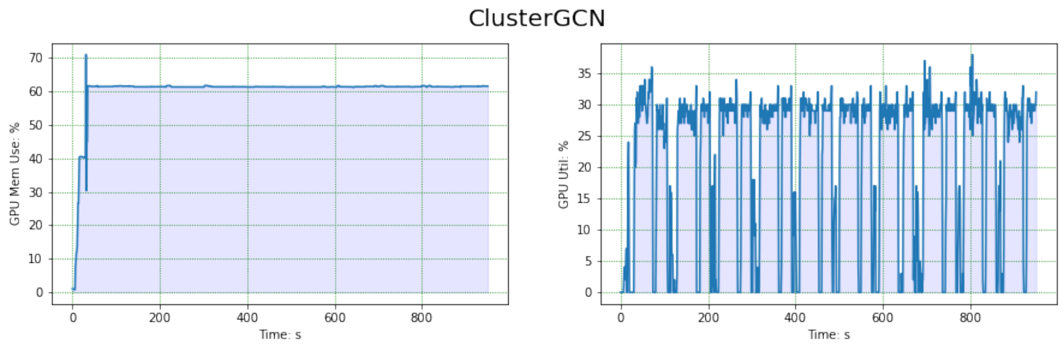

Figure 4.3: GPU Run-time performance on DRAM(ClusterGCN)

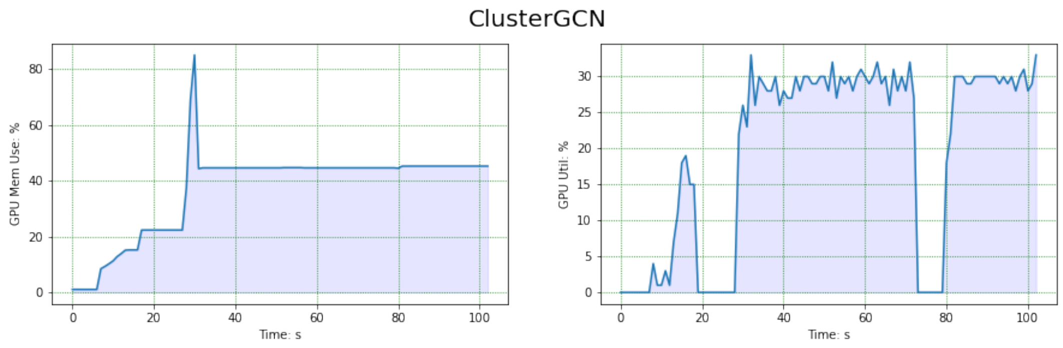

Figure 4.4: GPU Run-time performance on DRAM(GraphSage)

4.3.2 Result on PM

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
GraphSage	7362.5Mb	53.1%	12.1%	6.65Gb	4.322 Gb/s	34.3	2431.6 s
ClusterGCN	4698.5Mb	50.6%	14.9%	6.14Gb	5.0301 Gb/s	38.1	2154.9 s

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	335.113 ms	3.201 ms	412.605 ms	189.324 ms	40031.802 ms
ClusterGCN	186.329 ms	0.080ms	32.322 ms	7.313 ms	28614.124 ms

Table 4.10: Comparison of different PM-based GNN models(with MMAP) on large dataset

Using tables 4.10 and 4.11, we can see that when the model is run in a PM-based environment, the CPU experiences a severe IO wait, while the time of the sampling phase increases substantially. This indicates that the processing power of the CPU has been greatly exceeded and the data transferred into the CPU. As seen in our previous independent tests of dataloader, the front-end calls to python were time consuming and required heavy use of MMAP and DirectIO to access the data on disk for sampling during training. Although the time spent in the standalone tests was minimal, this is an

Model	GPU Memory Usage	GPU Idle Rate	GPU Util	Memory Usage	Disk IO Usage	CPU IOwait	Training Time
GraphSage	7211.6Mb	35.7%	21.0%	6.63Gb	4.237 Gb/s	15.0	1855.2 s
ClusterGCN	4873.9Mb	33.2%	23.7%	6.54Gb	5.3424 Gb/s	16.8.0	1683.1 s

Model	Linear Time	Message Time	Aggregate Time	Update Time	Sample Time
GraphSage	330.151 ms	3.280 ms	412.450 ms	188.231 ms	2986.371 ms
ClusterGCN	221.321 ms	0.041ms	32.347 ms	9.222 ms	1830.100 ms

Table 4.11: Comparison of different PM-based GNN models(with DirectIO) on large dataset

unacceptable time overhead during the high density of computing and sampling.

4.4 Discussion

Considering that the physical storage device used in this experiment to simulate PM is an SSD with a maximum throughput of 3.4GB/s, the IO throughput can hardly be called a performance bottleneck. The latency of SSD is about 100 times that of DRAM [32], so the main loss in performance should be in the latency of the storage medium by preliminary analysis. However, since each node in the graph structure needs to sample neighbouring nodes to update its own information, the number of sampling operations in the GNN model cannot be reduced. We have thought of increasing the number of CPU cores to reduce the CPU IO wait.

We increase the number of CPU core calls to 10 from the original 8, so that during training the other cores can submit the processed data to the GPU before one core has finished processing it. However, through preliminary experiments, we found that increasing the number of CPU cores involved in sampling did not result in this problem. We also found that even with multiple cores processing in parallel, when IO blocking is encountered, multiple cores still need to wait for data to be transferred. Conversely too many CPU cores involved can also cause processes to freeze, and when we increase the number of cores to 10, the amount of interrupts in the system increases significantly.

In summary, therefore, both GraphSage and ClusterGCN perform well below DRAM on PM devices, which is closely related to the latency of the SSD used to emulate Pmem. When processing large datasets on DRAM devices, ClusterGCN outperforms GraphSage due to the limitation of the size of the sampled neighbouring nodes by dividing the clusters, which is well reflected in the occupancy of DRAM during training.

Chapter 5

Conclusions and Future Work

In this work, we develop and use PM-based data loaders (MMAP-based and DirectIO-based) and combine them with a variety of GNN models, and explore their respective performance on DRAM and PM, respectively, and the reasons for the performance loss. We find that both models perform much lower on PM devices than on DRAM, mainly reflected by the low uitlity of the GPU and the long training time on large datasets. We conclude that the latency of the PM device is the bottleneck that limits the training performance of the GNN models. This reflects the high IO wait of the CPU during training, while multi-threaded CPU processing does not help this problem very much and only increases efficiency when the CPU's processing performance is a bottleneck.

Notably, as noted in the work by Wei-Lin Chiang et al [5], the current trend in technology has seen the emergence of high-bandwidth, low-latency system interconnects (CXL, CAPI, Gen-Z, etc.), which are expected to further narrow the gap between SSDs and traditional DRAMs. Also, from our research, we found that the latest version of the machine learning library DGL can load data directly onto the GPU for equal probability neighbour sampling. This avoids the time overhead of transferring data from CPU memory to GPU memory during each iteration.

5.1 Future work

In the future we will continue to explore the feasibility of GNN mass training on PM devices. As there is no real PM device for this experiment, we can only simulate it via Linux. Many real experimental discussions need to be conducted on real devices, such as Intel optane NVME. In the future we will continue to explore the feasibility of GNN mass training on PM devices. As there is no real PM device for this experiment, we can only simulate it via Linux. Many real experimental discussions need to be conducted on real devices, such as Intel optane NVME[33].

At the same time the GPU sampling acceleration scheme offered by DGL is equally attractive and offers an alternative way of thinking about implementing large-scale training to explore a conventional SSD+DGL implementation of large-scale training.

Bibliography

[1] Zhiyuan Liu and Jie Zhou. Introduction to graph neural networks. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(2):1–127, 2020.

[2] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

[4] Chenyi Zhuang and Qiang Ma. Dual graph convolutional networks for graph-based semi-supervised classification. In *Proceedings of the 2018 World Wide Web Conference*, pages 499–508, 2018.

[5] Yunjae Lee, Youngeun Kwon, and Minsoo Rhu. Understanding the implication of non-volatile memory for large-scale graph neural network training. *IEEE Computer Architecture Letters*, 20(2):118–121, 2021.

[6] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[7] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2):1–32, 2017.

[8] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] Geoffrey E Hinton and Richard Zemel. Autoencoders, minimum description length and helmholtz free energy. *Advances in neural information processing systems*, 6, 1993.

[12] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[13] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–12. IEEE, 2009.

[14] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, 2014.

[15] Shimin Chen, Phillip B Gibbons, Suman Nath, et al. Rethinking database algorithms for phase change memory. In *Cidr*, volume 11, pages 9–12, 2011.

[16] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[17] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[18] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems*, 31, 2018.

[19] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[20] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[21] Python Software Foundation. mmap — memory-mapped file support¶. Website, 2022. https://docs.python.org/3/library/mmap.html.

[22] Alexander Sandler. Direct io in python. Website, 2010. http://www.alexonlinux.com/direct-io-in-python.

[23] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

[24] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[25] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pages 89–98, 1998.

[26] Jason Baumgartner, Savvas Zannettou, Brian Keegan, Megan Squire, and Jeremy Blackburn. The pushshift reddit dataset. In *Proceedings of the international AAAI conference on web and social media*, volume 14, pages 830–839, 2020.

[27] Forbes Guthrie, Scott Lowe, and Kendrick Coleman. *VMware vSphere design*. John Wiley & Sons, 2013.

[28] Satish K Dhawan. Introduction to pci express-a new high speed serial data bus. In *IEEE Nuclear Science Symposium Conference Record, 2005*, volume 2, pages 687–691. IEEE, 2005.

[29] Maciej Maciejewski. Pmem.io: How to emulate persistent memory. https://pmem.io/blog/2016/02/how-to-emulate-persistent-memory/. Accessed 22 May, 2022.

[30] Andronik Ordian. directio 1.2. https://pypi.org/project/directio/description/. Accessed 22 May, 2022.

[31] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116, 2004.

[32] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[33] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

Appendix A

Linux Kernel Versions and Processes of PMEM Emulation

This guide is based on the official guidelines of Pmem.io. The link to the website is given in the main body text.

Since version 4.0, persistent memory devices and emulation are supported in the kernel, but it is recommended to use a kernel newer than 4.2 as it is easier to configure.

Memory areas are reserved by modifying kernel command line parameters to make them appear to be persistent memory. The memory area to use, from ss to ss+nn.[KMG] means thousand, megabyte, gigabyte.

memmap=nn[KMG]!ss[KMG]

For example memmap=4G!12G to reserve 4GB of memory between the 12th and 16th GB. The configuration is done in GRUB and varies from one Linux distribution to another. Here are two examples of GRUB configurations.

Ubuntu Server 15.04:

sudo vi /etc/default/grub

GRUB_CMDLINE_LINUX="memmap=nn[KMG]!ss[KMG]"

sudo update-grub2

CentOS 7.0

sudo vi /etc/default/grub

GRUB_CMDLINE_LINUX="memmap=nn[KMG]!ss[KMG]"

On BIOS-based machines

sudo grub2-mkconfig -o /boot/grub2/grub.cfg

On UEFI-based machines:

sudo grub2-mkconfig -o /boot/efi/EFI/centos/grub.cfg

To install a filesystem using DAX (available today for ext4 and xfs)

```
sudo mkdir /mnt/mem
sudo mkfs.ext4 /dev/pmem0 OR sudo mkfs.xfs -m reflink=0 /dev/pmem0
sudo mount -o dax /dev/pmem0 /mnt/mem
```

Appendix B

Dataloader Using Instructions

B.1 MMAP

To compile *mmaploader.c* to *mmaploader.so* as an example

Commond: *gcc mmaploader.c -shared -o mmaploader.so*

Importing header files in python files

*from ctypes import **

Introduce the c dynamic library in python, and use variables to receive references to the dynamic library

result = cdll.LodaLibrary(". /mmaploader.so")

Calling dynamic library methods

result.my_add(num)

B.2 DirectIO

Python console: *pip install directio*

```
import directio
fd_o = os.open(content, os.O_DIRECT | os.O_RDONLY)
content = directio.read(fd_o,10521600)
```