Benchmarking GNN-Based Recommender Systems on Intel Optane Persistent Memory

Yuwei Hu
Cornell University
Ithaca, New York, USA
yh457@cornell.edu

Jiajie Li
Cornell University
Ithaca, New York, USA
jl4257@cornell.edu

Zhongming Yu*
Tsinghua University
Beijing, China
yzm18@mails.tsinghua.edu.cn

Zhiru Zhang
Cornell University
Ithaca, New York, USA
zhiruz@cornell.edu

ABSTRACT

Graph neural networks (GNNs), which have emerged as an effective method for handling machine learning tasks on graphs, bring a new approach to building recommender systems, where the task of recommendation can be formulated as the link prediction problem on user-item bipartite graphs. Training GNN-based recommender systems (GNNRecSys) on large graphs incurs a large memory footprint, easily exceeding the DRAM capacity on a typical server. Existing solutions resort to distributed subgraph training, which is inefficient due to the high cost of dynamically constructing subgraphs and significant redundancy across subgraphs.

The emerging Intel Optane persistent memory allows a single machine to have up to 6 TB of memory at an affordable cost, thus making single-machine GNNRecSys training feasible, which eliminates the inefficiencies in distributed training. One major concern of using Optane for GNNRecSys is Optane’s relatively low bandwidth compared with DRAMs. This limitation can be particularly detrimental to achieving high performance for GNNRecSys workloads since their dominant compute kernels are sparse and memory access intensive. To understand whether Optane is a good fit for GNNRecSys training, we perform an in-depth characterization of GNNRecSys workloads and a comprehensive benchmarking study. Our benchmarking results show that when properly configured, Optane-based single-machine GNNRecSys training outperforms distributed training by a large margin, especially when handling deep GNN models. We analyze where the speedup comes from, provide guidance on how to configure Optane for GNNRecSys workloads, and discuss opportunities for further optimizations.

1 INTRODUCTION

Graph neural networks (GNNs) have emerged as a promising approach to building recommender systems, where the task of recommendation can be formulated as the link prediction problem on user-item bipartite graphs [10, 61, 63, 68]. More concretely, a GNN model generates a dense vector representation for each vertex (namely, the embedding vector), and a pair of vertex embeddings are used to carry out the per-edge prediction [26, 49, 53]. GNNs can generate high-quality vertex embeddings by incorporating multi-hop neighborhood information through iterative message passing [8, 24, 41]. For these reasons, recent years have seen a rapid surge of development on GNN-based recommender systems (GNNRecSys) in both academia and industry [10, 19, 31, 39, 61, 68, 74].

One major challenge faced by GNNRecSys workloads is that they consume a large amount of memory. First, real-world user-item

Our measurement shows that the read bandwidth of Optane is 40% that of DRAM and the write bandwidth of Optane is only 20% that of DRAM. This result is consistent with prior studies [38, 67].
We focus on the use of Optane as large volatile memory.

1. While the performance of GNNRecSys is negatively impacted by the relatively low bandwidth of Optane, using Optane together with DRAMs can largely recover the performance. Managing the hybrid Optane+DRAM memory system at a granularity of pages through the OS (in AppDirect Mode) works better than managing it at a granularity of cache lines by the hardware (in Memory Mode). This is because GNNRecSys workloads have a large memory access size due to the embedding vectors.

2. Both the memory consumption and execution time of GNNRecSys workloads are dominated by two types of sparse tensor compute kernels, namely, sampled dense-dense matrix multiplication (SDDMM) and sparse-dense matrix multiplication (SpMM). These two types of kernels require different settings for optimal performance, including the NUMA data placement policy, the number of threads, and whether to use non-temporal write instructions.

3. With the optimal setting, Optane-based single-machine GNNRecSys training demonstrates a significant speedup over distributed subgraph training when handling deep GNN models (with at least two layers). The speedup mainly comes from the ability of single-machine training to use a large batch size and avoid cross-batch redundant computation that is pervasive in distributed training.

These benchmarking results, for the first time, make a case for GNNRecSys on Optane. In addition to shedding light on why Optane is a good fit for GNNRecSys and providing guidance on how to configure Optane for GNNRecSys, our benchmarking study reveals several potential optimizations to further improve the performance of GNNRecSys on Optane.

The rest of the paper is organized as follows. Section 2 reviews the background of GNNRecSys, explains the inefficiencies of distributed subgraph GNN training, and introduces Optane. Section 3 describes the benchmarking setup. Section 4 presents workload characterization of the GNN model that we use for the benchmarking study. Section 5 reports bandwidth measurements of Optane and discusses potential implications. Section 6 and 7 present kernel-level and end-to-end benchmarking results, respectively. Section 8 discusses optimization opportunities. We survey related work in Section 9 and summarize in Section 10.

2 BACKGROUND

2.1 GNN-Based Recommender Systems

Recommender systems are crucial to the business of online services in the era of information explosion. Authors in [64] reported that up to 75% of movies watched on Netflix and 60% of videos consumed on YouTube come from their recommender systems.

The conventional approach to building recommender systems is collaborative filtering through matrix factorization [33, 42]. Figure 1a illustrates an example of a user-item bipartite graph where an edge can represent a click, a purchase, etc. The matrix factorization method encodes each user/item into an embedding vector such that the dot product between the embeddings of a <user, item> pair can indicate the user’s preference for the item; it updates the embeddings based on observed user-item interactions using gradient descent and then uses the obtained embeddings to predict missing interactions.

Recently, GNNs [28, 41, 58] are gaining popularity as an effective method for learning high-quality vertex embeddings, thus offering a new approach to building recommender systems. Unlike the matrix factorization method that directly feeds the user/item embeddings into the prediction function (e.g., dot product), GNNs refine the embeddings through message passing on the user-item interaction graph. More concretely, in the message passing paradigm, each vertex computes a new representation by aggregating messages from its incoming edges. Let \( G(V, E) \) denote a graph with a set of vertices \( V \) and a set of edges \( E \), \( (src, e, dst) \) denote an edge \( e \) pointing from vertex \( src \) to \( dst \), and \( x_v \) denote the initial embedding vector associated with vertex \( v \). The message passing paradigm carries out the following computations:

\[
m_v = \phi(x_{src}, x_{dst}), (src, e, dst) \in E
\]
would double the memory consumption. Our profiling shows that to tackle the memory capacity bottleneck in large-scale GNN training, a large batch size is desired [25, 70] since fewer steps are needed to converge, existing efforts resort to distributed GNN training frameworks, such as DistGNN [48], P3 [20], and DistDGL [73], among which only DistDGL is open-source. DistDGL builds upon DGL [60], a widely-used GNN framework.

Figure 2 illustrates the subgraph training approach adopted by DistDGL. Let us assume the GNN model has two layers; there are two machines, and the batch size on each machine is one (the aggregate batch size is therefore two). On machine 1, DistDGL selects $i_2$ as the target vertex and constructs a subgraph consisting of $i_2$ and its one- and two-hop neighbors — we can apply two message-passing layers on the subgraph to compute the embedding for $i_2$. Similarly, on machine 2, DistDGL constructs a subgraph for $i_4$. One can immediately see the issue of redundancy, in both computation and memory consumption, across these two subgraphs.

In addition, because of the exponential growth of the subgraph size, only a limited batch size is allowed. For faster training convergence, a large batch size is desired [25, 70] since fewer steps are needed to iterate the data points (i.e., one epoch).\(^4\) One method to reduce the subgraph size is sampling [11, 12, 28], i.e., picking a small number of neighbors for each vertex instead of considering all the neighbors. Sampling, however, incurs accuracy loss [40, 56]. Besides, sampling is less effective in reducing the subgraph size for deep GNN models [44, 45].

\[ h_{dst} = \bigoplus m_e, (src, e, dst) \in E \]  
\[ x^{\text{new}}_{dst} = \psi(h_{dst}) \]

Here $\phi$, $\bigoplus$, and $\psi$ are customizable functions for generating messages, aggregating messages, and updating the embedding, respectively. A GNN model iteratively applies Equations (1) (2) (3) — we call one such iteration a message-passing layer as shown in Figure 1c — so that a vertex can incorporate its multi-hop neighborhood information into its embedding. Prior studies [31, 61] and our experiments show that increasing the number of message-passing layers improves the performance of GNNRecSys.\(^3\) When auxiliary attributes are available, such as the user’s age, gender, etc., GNNs can utilize them to augment the initial embeddings. We refer interested readers to [63] for a comprehensive survey on the landscape of GNNRecSys research.

GNNRecSys workloads demand a large amount of memory. More concretely, the memory consumption of Equation (1) is $\text{len}(m) \times |E|$, of Equation (2) is $\text{len}(h) \times |V|$, and of Equation (3) is $\text{len}(x) \times |V|$. The above analysis only considers the forward propagation; training would double the memory consumption. Our profiling shows that on a graph with one million vertices and three hundred million edges, training a three-layer GNN model with $\text{len}(m)$, $\text{len}(h)$, and $\text{len}(x)$ all set to 128 requires 500 GB of memory, easily exceeding the DRAM capacity on a typical server. In comparison, running PageRank on the same graph only requires 3 GB of memory.

2.2 Distributed Subgraph GNN Training

To tackle the memory capacity bottleneck in large-scale GNN training, existing efforts resort to distributed GNN training frameworks, such as DistGNN [48], P3 [20], and DistDGL [73], among which only DistDGL is open-source. DistDGL builds upon DGL [60], a widely-used GNN framework.

Figure 2 illustrates the subgraph training approach adopted by DistDGL. Let us assume the GNN model has two layers; there are two machines, and the batch size on each machine is one (the aggregate batch size is therefore two). On machine 1, DistDGL selects $i_2$ as the target vertex and constructs a subgraph consisting of $i_2$ and its one- and two-hop neighbors — we can apply two message-passing layers on the subgraph to compute the embedding for $i_2$. Similarly, on machine 2, DistDGL constructs a subgraph for $i_4$. One can immediately see the issue of redundancy, in both computation and memory consumption, across these two subgraphs.

In addition, because of the exponential growth of the subgraph size, only a limited batch size is allowed. For faster training convergence, a large batch size is desired [25, 70] since fewer steps are needed to iterate the data points (i.e., one epoch).\(^4\) One method to reduce the subgraph size is sampling [11, 12, 28], i.e., picking a small number of neighbors for each vertex instead of considering all the neighbors. Sampling, however, incurs accuracy loss [40, 56]. Besides, sampling is less effective in reducing the subgraph size for deep GNN models [44, 45].

3 Matrix factorization can be viewed as a zero-layer GNN model.

4 When increasing the batch size, we need to increase the learning rate as well [25].
### Table 1: Different ways of using Optane.

|                      | Memory Mode                           | AppDirect Mode                        |
|----------------------|---------------------------------------|---------------------------------------|
|                      | managed by                            | NUMA file system                      |
|                      | granularity                           | OS programmer                         |
|                      | flexibility                           | arbitrary size                        |
|                      | code modification?                    | high                                  |
|                      |                                       | low                                   |
|                      |                                       | medium                                |
|                      |                                       | no                                    |
|                      |                                       | OS programmer                         |
|                      |                                       | arbitrary size                        |
|                      |                                       | high                                  |
|                      |                                       | low                                   |
|                      |                                       | medium                                |
|                      |                                       | no                                    |

### 3 BENCHMARKING SETUP

**Datasets.** Table 2 lists the datasets that we use for the benchmarking study. movielens-10m [29], gowalla [14], and amazon-book [30] are real-world datasets for recommending movies, locations, and books, respectively. They have been widely used in GNNRecSys research [10, 31, 61]. Among these three datasets, movielens-10m has the highest density, which is 1.3%. To work around the issue that industry-scale datasets for recommender systems are not publicly available, we follow the Kronecker expansion method proposed in [9] to synthetically generate large datasets by expanding existing small ones. Kronecker expansion can preserve the original graph’s characteristics, such as power-law degree distribution, community structure, item popularity, etc. We expand movielens-10m, gowalla, and amazon-book to three medium-size graphs (around 300M edges) and three large-size graphs (around one billion edges); m×25 denotes a graph that is expanded from movielens-10m by a factor of 25 in terms of the number of edges. We use movielens-10m, gowalla, and amazon-book to verify the GNN model’s accuracy, and the six synthetic datasets for performance benchmarking.

**Table 2: Datasets.**

| dataset               | # users | # items | # interactions | density |
|-----------------------|---------|---------|----------------|---------|
| movielens-10m         | 70K     | 11K     | 10M            | 1.34%   |
| gowalla               | 30K     | 41K     | 1M             | 0.08%   |
| amazon-book           | 53K     | 92K     | 3M             | 0.06%   |
| m×25                  | 345K    | 53K     | 250M           | 1.34%   |
| g×256                 | 478K    | 656K    | 26M            | 0.08%   |
| a×100                 | 526K    | 916K    | 296M           | 0.06%   |
| m×100                 | 699K    | 107K    | 1000M          | 1.34%   |
| g×1024                | 955K    | 131K    | 1052M          | 0.08%   |
| a×400                 | 1053K   | 183K    | 1194M          | 0.06%   |

**Optane Machine.** Figure 3 depicts the architecture of the Optane machine. It is a 2.2 GHz Intel Xeon Platinum 8276L machine with two sockets connected by the Intel Ultra Path Interconnect (UPI). Each socket has 28 physical cores, two integrated memory controllers (IMCs), six DRAM DIMMs, and six Optane DIMMs. Each DRAM DIMM is 32 GB; each Optane DIMM is 128 GB. The total DRAM capacity is 384 GB (32 × 6 × 2); the total Optane capacity is 1536 GB (128 × 6 × 2).

**Software.** We use DGL v0.7.2 with PyTorch v1.7 backend to implement GNN models. Transparent Huge Pages (THP) are enabled (default in Linux). We use PyTorch Profiler to get execution time breakdown, mprof to measure memory consumption, and pmu-tools to measure cross-NUMA accesses.

---

Yuwei Hu, Jiajie Li, Zhongming Yu*, and Zhiru Zhang

---

### 4 WORKLOAD CHARACTERIZATION

We choose neural graph collaborative filtering (NGCF) [61] for the benchmarking study, which is a representative GNN model that has been widely used in recommender systems. NGCF follows the message-passing paradigm described in Section 2.1. Specifically, NGCF’s message generation function, message aggregation function, and embedding update function are defined as follows. Here ⊙ denotes element-wise multiply and W1 and W2 are trainable weight matrices.

\[
m_e = (x_{src} \odot x_{dst}) \times W1 + x_{src} \times W2, (src, e, dst) \in E \quad (4)
\]

\[
h_{dst} = \text{sum}(m_e), (src, e, dst) \in E \quad (5)
\]

\[
x_{new}^w = h_{dst} \quad (6)
\]

There are various other GNN models that differ from NGCF in the following aspects: (1) simplifying the message generation function (e.g., removing W1 and W2 [31]); (2) using a different message aggregation function, such as max or importance pooling [68]; (3) jointly considering the user-item interaction graph and user-user social graph [19]. The above three differences in the model architecture do not impact the underlying computation pattern. More concretely, prior studies [37, 60] have shown that the computation of any GNN model that follows the message-passing paradigm can be mapped to two types of sparse tensor compute kernels — we will elaborate on this point later in this section.

We care about two parameters in the architecture of NGCF: the embedding length |ℓ|, and the number of message-passing layers. Increasing either one of these two parameters typically improves the model accuracy, but at the cost of higher memory consumption and computational complexity. Specifically, for full-graph training, the memory consumption and computational complexity grow linearly with both the embedding length and the number of layers; for subgraph training, the growth is linear with the embedding length but exponential with the number of layers. For the benchmarking study, we vary these two parameters — setting the number of layers to one or two or three, setting the embedding length to 128 or 256 — to create six variants of NGCF. We use NGCF-IL-128E to denote an NGCF model with one layer and the embedding length set to 128.
We recognize several inefficiencies in the original implementation of NGCF in DGL and apply a set of optimizations. Figure 4a illustrates the dataflow graph (one message-passing layer) of the original implementation for computing user embeddings; the dataflow graph for computing item embeddings is symmetric. Our first optimization is to switch the execution order between multiplying weight matrices and aggregating messages, that is, we convert $\sum((x_{src} \odot x_{dst}) \times W1 + x_{src} \times W2)$ into $\sum(x_{src} \odot x_{dst}) \times W1 + \sum(x_{src}) \times W2$, as illustrated in Figure 4b. This optimization reduces the computational complexity of multiplying weight matrices from $O(|E|)$ to $O(|U|)$, where $|E|$ is the number of edges and $|U|$ is the number of users. Our second optimization is to replace less-efficient scatter and sum operations with sparse tensor compute kernels that have been highly optimized by the DGL framework. Concretely, we use generalized SDDMM (sampled dense-dense matrix multiplication) and SpMM (sparse-dense matrix multiplication) to implement message generation and aggregation, respectively, as illustrated in Figure 4c. While the standard SDDMM performs dot product in its inner most loop, a generalized one supports element-wise add, which is required by NGCF. For SpMM, while the standard one performs add reduction, a generalized one supports max reduction among others. Moreover, the gradient calculation of (generalized) SDDMM and SpMM are also mapped to these two kernels. We refer interested readers to [37] for a thorough description on the connection between message passing and SDDMM/SpMM kernels. Our last optimization is to reuse the SDDMM results obtained during computing user embeddings to compute item embeddings (since $x_{src} \odot x_{dst} = x_{dst} \odot x_{src}$), instead of calculating SDDMM twice as in the original implementation.

Figure 5 shows that combining the three optimizations accelerates the inference of NGCF-3L-128E by 8.3x and training by 8.0x on the movielens-10m dataset. We will use the optimized implementation for benchmarking. Figure 6 shows the execution time breakdown. SDDMM and SpMM together take 91% of the total time for inference and 75% for training. Besides, the add operation takes 17% of the training time, which is mainly used for updating trainable weights with gradients during backpropagation.

8https://docs.dgl.ai/api/python/dgl.ops.html
5 OPTANE BANDWIDTH MEASUREMENTS

Since GNNRecSys workloads are memory bandwidth bound and less sensitive to the memory latency [20, 62], we focus on bandwidth measurements. Specifically, we measure the bandwidth of Optane under both sequential and random memory access patterns for both remote (i.e., cross-socket) and local accesses.

Figure 7a shows that for sequential memory accesses, the peak read bandwidth of Optane (with 28 threads) is 39.2 GB/s, which is 37% of what DRAM offers; the peak write bandwidth of Optane (with 2 threads) is 3.5 GB/s, which is 7% of that of DRAM; when using non-temporal write (nt write), the peak write bandwidth of Optane (with 8 threads) is 11.3 GB/s, which is 18% of that of DRAM. Compared to a normal write, an nt write achieves a higher bandwidth by bypassing the cache hierarchy and thus preventing the cache eviction mechanism from converting sequential stores into random stores [11].

Figure 7b shows that for random memory accesses, the bandwidth utilization of Optane increases as the access size increases. Specifically, the write bandwidth utilization saturates at an access size of 256 bytes; the read bandwidth utilization is not saturated even at 4096 bytes. For GNNRecSys workloads, the access size is typically hundreds of bytes depending on the embedding length, whereas for traditional graph processing workloads such as PageRank, it is only four bytes. Figure 7c shows that the remote read bandwidth of Optane is comparable to that of DRAM (34.5 vs. 35.8 GB/s); both are limited by the UPI bandwidth. The remote nt write bandwidth of DRAM is even higher than the local nt write bandwidth of Optane, which indicates that it could be beneficial to allocate memory on all the sockets to utilize more DRAMs even when the data can fit into Optane on a single socket.

To understand the implications of Optane on the performance of GNNRecSys workloads, we have the following questions to be answered through benchmarking:

1. To what extent does the relatively low bandwidth (especially write) of Optane hurt the performance of GNNRecSys?
2. Is it beneficial to use nt write for GNNRecSys?
3. What is the optimal NUMA configuration for GNNRecSys?
4. What is the tradeoff between accuracy and efficiency with different embedding lengths?

6 KERNEL-LEVEL BENCHMARKING

In this section, we aim to (1) understand the implications of Optane on the performance of SDDMM and SpMM, the two dominant compute kernels in NGCF and GNNRecSys workloads in general, and (2) give suggestions on how to configure Optane to achieve high performance.

Figure 8 shows the execution time of SDDMM and SpMM using Optane vs. DRAM, tested on the three synthetic million-edge graphs, i.e., m-x25, g-x250, and a-x100. Using Optane alone gives a lower bound for the achievable performance since in practice we always use Optane together with DRAM. The main results are: (1) When using normal write, the performance of SDDMM on Optane is 7.7–7.8× lower than that on DRAM; for SpMM, this number is 2.2–3.0×. SDDMM has a severe slowdown on Optane because SDDMM is write intensive and the write bandwidth of Optane is less than 10% that of DRAM (see Figure 7a). In comparison, SpMM is read intensive and therefore sees a more modest slowdown on Optane. (2) When switching from normal write to nt write, the performance of SDDMM improves by 1.1× on DRAM and 1.4× on Optane; the performance of SpMM, however, degrades significantly — more than 20× for both DRAM and Optane. This is because SpMM has a large degree of temporal locality due to its aggregation computation pattern; hence, using nt write that bypasses the cache hierarchy only hurts its performance. In comparison, SDDMM has no temporal locality in accessing its outputs and therefore can benefit from the higher write bandwidth brought by nt write. (3) Among the three datasets, which have a similar number of edges, SpMM runs fastest on m-x25 for both Optane and DRAM. This is because m-x25 has a higher density, leading to a higher degree of locality and consequently higher cache utilization. In remaining experiments, we use nt write for SDDMM and normal write for SpMM.

Figure 9 shows the impact of the number of threads on the performance of SDDMM and SpMM, tested on the m-x25 dataset. For SDDMM, the optimal performance is achieved with 4 threads on Optane and 28 threads on DRAM. This result is consistent with the observation we made in Figure 7a that the nt write bandwidth of Optane saturates at 4–8 threads. For SpMM, the optimal performance is achieved with 28 threads for both Optane and DRAM, which is consistent with the observation that the read bandwidth of both Optane and DRAM keeps increasing with the number of threads up to 28. Results on g-x256 and a-x100 show the same...
trend. Ideally, we should set the number of threads to 4 for SDDMM and 28 for SpMM. However, the DGL framework does not support tuning the number of threads per compute kernel. In remaining experiments, we still use 28 threads (per socket) for both kernels, which is optimal for SpMM but leads to around 20% performance drop for SDDMM on Optane compared with using 4 threads.

Figure 10 shows the impact of different Optane configurations, tested on the three synthetic billion-edge graphs, i.e., m-x100, g-x1024, and a-x400. Data exceeds the DRAM capacity but fits into Optane on a single socket. In the remaining experiments, we use AppDirect Mode and first allocate memory on all the sockets. This result suggests that we should allocate memory on all the sockets to utilize more DRAMs even when data fits into Optane on a single socket. In the remaining experiments, we use AppDirect Mode and first allocate memory on DRAMs.

Figure 11 shows the impact of NUMA data placement policies. We consider two policies: (1) interleaving the pages across socket in a round-robin fashion and (2) blocking the pages and distributing the blocks among sockets. We find that for SDDMM, blocked data placement works better; for SpMM, interleaved data placement works better. SDDMM and SpMM prefer different NUMA data placement policies because of their different computation patterns. For SDDMM, each thread writes to a contiguous range of memory locations; therefore, blocked data placement can guarantee that...
each thread only needs to access its local socket. For SpMM, each thread gathers values from incontiguous memory locations and aggregates the values; although interleaved data placement can not fully eliminate cross-NUMA accesses, it incurs fewer than blocked data placement. For end-to-end benchmarking, we use blocked NUMA data placement for optimal overall performance.

The above results provide the following guidelines: (1) using ntwrite for SDDMM and normal write for SpMM; (2) preferring AppDirect Mode over Memory Mode; (3) preferring blocked NUMA data placement over interleaved. We adopt the three guidelines when doing end-to-end benchmarking. Tuning the number of threads per kernel can also be beneficial, although we did not explore this optimization as it is not supported by the DGL framework.

7 END-TO-END BENCHMARKING

7.1 Model Accuracy

In this subsection, we evaluate (1) the model accuracy of multiple variants of NGCF that have different embedding length and number of layers and (2) impact of sampling on the model accuracy.

We split the edges of a graph into 90% for training and 10% for testing. The evaluation metric is recall@20, i.e., recall calculated based on the top-20 recommended items. The loss function is Bayesian personalized ranking (BPR) [55], which assumes that observed interactions should be assigned higher prediction values than unobserved ones. To calculate the BPR loss requires a tuple of a user, one of the user’s interacted items, and one un-interacted item. The implementation in the NGCF paper [61] computes 1K tuples at each step (hence a batch size of 1K) with a learning rate of 0.0001. For faster training, especially on large graphs, we need to increase the batch size and scale the learning rate accordingly. We tried both square root scaling [43] and linear scaling [25], and found the latter works better. By linearly scaling the learning rate, we managed to increase the batch size to 100K while achieving the same recall@20 within the same number of epochs as using the original 1K batch size.

Table 3 reports recall@20 on the amazon-book dataset. Both increasing the embedding length and increasing the number of layers lead to improved recall@20. We see the same trend on movielens-1m and gow11a as well, confirming that a larger GNN model is desired for building higher-quality recommender systems.

Table 4 reports recall@20 degradation due to sampling for NGCF-3L-256E. When the sampling factor is 10 (i.e., sampling 10 neighbors for each vertex), the recall@20 of NGCF-3L-256E degrades from 0.072 to 0.066, which is even worse than the recall@20 of NGCF-2L-256E without sampling. When the sampling factor is 100, which is larger than the average degree of amazon-book, a still, a degradation of 0.001 is incurred. The reason is that amazon-book, as well as other user-item interaction graphs, has a power-law degree distribution where a small portion of vertices have a degree significantly larger than the average, as shown in Figure 12. Those high-degree vertices (either popular items or active users) tend to be more important for the task of recommendation, but they also suffer more severe information loss from sampling.

Table 3: Recall@20 of different NGCF variants — The dataset is amazon-book; a higher recall is better.

| embedding length | 1 layer | 2 layers | 3 layers |
|------------------|---------|----------|----------|
| 128              | 0.061   | 0.065    | 0.066    |
| 256              | 0.065   | 0.069    | 0.072    |

Table 4: Recall@20 degradation due to sampling — The dataset is amazon-book; the model is NGCF-3L-256E.

| sampling factor | 10  | 20  | 50  | 100 |
|-----------------|-----|-----|-----|-----|
| degradation     | -0.006 | -0.004 | -0.002 | -0.001 |

Figure 12: Degree distribution of amazon-book.

7.2 Comparison With DistDGL

In this subsection, we compare Optane-based single-machine NGCF training with distributed training using DistDGL, in terms of hardware cost ($) and performance. We perform experiments on m-x25, g-x256, and a-x100 datasets.

We run DistDGL on an in-house cluster of six servers. Each server is a two-socket 32-core 2.8 GHz Intel Xeon Gold 6242 machine with 384 GB DDR4 memory. The servers are in the same rack and connect to 10 Gigabit Ethernet. The aggregate memory capacity of the cluster is 2304 GB, which is 20% larger than the memory capacity of the Optane machine (1536 GB of Optane plus 384 GB of DRAM). For the cluster, the total price of the DRAMs is $67K; for the Optane machine, the price of Optane+DRAM is $11K [1].

Table 5 reports the maximum aggregate batch size that is allowed by the memory capacity when using DistDGL. An aggregate batch size of six means a batch size of one per machine. The batch size quickly decreases as the number of layers of the NGCF model increases. Specifically, without sampling, NGCF-1L-128E allows a batch size of 24K, while NGCF-2L-128E only allows 384; further increasing the number of layers to three would run out of memory even at a batch size of one per machine. Sampling increases the batch size and makes three-layer NGCF models possible to train. For NGCF-1L-128E and NGCF-1L-256E, we set the batch size to 100K not because of the memory capacity limit, but to ensure training convergence. All Optane experiments use a batch size of 100K.

Table 6 reports the performance comparison between Optane-based single-machine training and DistDGL, measured by the execution time for processing 100K edges. The key observation is that Optane-based single-machine training achieves significant speedup over DistDGL on deep NGCF models (with two or three layers), because the execution time of Optane-based single-machine training increases roughly linearly with the number of layers while
Table 5: Maximum aggregate batch size allowed by the memory capacity when using DistDGL. The sampling factor is 100; “/” means out-of-memory even at a batch size of one per machine.

|                  | NGCF-1L-128E | NGCF-1L-256E | NGCF-2L-128E | NGCF-2L-256E | NGCF-3L-128E | NGCF-3L-256E |
|------------------|--------------|--------------|--------------|--------------|--------------|--------------|
| w/o sampling     | 24K          | 12K          | 384          | 192          | /            | /            |
| w/ sampling      | 100K         | 100K         | 12K          | 6K           | 256          | 128          |

Table 6: Speedup of Optane-based single-machine training over DistDGL. Time (unit: sec) is for processing 100K edges.

|                  | DistDGL w/o sampling | DistDGL w/ sampling | Optane | speedup over w/o sampling | speedup over w/ sampling |
|------------------|-----------------------|---------------------|--------|---------------------------|--------------------------|
| NGCF-1L-128E     | m-x25: 93             | 7                   | 100    | 0.9×                       | 0.07×                     |
|                  | g-x256: 27            | 13                  | 103    | 0.3×                       | 0.13×                     |
|                  | a-x100: 30            | 14                  | 155    | 0.2×                       | 0.09×                     |
| NGCF-1L-256E     | m-x25: 214            | 11                  | 293    | 0.7×                       | 0.04×                     |
|                  | g-x256: 64            | 20                  | 316    | 0.2×                       | 0.06×                     |
|                  | a-x100: 74            | 19                  | 351    | 0.2×                       | 0.05×                     |
| NGCF-2L-128E     | m-x25: 44434          | 61                  | 187    | 237.1×                     | 0.3×                      |
|                  | g-x256: 13623         | 156                 | 228    | 59.9×                      | 0.7×                      |
|                  | a-x100: 12417         | 181                 | 308    | 40.3×                      | 0.6×                      |
| NGCF-2L-256E     | m-x25: 63311          | 156                 | 596    | 106.1×                     | 0.3×                      |
|                  | g-x256: 27936         | 318                 | 579    | 48.2×                      | 0.5×                      |
|                  | a-x100: 23680         | 337                 | 743    | 31.9×                      | 0.5×                      |
| NGCF-3L-128E     | m-x25: /              | 6146               | 287    | /                          | 21.4×                     |
|                  | g-x256: /             | 18139              | 335    | /                          | 54.1×                     |
|                  | a-x100: /             | 24036              | 425    | /                          | 56.6×                     |
| NGCF-3L-256E     | m-x25: /              | 16211              | 906    | /                          | 17.9×                     |
|                  | g-x256: /             | 47056              | 921    | /                          | 51.1×                     |
|                  | a-x100: /             | 58779              | 1051   | /                          | 55.9×                     |

Figure 13: Execution time breakdown of DistDGL. The dataset is m-x25; w/o sampling; execution time of each part is averaged over six machines.

Optane takes 100 seconds for NGCF-1L-128E and 287 seconds for NGCF-3L-128E, while DistDGL with sampling takes 7 seconds for NGCF-1L-128E and 6146 seconds for NGCF-3L-128E. On NGCF-2L-128E and NGCF-2L-256E, using Optane brings 32–237× speedup over DistDGL without sampling. On NGCF-3L-128E and NGCF-3L-256E, even with sampling, using Optane still brings up to 57× speedup. The results demonstrate the advantages of Optane-based single-machine training for handling deep GNN models.

To further examine the inefficiencies of DistDGL, we plot its execution time breakdown in Figure 13. Building subgraph takes 16–32% of the total time, even higher than the forward propagation, which takes only about 10%. The backward propagation takes more than 60% of the total time, which is 6× higher than the forward propagation. In comparison, for Optane-based single-machine training, the execution time of backward propagation is only 2–3× higher than forward propagation. The slowness of the backward propagation in DistDGL is due to two factors: (1) exchanging gradients across machines incurs communication overhead; (2) since DistDGL performs synchronous training, the execution time is determined by the slowest machine.

8 OPTIMIZATION OPPORTUNITIES

Cache Optimization for SDDMM and SpMM Kernels. There have been attempts to optimize cache utilization of SDDMM and SpMM kernels through locality-enhancing scheduling. For example, FeatGraph [37] combines graph partitioning with embedding tiling to strike a balance between efficiency of accessing the graph structure and efficiency of accessing embeddings, achieving more than 2× speedup. Hong et al. [36] propose an adaptive tiling strategy that takes into account the degree of each vertex. Various graph reordering methods have also been proposed [6, 7, 51]. All these techniques, however, are evaluated on DRAM-only machines, and it is unclear how effective they are on Optane. Specifically, these techniques are designed without awareness of the asymmetric read and write bandwidth of Optane. We envision that to achieve optimal performance on Optane, a partitioning or reordering technique should prioritize high cache utilization for write, instead of treating read and write equally.
Hybrid Memory Management Tailored to GNNRecSys. Although this benchmarking study focused on using Optane in Memory Mode and in AppDirect Mode through existing NUMA utilities, we envision there is room for performance improvement by tailoring the hybrid memory management policy to GNNRecSys. We observed that turning off NUMA page migration achieves performance comparable to that with NUMA page migration on, or even 5% higher in certain cases likely due to less time in kernel space. This observation indicates that managing the hybrid Optane+DRAM memory system through existing NUMA utilities is far from optimal for GNNRecSys. AutoTM [35] analyzed inefficiencies of the NUMA approach in training convolutional neural networks (CNNs) and proposed to optimize the location and movement of tensors between DRAM and Optane based on an integer linear programming (ILP) formulation. AutoTM requires each tensor to fit into DRAM, which is not guaranteed for GNNRecSys when the graph is large. This is a promising avenue for future research to extend AutoTM to support hybrid memory management at a sub-tensor granularity.

Larger Batch Size. Optane-based full-graph training outperforms DistDGL mainly because of its ability to use a large batch size, thereby reducing cross-batch redundant computation. While the batch size of DistDGL is limited by the DRAM capacity, the batch size of single-machine full-graph training can be arbitrarily large up to the number of edges of the entire graph. In this work, we set the batch size to 100K because empirical evaluation confirmed that it maintains training convergence. Recently, training algorithms that are specially designed for large-batch neural network training are proposed [69, 70], and show promising speedup on CNNs and Transformers [16, 57]. It is worth studying whether we can leverage these advances to further increase the batch size of GNNRecSys training beyond 100K.

9 RELATED WORK

GNN Characterization and Acceleration. Prior efforts for characterizing and accelerating GNN workloads mainly focus on GNN models that are used in vertex classification tasks [21, 22, 37, 40, 56], most notably the GCN model [41]. Compared with GCN, NGCF consumes much more memory for two reasons. First, GCN has a simpler message generation function — just multiplying the embedding vector of the source vertex with a normalization value, which is a constant scalar. As a result, for GCN, message generation and aggregation can be fused into a single SpMM kernel. In contrast, NGCF requires both SpMM and SDDMM. Second, GCN typically has two layers, whereas NGCF has more layers (the default setting in the NGCF paper [61] is three). Hence, NGCF, as well as other GNN models for recommender systems, has a more urgent demand for the high memory capacity of Optane than GCN.

GNN Dataflow Optimization. In this work, we manually optimized the dataflow of NGCF. Recently, Graphiler [65] is proposed to optimize GNN dataflow at the compiler level. Graphiler adopts a pattern matching approach — it iteratively traverses the dataflow graph to match subgraphs with predefined patterns and replace them with optimized ones. Graphiler currently has limited coverage on possible dataflow optimizations; for example, it does not support the optimization of reusing SDDMM results that we applied to NGCF. It is worth further research towards automatic and intelligent GNN dataflow optimization.

Workload Benchmarking on Optane. There is an active body of research on using Optane to run data-intensive applications and evaluate its implications [5, 17, 23, 46]. Among these prior studies, the most relevant one to our work is [23], which benchmarked traditional graph processing workloads such as BFS and PageRank on Optane. Results in [23] demonstrate that Memory Mode works better than AppDirect Mode for traditional graph processing workloads, whereas our benchmarking study shows the opposite for GNNRecSys workloads. This difference is because traditional graph processing workloads have a memory access size of a few bytes while GNNRecSys workloads have a much larger one due to the embedding vectors. Both [23] and our work use Optane as volatile memory. Persistent use cases of Optane include key-value stores [47, 71], text search [4], genomics [59], etc.

Deep Learning for Recommender Systems. Prior to GNNs, neural networks such as multi-layer perceptrons (MLPs) have been applied to recommender systems to replace the hard-coded prediction function (typically dot product). Efforts along this direction include Wide & Deep [13], DeepFM [27], NCF [32], DLRM [32], to name a few. GNNRecSys is complementary to these models, that is, the embeddings that a GNN model generates through message passing can be fed into these models as (part of) the input. Like GNNRecSys workloads, these models also demand a large amount of memory; existing solutions [50, 72] leverage SSD+DRAM. It remains to be explored what a role Optane can play there.

10 CONCLUSION

This work analyzes Optane-based single-machine GNNRecSys training and compares it with distributed training. Our experiments show that the large memory capacity of Optane makes it a good fit for GNNRecSys. Specifically, the main takeaways are: (1) AppDirect Mode works better than Memory Mode because GNNRecSys workloads have a large memory access size due to the embedding vectors. (2) SDDMM and SpMM, the two dominant compute kernels in GNNRecSys workloads, require different settings in terms of the NUMA data placement policy, the number of threads, and whether to use non-temporal write instructions. (3) When properly configured, Optane-based single-machine GNNRecSys training outperforms distributed training by a large margin when handling deep GNN models, mainly because of the ability to use a large batch size and avoid cross-batch redundant computation. In addition, our benchmarking study reveals several opportunities for further optimizations, such as designing locality-enhancing techniques that take into account the asymmetric read and write bandwidth of Optane, tailoring the hybrid memory management policy to GNNRecSys, and leveraging advanced training algorithms to enable a larger batch size. It is our hope that this paper provides practical guidelines for running GNNRecSys on Optane as well as inspiring future research towards more efficient GNNRecSys on Optane.

ACKNOWLEDGMENTS

This research was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Award #2118709, and research gifts from Intel and Facebook (now Meta).
REFERENCES

[1] Intel’s Optane DIMM Price Model. https://thememoryguy.com/intels-optane-dimm-price-model/.

[2] MMKIND. https://github.com/mmkind/mmkind.

[3] PMDK. Persistent Memory Development Kit. https://github.com/psmdev/pmdk.

[4] Shoab Akram. Exploiting Intel Optane Persistent Memory for Full Text Search. Int’l Symp. on Memory Management, 2021.

[5] Shoab Akram. Performance Evaluation of Intel Optane Memory for Managed Workloads. ACM Trans. on Architecture and Code Optimization (TACO), 18(3), 2021.

[6] Junya Arai, Hiroaki Shinokawa, Takeshi Yamamuro, Makoto Oinuma, and Sotetsu Iwamura. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. Int’l Parallel and Distributed Processing Symp. (IPDPS), 2016.

[7] Vignesh Balaji and Brandon Lucia. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. IEEE Int’l Symp. on Workload Characterization (IISWC), 2018.

[8] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational Inductive Biases, Deep Learning, and Graph Networks. arXiv preprint arXiv:1806.01261, 2018.

[9] Francois Bellet, Karthik Lakshminarayanan, Walid Khreich, Yi-Fan Chen, and John Anderson. Scalable Realistic Recommendation Datasets through Fractal Expansions. arXiv preprint arXiv:1901.08910, 2019.

[10] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph Convolutional Matrix Completion. arXiv preprint arXiv:1706.02263, 2017.

[11] Jianfei Chen, Jun Zhu, and Le Song. Stochastic Training of Graph Convolutional Networks with Variance Reduction. Int’l Conf. on Machine Learning (ICML), 2018.

[12] Jie Chen, Tengdus Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. Int’l Conf. on Learning Representations (ICLR), 2018.

[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmen, Tal Shaked, Tushar Chandra, Heidi Azadhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & Deep Learning for Recommender Systems. Workshop on Deep Learning for Recommender Systems, 2016.

[14] Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and Mobility: User Movement in Location-Based Social Networks. Int’l Conf. on Knowledge Discovery and Data Mining (KDD), 2011.

[15] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. MaxFM: A Factorization-Machine-based Neural Network for CTR Prediction. Int’l Joint Conf. on Artificial Intelligence (IJCAI), 2017.

[16] Will Hams, Zhitao Yuan, and Jure Leskovec. Inductive Representation Learning on Large Graphs. Conf. on Neural Information Processing Systems (NIPS), 2017.

[17] Maxwell Harper and Joseph A Konstan. The MovieLens Datasets: History and Context. ACM Trans. on Interactive Intelligent Systems, 2015.

[18] Ruining He and Julian McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. Int’l World Wide Web Conf. (WWW), 2016.

[19] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. Int’l Conf. on Research and Development in Information Retrieval (SIGIR), 2020.

[20] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural Collaborative Filtering. Int’l World Wide Web Conf. (WWW), 2017.

[21] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. Int’l Conf. on Research and Development in Information Retrieval (SIGIR), 2016.

[22] Mark Hildebrand, Julian T Angeles, Jason Lowe-Power, and Venkatesh Akella. A Case Against Hardware Managed DRAM Caches for NVRAM Based Systems. Int’l Symp. on Performance Analysis of Systems and Software (ISPASS), 2021.

[23] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

[24] Changwam Hong, Aravind Sukumaran-Rajat, Irat Nisa, Kunal Singh, and P Sadayappan. Adaptive Sparse Tiling for Sparse Matrix Multiplication. ACM SIGPLAN Conf. on Principles and Practice of Parallel Programming (PPoPP), 2019.

[25] Yewen Hu, Zhaow Ye, Junjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhira Zhang, and Yida Wang. FeatGraph: A Flexible and Efficient Backend for Graph Neural Networks. Int’l Conf. on High Performance Computing, Networking, Storage and Analysis (SC), 2020.

[26] Joseph Izraelevitz, Jian Yang, Jun Yang, Lu Zhang, Juno Kim, Xiao Liu, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv preprint arXiv:1903.05714, 2019.

[27] Huifeng Guo, Ruiming Tang, Yuming Ye, Zhenguo Li, and Xiaqiang He. DeepFM: A Factorization-Machine-based Neural Network for CTR Prediction. Int’l Joint Conf. on Artificial Intelligence (IJCAI), 2017.

[28] Matthew Harper, Greg Hurley, and Jure Leskovec. DeepGraph: Inductive Learning. Int’l Conf. on High Performance Computing, Networking, Storage and Analysis (SC), 2021.

[29] F Maxwell Harper and Joseph A Konstan. The MovieLens Datasets: History and Context. ACM Trans. on Interactive Intelligent Systems, 2015.

[30] Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Matrix Factorization Techniques for Recommender Systems. Computer, 42(8), 2009.

[31] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. arXiv preprint arXiv:1404.5997, 2014.

[32] Guoshao Li, Matthias Muller, Bernard Ghanem, and Vladlen Koltun. Training Graph Neural Networks with 1000 Layers. arXiv preprint arXiv:1901.08910, 2019.

[33] Guoshao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs Go As Deep As CNNs? Conf. on Computer Vision and Pattern Recognition (CVPR), 2019.

[34] Joseph Izraelevitz, Jian Yang, Jun Yang, Lu Zhang, Juno Kim, Xiao Liu, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv preprint arXiv:1903.05714, 2019.

[35] Ankit Jain, Isaac Liu, Ankur Sarda, and Piero Molino. Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations. https://eng.uber.com/uber-eats-graph-learning, 2019.

[36] Zhiaoh Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC. Conf. on Machine Learning and Systems (MLSys), 2020.

[37] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. arXiv preprint arXiv:1609.02907, 2016.

[38] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. Computer, 42(8), 2009.

[39] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. arXiv preprint arXiv:1404.5997, 2014.

[40] Guoshao Li, Matthias Muller, Bernard Ghanem, and Vladlen Koltun. Training Graph Neural Networks with 1000 Layers. Int’l Conf. on Machine Learning (ICML), 2021.

[41] Guoshao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs Go As Deep As CNNs? Conf. on Computer Vision and Pattern Recognition (CVPR), 2019.

[42] Huifeng Guo, Ruiming Tang, Yuming Ye, Zhenguo Li, and Xiaqiang He. DeepFM: A Factorization-Machine-based Neural Network for CTR Prediction. Int’l Joint Conf. on Artificial Intelligence (IJCAI), 2017.

[43] Will Hams, Zhitao Yuan, and Jure Leskovec. Inductive Representation Learning on Large Graphs. Conf. on Neural Information Processing Systems (NIPS), 2017.

[44] Ruining He and Julian McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. Int’l World Wide Web Conf. (WWW), 2016.

[45] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. Int’l Conf. on Research and Development in Information Retrieval (SIGIR), 2020.

[46] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural Collaborative Filtering. Int’l World Wide Web Conf. (WWW), 2017.

[47] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. Int’l Conf. on Research and Development in Information Retrieval (SIGIR), 2016.
