OMeGa: Boosting Large-scale Graph Embeddings with Heterogeneous Memory Processing

Peng Fang[†], Siqiang Luo[‡], Fang Wang[†*], Bolong Zheng[†*], Hong Jiang[§], Dan Feng[†*], Hechang Pan[†], Xingyu Wan[†]
[†]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
[‡]*College of Computing and Data Science, Nanyang Technological University, Singapore*
[§]*Department of Computer Science and Engineering, University of Texas at Arlington, USA*

***Abstract*—Graph embedding, which maps graph nodes to low-dimensional vectors, is a widely used technique for graph representation learning. However, most existing graph embedding models suffer from high memory consumption, limiting their scalability to large graphs. Heterogeneous memory systems that combine DRAM and Persistent Memory (PM) offer new opportunities for scaling up memory capacity. Despite this advantage, the performance gap (on the order of $5\times$) between DRAM and PM is magnified (by 3.3-$4.2\times$) under non-uniform memory access (NUMA) architecture. Additionally, the inherent sparsity of graphs induces numerous random accesses in the fundamental Sparse Matrix and Dense Matrix Multiplication (*SpMM*) operations of graph embedding, hindering high-performance heterogeneous memory processing.**

To address these challenges, this paper presents OMeGa that focuses on Optimizing heterogeneous Memory processing for large-scale Graph embedding. OMeGa leverages an *entropy-aware thread allocation*, simultaneously achieving workload balancing and tail latency reduction across threads. It also incorporates a *workload feature-aware prefetcher* to alleviate random accesses during streaming heterogeneous processing. In addition, OMeGa devises a *NUMA-aware data placement*, aiming to minimize the adverse impact of NUMA on heterogeneous memory. The experiments conducted on billion-scale graphs demonstrate that OMeGa exhibits an average acceleration of 32.03$\times$ with strong scalability. This pioneering capability enables the efficient generation of large-scale graph embeddings, free from the memory size constraints and performance disparities typically encountered in heterogeneous memory systems.

***Index Terms*—Graph Embedding; Heterogeneous Memory; NUMA Architecture; SpMM Operations**

I. Introduction

Over the past decade, graph embeddings have been proven extremely effective in graph representation learning in applications across many domains including link prediction [1], classification [2], clustering [3], and recommendation [4]. In these applications, graphs can have millions of nodes and billions of edges. For instance, the *Twitter-2010* graph includes over 41 million user nodes and more than one billion edges, on top of which it is required to perform tasks such as link prediction and classification tasks [5]. As another example, the graph representing users and products at *Alibaba* also consists of more than two billion user-product edges, forming a giant bipartite graph for its recommendation tasks [6]. This has in turn challenged the efficiency and scalability of the state-of-the-art graph embedding models on large-scale graphs. For example, it would take weeks for LINE [7] and months for DeepWalk [8] / node2vec [9] to learn embeddings for a graph with 100 million nodes and 500 million edges using 20 threads on a modern server [10]. Recently, a fast and scalable graph embedding method ProNE [10] is proposed, outperforming Deepwalk, node2vec and LINE by 10-400$\times$. However, it achieves its performance supremacy at the cost of more than 500 GB of active working memory for learning due to the expensive matrix factorization operations, far exceeding the 32-256 GB working memory of an ordinary commercial server. Although employing memories in a distributed or SSD-based manner, like DistGER [11], DistDGL [12], ZeRO-Infinity [13], and Ginex [14], can partially alleviate this capacity limitation and improve scalability, their performance is constrained by high communication costs among networked clusters/nodes or high I/O overhead during memory interactions. The large memory consumption also limits the availability of training to the system since DRAM is close to its physical process limit [15] and faces severe scalability issues [16].

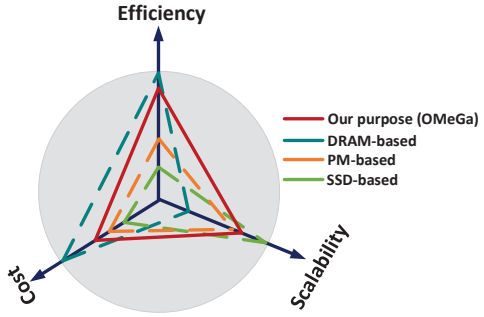

Fig. 1: DRAM-based solutions are efficient but limited by capacity and cost, PM-based solutions balance cost and capacity but face efficiency challenges, and SSD-based solutions offer high capacity but suffer from expensive I/O overhead. OMeGa achieves both high performance and large capacity on low-cost heterogeneous memory.

The emerging high-density memory devices, specifically, *Persistent Memory* (PM), are poised to meet ever-increasing demands for memory capacity PM [17] offers up to 2.1$\times$ lower price per capacity than DRAM [18]. PM leverages the byte-addressable memory technology with the same form factor as DDR4, and when fully populated, can offer up to 12 TB

*Corresponding author

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00252

of capacity [19] due to the high memory density, and it is starting to be deployed in datacenters and clouds to boost the performance of memory-hungry applications [20]–[22]. However, PM exhibits a substantial performance disparity with DRAM, with 1/6 and 1/3 the bandwidth of DRAM in terms of writes and reads, respectively [23], which significantly drags down the efficiency of graph embedding on PM. For example, when running ProNE [10] on *Twitter* graph ($|V| = 11.3$ M and $|E| = 127$ M), PM's capacity/price advantage of $2\times$ over DRAM (785$ on DRAM vs. 312$ on PM) is more than offset by its performance slowdown of $4.96\times$ compared to DRAM. To achieve both large capacity and high performance, a promising solution is to pair PM with DRAM to form a heterogeneous memory system [24]–[26], leveraging their complementary strengths while mitigating individual weaknesses.

Overcoming the physical limitations of heterogeneous memories and adapting to the running characteristics of graph embedding are crucial for providing efficient heterogeneous memory processing. On the one hand, modern commodity multi-core machines have shifted toward the non-uniform memory access (NUMA) architecture [27], where the CPU cores and DRAM/PM are grouped into nodes that are interconnected via inter-node links, creating an access asymmetry between intra- and inter-node memory access latencies. Compared with DRAM, PM further exacerbates the asymmetry of memory access latencies in NUMA systems [16]. As a result, the remote and local latencies are $3.3\times$ and $4.2\times$ higher than the DRAM-base system, respectively, indicating that the performance gap between DRAM and PM is further amplified under the NUMA architecture. On the other hand, Sparse Matrix and Dense Matrix Multiplication (*SpMM*) is the fundamental linear algebra kernel for graph embedding [28]–[30], accounts for nearly 70% of the total overhead [10]. The inherent sparsity of most real-world graphs introduces numerous random memory accesses in *SpMM* operations, which increases the risk of workload imbalance and tail latency in thread allocation while further diminishing the efficiency of graph embedding computations on heterogeneous memory architectures.

To address the above challenges and achieve efficient and scalable graph embeddings, we present a new graph embedding system, OMeGa, which focuses on **O**ptimizing heterogeneous **Me**mory processing for large-scale **Ga**ph embedding. **First and foremost**, OMeGa takes into account both workload balancing and tail latency reduction for *SpMM* operations. It leverages an *entropy-aware thread allocation* scheme (EaTA), which first measures the graph's inherent sparsity based on entropy theory [31], and then adaptively generates the optimal workload for each thread to improve the efficiency of parallel computations. **Second**, to further mitigate the data movements caused by random accesses in *SpMM*, OMeGa introduces a *workload feature-aware prefetcher* (WoFP). During streaming processing between DRAM and PM, WoFP identifies frequently accessed data objects and places them in DRAM, thus achieving high-performance heterogeneous memory processing. **Last but not least**, we start with a quantitative analysis to understand the access characteristic of the PM on NUMA architecture and then design a *NUMA-aware data placement* (NaDP) that maintains global sequential read and local write access principles during *SpMM* operations to minimize the adverse impact of NUMA to heterogeneous memory processing. Figure 1 shows the design objectives of OMeGa, which aims to achieve both large capacity and high performance on low-cost heterogeneous memory, thereby providing efficient and scalable graph embedding.

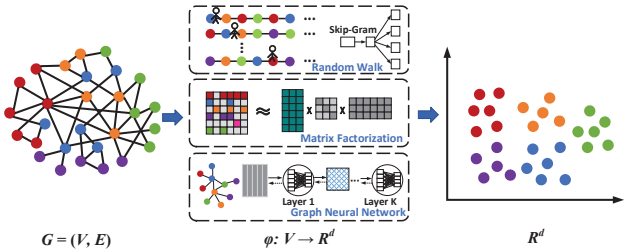

Fig. 2: Schematic diagram of graph embedding technologies.

Our contributions and roadmap are summarized below.

- We propose an efficient, scalable graph embedding system, OMeGa, which, to our best knowledge, is the first graph embedding framework designed and implemented for heterogeneous memory.
- We design an *entropy-aware thread allocation* scheme (EaTA) that focuses on workload balancing and tail latency reduction in parallel computations for graph embedding.
- We devise a *workload feature-aware prefetcher* (WoFP) to alleviate data movement and achieve high-performance heterogeneous memory processing.
- We introduce *NUMA-aware data placement* (NaDP) that minimizes the adverse impact of NUMA to heterogeneous memory and bridges the performance gap between PM and DRAM.
- We conduct extensive experiments to confirm that OMeGa achieves much better efficiency ($32.03\times$ speedup on average) and scalability than our baselines.

II. Preliminaries

A. Graph Embedding

Given an undirected graph $G = (V, E)$ with V as the node set and E as the edge set, the problem of graph embedding aims to learn a mapping function $\varphi : V \rightarrow R^d$ that projects each node to a d-dimensional space ($d << |V|$) to capture the structural properties of a graph. As shown in Figure 2, existing graph embedding technologies roughly fall into three categories: those based on random walk, matrix factorization, and graph neural networks. Random walk-based techniques are inspired by the well-known natural language processing model, word2vec [32]. By conducting sufficient random walks on graphs, substantial graph structural information is collected and fed into the word2vec model (Skip-Gram) to generate node embeddings. Matrix factorization (MF)-based techniques [10], [33]–[36] construct feature representations based on the adjacency or Laplacian matrix. In theory, it is equivalent to random walk-based embedding methods [8], [9], [37], [38], because the

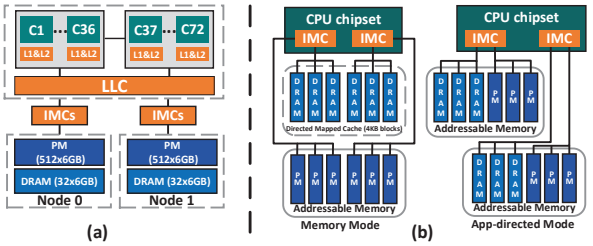

Fig. 3: (a) Memory hierarchy of our two sockets machine, and (b) Configuration mode of PM: Memory and App-directed.

latter can be viewed as an implicit matrix factorization [35]. Graph neural networks (GNN) [39]–[42] focus on generalizing graphs into semi-supervised or supervised learning by using message-passing kernels to aggregate information from a vertex's neighbors. Compared with GNN, the MF-based models are parameter-free, eliminating the need to set suitable tuning learning rates or GNN layers, which can be sensitive to the GNN-based models [43].

During the embedding generation, Sparse Matrix and Dense Matrix Multiplication (*SpMM*) is fundamental and essential for various computations, such as PageRank [44] calculation in random walks, message aggregation in GNN, and matrix operations ubiquitous in MF. In the most notable and representative MF-based model ProNE [10], *SpMM* operations are executed in parallel within the randomized t-SVD (truncated Singular Value Decomposition) [45] and Chebyshev polynomials [46], constituting nearly 70% of the total overhead.

B. Persistent Memory

Persistent memory (PM), also known as non-volatile memory (NVM), combines the byte-addressability and low latency of DRAM with the persistence and large capacity of SSDs. PMs are in the form of standard byte-addressable DDR4 DIMMs on the CPU memory bus, similar to DRAM DIMMs. They communicate with memory controllers via a custom protocol that is mechanically and electrically compatible with DDR4. However, PM modules have a higher capacity than DRAM, with options of 128/256/512 GB per DIMM. In addition, similar to non-uniform memory access (NUMA) systems that divide memory into sockets, PM modules are also distributed among sockets. Figure 3(a) shows an example of a two-socket-node machine with a 6 TB of PM and 384 GB of DRAM split between sockets. Each processor has two integrated memory controllers (iMCs) supporting six channels each, pairing six DRAM and six PM modules per socket.

PM typically operates in two modes as shown in Figure 3(b). **Memory Mode:** In this mode, PM expands the main memory capacity without providing persistence, while DRAM acts as a direct-mapped write-back cache (with a block size of 4 KB) for the PM modules. This cache mechanism is transparent to the user, allowing the system to automatically manage the flow of data between DRAM and PM for optimized performance. **App-directed Mode:** In contrast, app-directed mode treats PM as a separate byte-addressable persistent memory that is outside the traditional main memory hierarchy. Unlike memory

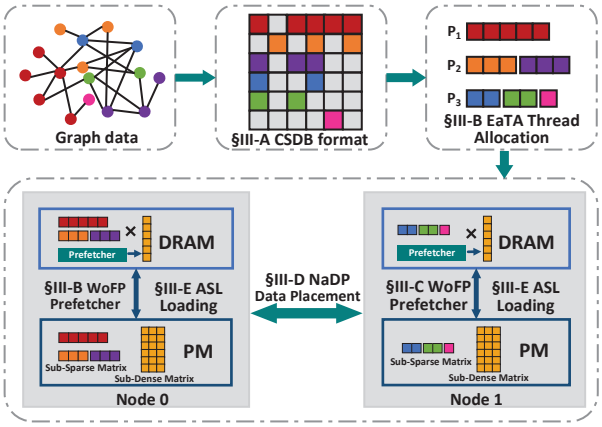

Fig. 4: The workflow of OMeGa.

mode, there is no DRAM cache in this configuration. Instead, a file system is used to manage the device, and applications can directly access the PM using load and store instructions, while employing ordering facilities to enforce consistency and ensure crash recovery [17]. The app-directed mode gives applications the ability to selectively store data in either DRAM or PM, offering fine-grained control over where data is placed. This configuration is particularly beneficial for workloads that require explicit control over memory allocation or need to store large datasets persistently [47]. In this work, we focus on using the PM in the App-directed mode and compare our optimizations with PM-only and DRAM-only environments, respectively (§IV).

III. Design of OMeGa

We design an efficient and scalable graph embedding framework under the emerging DRAM-PM hybrid storage architecture, called OMeGa, aiming to provide large capacity and high performance heterogeneous memory processing. Figure 4 summarizes our proposed OMeGa workflow. Initially, the graph data is represented by *compressed sparse degree-block* (CSDB, §III-A) format. To launch *SpMM* in graph embedding, OMeGa first utilizes an *entropy-aware thread allocation* scheme (EaTA) to adaptively generate the optimal workload for each thread (§III-B). Subsequently, a *NUMA-aware data placement* (NaDP) minimizes the detrimental effects of NUMA (§III-D). During streaming heterogeneous memory processing, OMeGa introduces a *workload feature-aware prefetcher* (WoFP, §III-C) to avoid unnecessary data movement and *asynchronous adaptive streaming loading* (ASL, §III-E) to support streamlined execution.

A. Compressed Sparse Degree-Block Format

Existing graph data structures fall short in reaching high data scalability. Given an example graph ($|V|$=7, $|E|$=11) shown in Figure 5(a), the node list is $[v_0, v_1, v_2, v_3, v_4, v_5, v_6]$, the column list ($col\_list$) in a matrix which is constructed by graph indicates the corresponding edge list, and edge weight list (i.e., non-zero data list in a matrix, nnz_list) is assigned by the weight of each edge (initially set to 1). Although the

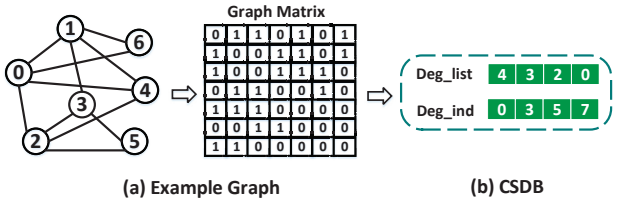

Fig. 5: The compressed graph data format CSDB in OMeGa.

widely used CSR [48] can represent the graph data well in some cases, the size of required indices increases proportionally with the number of nodes in a graph (i.e., $\mathcal{O}(|V|)$).

Considering the skewness in the degree distribution of real-world graphs, OMeGa introduces a new compressed graph data representation, the *compressed sparse degree-block* (CSDB) format, designed to efficiently handle this structural imbalance. CSDB uses two index arrays, Deg_list and Deg_ind, as shown in Figure 5(b), to represent the graph structure in a more compressed way. Specifically, Deg_list denotes the unique degree list for a graph, while Deg_ind represents the starting offset of the node block with the same node degree in the degree list. The sizes of Deg_list and Deg_ind are far smaller than the number of nodes in real-world graphs (i.e., $\mathcal{O}(|Degree|)$). For the example graph in Figure 5(a), Deg_list and Deg_ind in CSDB are $[4, 3, 2, 0]$ and $[0, 3, 5, 7]$, respectively. Suppose that we need to access the neighbors of node v_1 (get all non-zero elements of the row 1 in the adjacency matrix). We note that the index of v_1 is 1, with node degree block index $\in [Deg\_ind(0), Deg\_ind(1))$. Then, according to $Deg\_list(0)$, the degree of v_1 is 4 (i.e., $Degree(v_1)$). The starting offset Deg_ptr in col_list or nnz_list for v_i can be defined as:

$$Deg\_ptr(v_i) = \sum_{i=1}^{i} Degree(v_{i-1}) \tag{1}$$

Then, $Deg\_ptr(v_1)$ is 4, and thus the neighbors of v_1 can be generated from col_list by $Deg\_ptr(v_1)$ and $Degree(v_1)$ as $[v_0, v_3, v_4, v_6]$, as well as getting the weight of the corresponding edge from nnz_list.

To ensure compatibility with matrix calculations, we have also developed various operators based on CSDB, encompassing multiplication, addition, subtraction, and transposition. Given the sequential access pattern during matrix computations, the lightweight CSDB format ensures high performance with minimal overhead (§IV-I).

B. Entropy-aware Thread Allocation

Sparse Matrix and Dense Matrix Multiplication (*SpMM*) represents fundamental operations in graph embedding. For instance, in ProNE, *SpMM* is executed in parallel and accounts for nearly 70% of the total overhead. Specifically, the sparse matrix is composed of the graph data with V as the node set and E as the edge set, while the dense matrix is initially randomly generated with a $|V| \times |V|$ size, and its elements are iteratively updated in subsequent calculations.

Cost analysis for the parallel *SpMM*. Thread allocation is a critical step in parallel computing which involves assigning threads to process the components of a computation task

Algorithm 1 The execution of *SpMM* at a allocated workload

Input: sparse matrix A, dense matrix B, block start bst, row start rst, row end red, result matrix C
Output: result matrix C

```
1: for column t in B do
2:     ind = bst // Initial index for the workload allocated from A.
3:     for row j in [rst, red) do
4:         start = ind, end = ind + A.degree[j] //❶
5:         tmp = 0 // Intermediate result for C.
6:         while start ≠ end do
7:             get_sparse_nnz = A.nnz_list[start] //❷
8:             get_dense_nnz = B(A.col_list[start], t) //❸
9:             tmp += get_sparse_nnz × get_dense_nnz //❹
10:            start++
11:        C(j, t) = tmp //❺
12:        ind += A.degree[j]
```

in optimizing the performance and resource utilization. The parallel *SpMM* in our scenarios involves assigning rows of a graph matrix to different threads and multiplying by each column of a dense matrix. Given a graph matrix A, represented by our proposed graph data format CSDB (detailed in §III-A), the column list (col_list) in a matrix, constructed from the graph, indicates the corresponding edge list, while the edge weight list (i.e., non-zero data list in a matrix, nnz_list) is assigned by the weight of each edge, initially set to 1. Algorithm 1 shows the execution of a portion of the overall workload at a thread, where rst and red determine the rows allocated for *SpMM*. The cost at one thread can be considered as the total overhead of ❶ $read\_index$ (Line 4), ❷ get_sparse_nnz (Line 7), ❸ get_dense_nnz (Line 8), ❹ $accumulation$ (Line 9) and ❺ $write\_result$ (Line 11). Since *SpMM* works based on the indices of the non-zero elements (nnz) in the sparse matrix represented by CSDB format, ❶ and ❷ are sequential operations. However, due to the sparsity of A, the corresponding elements fetched in the dense matrix B will have discontinuous row indices (i.e., $A.col\_list[start]$). As a result, the throughput of ❸ cannot sufficiently utilize the sequential bandwidth, leading to the risk of random access occurrence. Similar to the dense matrix B, the resulting matrix C is stored in column-major order by default, which enables the sequential updating of elements in ❺. The cost of ❹ is dependent on both the allocated workload and the physical CPU throughput. Therefore, suppose a workload W_i is allocated to a thread p_i, where W_i is defined as the set of all nnzs in a subset of rows ($Rows_i$) from the sparse matrix A, then the overhead at p_i can be further derived as:

$$T(p_i) = \underbrace{\frac{Rows_i}{BW_{r\_seq}}}_{❶} + \underbrace{\frac{W_i}{BW_{r\_seq}}}_{❷} + \underbrace{\frac{W_i}{BW_{r\_seq} \times W^i_{sca}}}_{❸} + \underbrace{\frac{W_i}{BW_{CPU}}}_{❹} + \underbrace{\frac{W_i}{BW_{w\_seq}}}_{❺}, \tag{2}$$

where BW_{r_seq}, BW_{w_seq}, and BW_{CPU} are the bandwidth of sequential read, sequential write, and CPU cumulative, respectively. W^i_{sca} is the inherent scatter factor of W_i, determined by the scattered degree of column indices in the sparse matrix. W^i_{sca} captures the discontinuity of corresponding row indices (i.e., $A.col\_list[start]$) from the dense matrix, affecting the throughput of ❸ get_dense_nnz operation. W^i_{sca} can be defined as the average number of non-zero element indices per row in the allocated workload divided by $|V|$ (i.e., total

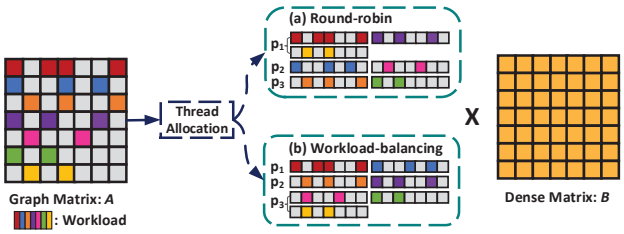

Fig. 6: (a) Round-robin and (b) workload-balancing thread allocation scheme for the graph G on 3 threads.

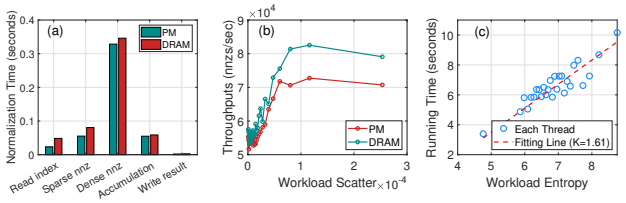

Fig. 7: (a) Execution time breakdown of *SpMM*, (b) throughputs and workload inherent scatter factor of ❸ get_dense_nnz, and (c) the relationship between running time and the workload entropy.

number of columns). The smaller the value of W^i_{sca}, the more scattered the access to the contiguous data of the dense matrix stored on the chip, reducing the computational efficiency.

Considering the thread allocation scheme, the widely used parallel development kit adopts the round-robin manner to allocate the workload to each thread by default, as shown in Figure 6(a), where p_1, p_2 and p_3 are assigned workload of 9, 5, 5, respectively. However, the irregularity of the matrix leads to an unbalanced workload across threads and naturally overlooks the inherent scatter factor of the workload. Figure 6(b) shows another common solution that applies a workload-balancing thread allocation strategy (WaTA) to let each thread have the same workload (W_i) [49], where the workload refers to the number of matrix elements of the sparse matrix A allocated to a thread. Formally, the WaTA scheme assigns $\frac{total\_workload}{\#threads}$ to each thread, such that in Figure 6(b), p_1, p_2 and p_3 are assigned workloads of 7, 6, and 6, respectively. However, the inherent scatter factor of the workload may lead to high tail latency (i.e., a long completion time of the last straggler thread). To explain, Figure 7(a) displays the cost analysis of each operation in *SpMM*, clearly observing that the ❸ get_dense_nnz dominantly affects the efficiency of parallel multiplications. Since each assigned W_i on WaTA may have different W^i_{sca}, the efficiency for each workload can be distinct, and thus WaTA may not guarantee the balanced-computing among threads (refers to Figure 13(a)). Figure 7(b) further exhibits the throughput and the workload inherent scatter factor of the ❸ get_dense_nnz for different threads (plotted as circles) by WaTA on PM and DRAM, and recall that the throughput is the amount of nnzs fetched per second. It can be observed that both curves have the same trend, and the more inherently scattered the workload, the lower the thread's bandwidth, resulting in higher tail latency due to insufficient utilization of sequential bandwidth caused by the occurrence of random accesses. In addition, the inherent scattered factor W^i_{sca} also plays a significant role in determining the number of rows that are allocated in W_i (i.e., $Rows_i$ in Equation 2). When the W_i among threads is fixed, the more scattered the indices, the more rows are allocated to W_i (i.e., $Rows_i \propto \frac{W_i}{W^i_{sca}}$), and hence the cost of the ❶ $read\_index$ operation can be approximated as $\frac{W_i}{BW_{r\_seq} \times W^i_{sca}}$.

Our proposed solution. To consider both workload balance and tail latency reduction, OMeGa incorporates a novel entropy-aware thread allocation (EaTA) scheme that first measures the sparsity inherent in the matrix based on the entropy theory [31], and then adaptively generates the optimal workload for each thread. Entropy is typically used to measure the degree of random access communication [50] or quantify the property in the random walk process on graph [51]. In our case, a smaller W^i_{sca} indicates more random accesses are introduced in the allocated workload W_i. As a result, the throughput of p_i used to fetch the indices or nnz (i.e., ❸ get_dense_nnz) will be reduced from sequential bandwidth (BW_{r_seq}) to random bandwidth (BW_{r_rand}). Figure 7(c) exhibits the relationship between running time ($T(p_i)$) and the workload entropy (H_i) for each thread, where the dashed red line is fitted by the least square loss with a slope K. We observe that the two variables show a strong linear relationship represented as $T(p_i) = K \cdot H_i$, where H_i is:

$$H_i = \sum_{j=n}^{m} -\frac{|Row_j|}{W_i} \log \frac{|Row_j|}{W_i} \tag{3}$$

n and m are the starting and ending indices of W_i (i.e., rst and red in Algorithm 1) in sparse matrix A, respectively. $|Row_j|$ denotes the number of non-zero elements (nnz) in Row_j.

Based on our previous analysis, W^i_{sca} is a key factor affecting computational efficiency (Figure 7(a) and (b)), i.e., for a thread p_i, $T(p_i) \sim \frac{W_i}{BW_{r\_seq} \times W^i_{sca}}$, and hence the measurement of $T(p_i)$ can be further quantified as:

$$\frac{W_i}{BW_{r\_seq} \times W^i_{sca}} = K \cdot H_i \tag{4}$$

In term of access impact, the relation of H_i and W^i_{sca} is:

$$W^i_{sca} = 1 - Z(H_i) + \beta \cdot Z(H_i) \tag{5}$$

where H_i is normalized by $Z(H_i) = \frac{H_i}{log|V|}$ ($0 \le H_i \le log|V|$), and β represents $\frac{BW_{r\_rand}}{BW_{r\_seq}}$. Intuitively, for the term $BW_{r\_seq} \times W^i_{sca}$, if $Z(H_i)$ is close to 1, it results in a large number of random accesses, with the bandwidth determined by BW_{r_rand}. Conversely, when $Z(H_i)$ approaches 0, the access pattern becomes fully sequential, and the bandwidth corresponds to BW_{r_seq}. From Equations 4 and 5, W_i can be expressed as:

$$W_i = K \cdot H_i \cdot BW_{r\_seq} \cdot (1 - Z(H_i) + \beta \cdot Z(H_i)) \tag{6}$$

The optimal workload W^p_i for each thread p_i is generated by the optimized workload-balancing scheme with the entropy-aware measurement as follows:

$$\frac{W_i}{W^p_i} = \frac{H_i \cdot (1 - Z(H_i) + \beta \cdot Z(H_i))}{H^p_i \cdot (1 - Z(H^p_i) + \beta \cdot Z(H^p_i))}$$
$$\implies W^p_i = W_i \cdot \frac{H^p_i \cdot (1 - Z(H^p_i) + \beta \cdot Z(H^p_i))}{H_i \cdot (1 - Z(H_i) + \beta \cdot Z(H_i))} \tag{7}$$

Equation 7 adjusts the initial workload W_i based on the entropy measure H_i and its optimized version H_i^p for each thread. By leveraging both H_i and H_i^p, this scheme adapts the workload to reduce tail latency, ensuring that threads with higher randomness (larger $Z(H_i)$) have their allocated workloads adjusted more, achieving both balance and efficiency.

Implementation of EaTA. Algorithm 2 shows the procedure for our proposed solution. Here, EaTA gets W_i as the initial workload by a dynamic workload-balancing scheme ($\frac{unallocated\_Workload}{unallocated\_threads}$) and then derives H_i based on Equation 3 (Lines 2, 4 and 11). To reduce the tail latency brought by the inherent scatter factor in the workload, the inherent scatter factor of each workload should be relatively close, and hence EaTA leverages a dynamic average H_i^p among workloads ($\frac{allocated\_H_i^p}{allocated\_threads}$) to calculate the optimal workload W_i^p allocated to each thread based on Equation 7 (Lines 6, 9 and 12). Initially, H_i^p is defined as the average entropy for a given number of threads, i.e., $\sum_{j=0}^{|V|} -p_j \log(p_j)$, where $p_j = \frac{|Row_j|}{|V|/threads\_num}$ (Line 2). EaTA operates online as an efficient lightweight scheme with $\mathcal{O}(|V| \cdot W_i)$ complexity.

Algorithm 2 Entropy-aware thread allocation scheme in OMeGa

Input: $threads\_num$, total_workload W_total, sparse matrix A
Output: $W_{pos}(rst, bst)$ //Starting index of row and element in A.
1: $i = 1$, $rst = 0$, $bst = 0$
2: $W_i = \frac{W\_total}{threads\_num}$, H_i^p//Initialize the W_i and H_i^p.
3: **while** $rst < A.|V|$ **do**
4: Get the H_i according to the W_i //Eq. 3
5: Generate rst and bst of W_i
6: $W_i^p = W_i \cdot \frac{H_i^p \cdot (1 - Z(H_i^p)) + \beta \cdot Z(H_i^p))}{H_i \cdot (1 - Z(H_i)) + \beta \cdot Z(H_i))}$ // Eq. 7
7: Update rst and bst according to W_i^p
8: Generate $W_i^{p\prime}$ and $H_i^{p\prime}$ based on the updated rst
9: W_i_tmp+=$W_i^{p\prime}$, $H_i^p\_tmp$+=$H_i^{p\prime}$, i++
10: $W_{pos}.push\_back(rst, bst)$//Write back rst and bst of W_i to W_{pos}.
11: $W_i = \frac{W\_total - W_i\_tmp}{threads\_num - i}$ //Update the initial W_i.
12: $H_i^p = \frac{H_i^p\_tmp}{i}$ //Update the objective H_i^p.

C. Workload Feature-aware Prefetcher

While EaTA addresses workload balancing and tail latency, the data movement induced by random access continues to impede the efficiency of graph embedding, especially in heterogeneous memory scenarios with performance gaps. To this end, OMeGa further introduces the *workload feature-aware prefetcher* (WoFP) on top of EaTA to optimize memory access patterns within each allocated workload.

Basic idea of WoFP. Conventional hardware-managed caches are limited in capacity and respond passively to access patterns, which may result in frequent PM accesses when handling large-scale graph workloads. WoFP identifies the frequently accessed data objects during *SpMM* operation based on workload features and places them in DRAM, thereby minimizing unnecessary data movement during the streaming processing between DRAM and PM. As analyzed in Algorithm 1, the sparsity of the graph matrix A results in discontinuous row indices for the corresponding elements fetched in the dense matrix B (i.e., $A.col\_list[start]$), leading

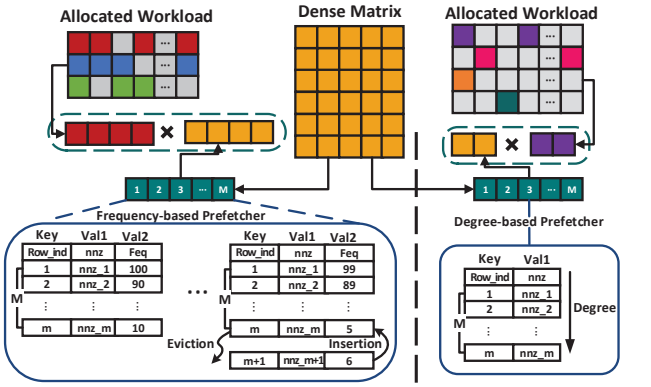

Fig. 8: WoFP performs prefetching based on frequency (left panel) and in-degree (right panel), respectively.

to random access and thus decreased memory access performance. Since each column t in B needs to be multiplied by every row j in the workload allocated from A, there is a possibility of reusing the $B(A.col\_list[start], t)$ in each loop. In light of this locality, WoFP tailors the prefetcher for each allocated workload, optimizing memory access without altering the workload distribution determined by EaTA, thereby enhancing execution efficiency without introducing imbalance.

As shown in Figure 8, WoFP utilizes a hashmap structure to store the identified frequently accessed elements. Specifically, it uses row indices ($A.col\_list[start]$) as keys and pairs them with corresponding elements from the dense matrix ($B(A.col\_list[start], t)$) along with the prefetching measurement values as values, thereby constructing a key-value mapping. In each loop iteration, it selects the *top-M* frequently accessed key-value pairs to be placed in DRAM, enabling rapid retrieval from the prefetcher during parallel *SpMM* operations, thus mitigating random accesses and reducing unnecessary data movements in the DRAM-PM streaming processing.

Implementation of WoFP. To improve the prefetching efficiency, WoFP introduces a hybrid prefetcher based on the feature of workload allocated by EaTA. For workloads with a higher average number of row indices, WoFP employs prefetching based on the frequency of row index occurrences within the loop (frequency-based prefetcher). For workloads with a lower average number of row indices, it utilizes the in-degree of the vertex (i.e., the number of non-zero element indices per column in A) as the metric (degree-based prefetcher). This is because a higher in-degree suggests a higher probability that the associated row indices are the same. Given that real-world graphs usually adhere to a power-law degree distribution, most workloads have fewer row indices on average, and it is more convenient to count in-degrees.

The specific execution of WoFP is as follows: given the allocated workload for a thread as W_i, WoFP initially determines the type of prefetcher to employ based on the condition defined as $\frac{W_i}{Rows} \geq |V| \cdot \eta$, where $Rows$ represents the number of rows in the allocated workload, $|V|$ denotes the size of the vertex set (i.e., the total rows of the graph matrix A), and η is an empirical parameter used to establish

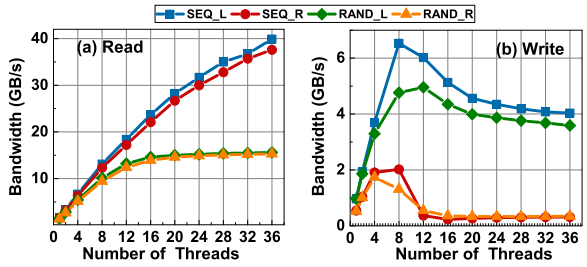

Fig. 9: The sequential (SEQ) / random (RAND) read and write bandwidth of local (L) / remote (R) PM for the different #threads.

the prefetcher type selection criterion. This judgment condition $|V| \cdot \eta$ means the reference base for the number of indices in each row, and it can be adjusted adaptively according to the size of the graph dataset. If the allocated workload meets the above condition, WoFP selects the frequency-based prefetcher, otherwise, it uses the degree-based prefetcher.

Subsequently, WoFP constructs the prefetcher with a size of $M = W_i \cdot \sigma$ for each allocated workload, where σ denotes the prefetching size parameter. It accommodates the frequently accessed objects in the *top-M* key-value structure. Different from the degree-based prefetcher, which statically utilizes the descending in-degree of the vertex to populate the prefetcher (i.e., key: $A.col\_list[start]$, value: $B(A.col\_list[start], t)$), the frequency-based prefetcher dynamically selects the *top-M* most frequently used row indices for prefetching. It counts the frequency of row index occurrences during the execution of the *SpMM* in a back-end thread, and updates them together with the corresponding element values as the values into the *top-M* prefetcher (i.e., key: $A.col\_list[start]$, value1: $B(A.col\_list[start], t)$, value2: $frequency$). Although dynamic prefetcher entails eviction and insertion operations for objects in the *Top-M*, as aforementioned, only a small portion of workloads necessitates such prefetching, and the benefits of placing them in high-performance DRAM are substantial (§IV-D).

D. NUMA-aware Data Placement

Non-uniform memory access (NUMA) is a shared memory system that aims to overcome the poor scalability of the traditional uniform memory access system architecture. However, the performance gap between DRAM and PM is further amplified under the NUMA architecture, which is undesirable for large-capacity and high-performance heterogeneous memory.

Observational analysis of the access characteristics for PM. Figure 9 reports the sequential/random read and write bandwidth of local/remote PM for different numbers of threads on our test system (3 PMs and 18 CPU cores per NUMA node). The results are collected by the NUMACTL [52] and FIO [53] tools. As shown in the results, regarding the read operations, the peak bandwidth of sequential remote accesses is comparable to that of local sequential and is much higher than that of random local and random remote ones (2.41× and 2.45×, respectively). This finding is in line with the previous study conducted on DRAM-based systems [54]. For the write operations, regardless of whether they are sequential

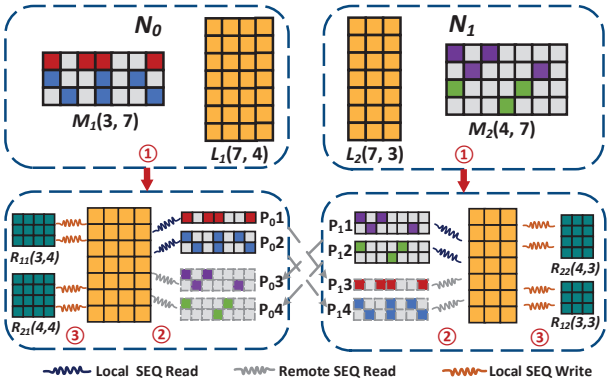

Fig. 10: NUMA-aware data placement on two-socket nodes.

or random, local write operations are always better than those remote ones. The peak bandwidth of the remote PM write is decreased to 69.2% compared with the local ones. Especially, the peak bandwidth of sequential local writes is greater than sequential remote and random remote ones by 3.23× and 4.99×, respectively. More importantly, we also note that the remote and local read latencies are 3.3× and 4.2× higher than the DRAM-base system by the Intel Memory Latency Checker [55], respectively. Based on these observations, we conclude that a high-performance DRAM-PM system should prioritize global sequential read and local write accesses, thereby minimizing the adverse impact of NUMA.

Running characteristics of the operator. Although EaTA (§III-B) and WoFP (§III-C) improve the efficiency of parallel *SpMM* for graph embedding generation, the performance is still bounded by the impact of remote access under the NUMA architecture. We statistically analyze the local and remote accesses (including read and write operations) of OMeGa with components of EaTA and WoFP on two sockets (using 30 threads), collected using the Intel VTune Profiler [56]. It can be found that the portion of the average remote access is more than 43% on our evaluated graphs. Given that the remote latency and local latency on PM are higher than the DRAM-base system, respectively, the efficiency of *SpMM* operation on PM is 4.5× slower than DRAM.

In addition, from Algorithm 1, the indices in sparse matrix represented by our proposed CSDB (detailed in §III-A), such as Deg_list, Deg_ind, col_list and nnz_list, are sequentially incorporated to perform the get_sparse_nnz operation in *SpMM* (Line 3-9). Although the get_dense_nnz operation is affected by the non-continuous row indices (Line 8), EaTA and WoFP schemes can mitigate the impact and ensure the balanced-computing among threads. For the intermediate data, such as the temporary value that is written to the result matrix (Line 11), it is a sequential write operation due to the column-major order stored organization for the result matrix. Therefore, OMeGa takes into consideration both the hardware features and the running characteristics of the operators to design a NUMA-aware data placement (NaDP) solution.

Basic idea of proposed NaDP. Existing OS-provided NUMA allocation policies [57], such as Local (allocating memory

on a specified node and using other nodes if the preferred node has insufficient memory) and Interleaved (interleaving memory allocation across nodes in a round-robin fashion), which do not consider the adverse impacts of PM and lack fine-grained NUMA-aware allocation to meet the application-specific requirements. In contrast, our proposed approach, NaDP, achieves the NUMA access principle of **global sequential read and local write** by manually allocating memory within the application and using the binding-threads on different sockets to touch the memory pages.

The fundamental idea behind NaDP consists of three main aspects: (1) ***NUMA-aware memory allocation:*** The sparse and dense matrices are partitioned according to the number of socket nodes to improve data locality and balance the workload, which is essential for efficient parallel execution. (2) ***CPU-binding based computing:*** Threads are bound to specific CPUs, enabling them to sequentially fetch sub-sparse matrices, either locally or remotely, and perform computations on the locally stored sub-dense matrices, ensuring global sequential access. (3) ***Local-priority based updating:*** Intermediate data is written into local two-dimensional arrays constructed for each sub-matrix multiplication, ensuring efficient local write operations. As explained earlier in §III-B, the get_sparse_nnz operation is strictly sequential, ensuring sequential reads across socket nodes, regardless of whether the access is local or remote. This guarantees global sequential reading. At the same time, the write operations remain local and sequential, as the intermediate matrices are stored locally on each socket node, with temporary values sequentially written to the result matrices.

Figure 10 exhibits an example of the NUMA-aware data placement (NaDP) on two-socket nodes (N_0 and N_1) during *SpMM* operation. Given a sparse matrix $M(7,7)$ and a dense matrix $L(7,7)$, ❶ NaDP partitions M and L as $M_1(3,7)$, $M_2(4,7)$, $L_1(7,4)$ and $L_2(7,3)$, respectively, where M_1 and L_1 are placed on N_0, and M_2 and L_2 are placed on N_1. To hold the intermediate data, NaDP also constructs $R_{11}(3,4)$ and $R_{21}(4,4)$ on N_0, and $R_{22}(4,3)$ and $R_{12}(3,3)$ on N_1, respectively. ❷ Based on EaTA, suppose that the number of threads is 8 ($P_0 1, P_0 2, P_0 3$, and $P_0 4$ are bound to N_0, and $P_1 1, P_1 2, P_1 3$, and $P_1 4$ are bound to N_1), then for the $M_1 \times L_1$ operation, $P_0 1$ and $P_0 2$ locally fetch the non-zero element (nnz) from M_1 in a sequential manner, followed by performing the multiplication with L_1. Additionally, WoFP is utilized to enhance processing efficiency. ❸ Meanwhile, the intermediate data are sequentially written into R_{11}. For the $M_2 \times L_1$ operation, $P_0 3$ and $P_0 4$ remotely fetch the nnz of M_2 from another node N_1 in a sequential manner and then execute the multiplication operations with L_1 at N_0. We note that the results are also sequentially written into R_{21}. For the $M_2 \times L_2$ and $M_1 \times L_2$ operations, $P_1 1, P_1 2, P_1 3$, and $P_1 4$ use the same execution mechanism to complete the multiplication operations. Finally, NaDP simply combines the sub-matrices ($R_{11}, R_{21}, R_{22}, R_{12}$) across N_1 and N_0 to generate the result matrix $R(7,7)$. Although this step produces few remote accesses, NaDP minimizes the remote write operation in the above calculation procedure.

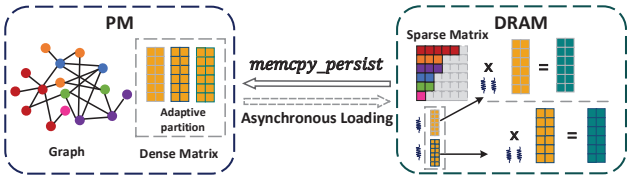

Fig. 11: Asynchronous adaptive streaming loading for OMeGa.

E. Implementation of OMeGa

Alongside the aforementioned optimizations, OMeGa further introduces systematic enhancements *asynchronous adaptive streaming loading* (ASL) on heterogeneous memory.

Considering that the dense matrices and intermediate data generated during the graph embedding process far exceed the scale of the sparse matrix composed of graph data, making it challenging to store them directly in DRAM, the data needs to be streaming loading between DRAM and PM. To address this, OMeGa introduces the asynchronous adaptive streaming loading (ASL) strategy on top of the NaDP proposed in §III-D, further enhancing heterogeneous memory processing with large capacity and high performance.

Specifically, as shown in Figure 11, to maximize the utilization of the high-performance DRAM's processing capabilities, ASL adaptively divides the dense matrix data allocated to each NUMA node based on available memory capacity. Subsequently, it asynchronously loads batches of data from PM to DRAM for computation. For determining the granularity of the streaming data adaptation, ASL analyzes the peak memory consumption during running time, as follows:

$$M_l + M_{al} + M_s + M_r + M_{ri} + M_{li} \leq M_{total} \tag{8}$$

Here, M_l, M_s, and M_r represent the memory size of the streaming data, the sparse matrix, and the results matrix, respectively. M_{ri} and M_{li} denote the memory size of intermediate results for M_r and M_l, respectively. M_{al} refers to the asynchronous streaming data. M_{total} represents the available capacity of DRAM. Suppose the partitions for the dense matrix are denoted by n, and the dimension of the matrix is represented by d (also known as the dimensional space of the generated embedding). Then, the memory sizes M_l, M_{al}, and M_{li} can be expressed as $\frac{d}{n} \cdot |V| \cdot size(type)$. M_r and M_{ri} are $d \cdot |V| \cdot size(type)$, and M_s corresponds to the memory footprint of our proposed CSDB format. Here, $size(type)$ refers to the data type. The optimal n can be derived as:

$$n \geq \frac{3d \cdot |V| \cdot size(type)}{M_{total} - M_s - 2d \cdot |V| \cdot size(type)} \tag{9}$$

IV. Experimental Results

We evaluate the efficiency of our proposed framework, OMeGa, by comparing it against various baselines, including OMeGa-DRAM (ideal baseline), OMeGa-PM (the worst baseline), ProNE-DRAM [10], heterogeneous memory-based ProNE (ProNE-HM), Ginex [14] and MariusGNN [58]. We also analyze the contributions of key optimizations, such as

TABLE I: Datasets statistics.

Graph	#nodes	#edges	#degrees
PK	1.63 M	44.60 M	803
LJ	4.85 M	85.70 M	1 641
OR	3.07 M	234.47 M	2 863
TW	11.32 M	127.11 M	5 373
TW-2010	41.65 M	2.41 B	15 760
FR	65.61 M	3.61 B	3148

TABLE II: Running time of EaTA and competitors for performing *SpMM*.

Graph	RR	WaTA	EaTA
PK	16.23 s	3.76 s	**2.16 s**
LJ	36.52 s	10.15 s	**7.12 s**
OR	77.60 s	24.27 s	**18.91 s**
TW	40.17 s	7.43 s	**7.17 s**
TW-2010	1565.38 s	316.95 s	**295.29 s**
FR	16566.25 s	2530.97 s	**2432.11 s**

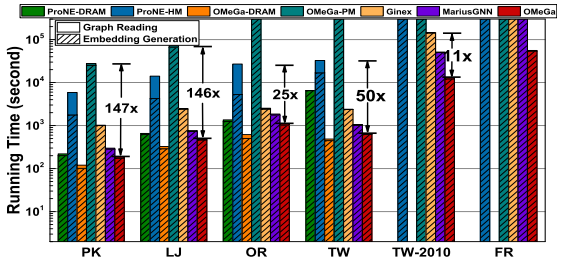

Fig. 12: Overall performance of OMeGa and competitors.

CSDB (III-A), EaTA (§III-B), WoFP (§III-C), and NaDP (§III-D). Additionally, we evaluate OMeGa's scalability, demonstrate its potential as an alternative to distributed frameworks (DistGER [11] and DistDGL [12]), compare its *SpMM* performance against state-of-the-art methods (SEM-SpMM [59] and FusedMM [60]), and analyze its parameter sensitivity. Our codes and datasets are available at [61].

A. Experimental Setup

Environment. We conduct experiments on two-socket Optane DIMM [1] with 2.60GHz Intel(R) Xeon(R) Gold 6240 CPU (36 cores, hyper-threading). Each socket is equipped with 96GB DRAM (3×32GB) and 768GB PM (3×256GB). Additionally, the machine has a 3.84TB Intel P5510 NVMe SSD.

Datasets. We use six widely-used real-world graphs: *soc-Pokec* (PK) [62], *soc-LiveJournal* (LJ) [63], *Com-Orkut* (OR) [64], *Twitter* (TW) [65], *Twitter-2010* (TW-2010) [66], and *Com-Friendster* (FR) [67]. These datasets span both sparse and dense graph structures, as summarized in Table I. To further evaluate scalability across different graph densities, we also use synthetic graphs generated by an RMAT generator [68].

Baselines. Besides demonstrating the merit of OMeGa, we first compare it against the same type graph embedding system ProNE [10] on DRAM (ProNE-DRAM) and heterogeneous memory (ProNE-HM, a DRAM-PM implementation, where matrix operations are handled on DRAM), respectively, to show our key improvements. We also implement two baselines for OMeGa, one is on top of DRAM-only (OMeGa-DRAM) which is served as the ideal baseline, whereas the other is on top of PM-only (OMeGa-PM) to show the performance comparison between DRAM and PM. We further benchmark OMeGa against recent SSD-based graph embedding systems, Ginex [14] and MariusGNN [58] (both utilize NVIDIA V100 GPU), as well as distributed systems, DistGER [11] and DistDGL [12], emphasizing the advantages of heterogeneous memory processing. Finally, we evaluate OMeGa's *SpMM* performance against state-of-the-art *SpMM*-oriented systems, including the SSD-based SEM-SpMM [59] and the in-memory-based FusedMM [60].

B. Overall Performance

Figure 12 gives a comprehensive running time comparison between OMeGa and the other six alternatives on six real-world graphs. The reported executed time includes the running time of the graph reading and embedding generation procedure. The results show that OMeGa significantly outperforms the competitors on all these graphs, except for OMeGa-DRAM due to it serving as our ideal baseline. Overall, OMeGa achieves an average acceleration of 32.03×. Note that we fail to run the DRAM-only based systems ProNE-DRAM and OMeGa-DRAM on the billion-scale graph datasets (*TW-2010* and *FR*) because the inherent memory-dependency of the models is the bottleneck of their performance.

Recall that OMeGa is a specialized design on heterogeneous memory for graph embedding and our key improvements are discussed in §III. Compared with ProNE-DRAM and ProNE-HM (where *SpMM* constitutes nearly 70% of the total overhead), Figure 12 shows that our system OMeGa achieves an average speedup of 3.45× and 33.65×, respectively. Even for the DRAM-only environment, OMeGa-DRAM is also on average 4.99× faster than ProNE-DRAM. Due to the performance gap between DRAM and PM, OMeGa-PM is on average 146.67× slower than OMeGa on *PK* and *LJ* graphs. For the remaining larger-scale graphs, it does not terminate in one day. Compared to the DRAM-PM implemented ProNE-HM, OMeGa-PM performs all operations in a PM-only setting, resulting in worse performance. OMeGa aims to achieve the large-capacity and high-performance heterogeneous processing, compared to OMeGa-DRAM and OMeGa-PM, it significantly narrows the performance gap between PM and DRAM from orders of magnitude scale to an average of 54.9%.

Additionally, compared to state-of-the-art NVMe SSD-based graph embedding systems, our proposed OMeGa, operating on a DRAM-PM heterogeneous memory architecture, achieves an average speedup of 5.49× over Ginex [14] and 2.07× over MariusGNN [58] with both competitors requiring over a day to process the largest graph *FR*. These results demonstrate that, despite their powerful GPU acceleration, SSD-based systems are bottlenecked by high I/O overhead (e.g., SSD latency and bandwidth limitations), further underscoring the benefits of heterogeneous memory. We also extend our evaluation by comparing OMeGa with distributed graph embedding systems (DistGER [11] and DistDGL [12]) and SpMM-oriented systems (SEM-SpMM [69] and FusedMM [60]), as detailed in the §IV-G and §IV-H. For the quality of embeddings, since OMeGa uses ProNE as the model prototype and provides system support on heterogeneous memory, it maintains the effectiveness of graph representation of ProNE.

[1]The only PM product accessible in this study, but OMeGa is expected to be equally effective in other PM products like CXL when available.

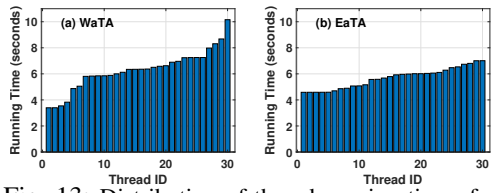

Fig. 13: Distribution of thread running times for OMeGa equipped with WaTA (a) and EaTA (b).

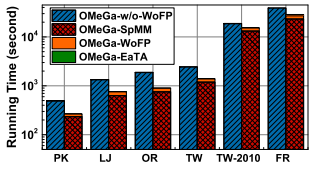

Fig. 14: Running time of SpMM with and without WoFP.

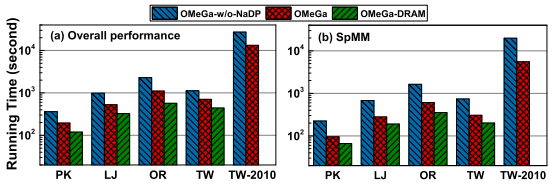

Fig. 15: The effect of NaDP for overall performance (a) and SpMM (b) on five real-world graphs.

C. Thread Allocation

To evaluate the performance of the proposed entropy-aware thread allocation scheme EaTA in OMeGa, we first compare the efficiency of EaTA with two alternatives: the Round-Robin (RR) and workload-balancing thread allocation (WaTA) for the parallel *SpMM*, which is an indispensable operation for graph embedding. Table II shows the execution time for each scheme to perform one *SpMM* operation. EaTA significantly outperforms RR and WaTA on all graphs, achieving a speedup ranging from 1.04× to 7.51×, with an average acceleration of 3.50×. RR is the default thread allocation scheme in the threads library. Although it is a simpler approach, it overlooks the skewness inherent in most real-world graphs, leading to an imbalanced workload distribution among the allocated threads. WaTA divides the workload evenly among the threads, and hence this scheme can alleviate the skewness problem of RR. Nevertheless, it still increases the risk of long tail latency since the inherent sparsity of the graph matrix.

In Figure 13, we present the distribution of thread running times on the *soc-LiveJournal* graph for OMeGa with WaTA (a) and EaTA (b), respectively. A comparison reveals that EaTA outperforms WaTA in reducing tail latency, with thread execution times becoming more evenly distributed, indicating a better alignment of workload distribution. Specifically, the standard deviation for EaTA is 0.78, much smaller than the 1.52 for WaTA. Furthermore, EaTA achieves a 31% reduction in P99 (99th percentile latency) and a 24% reduction in P95 (95th percentile latency) tail latency compared to WaTA. Our results show thread allocation overhead is negligible, under 1% of runtime (as shown in Figure 14). Overall, EaTA is a lightweight scheme with substantial benefits.

D. Prefecher Performance

Considering the inherent sparsity of the graph leads to numerous random accesses, OMeGa introduces the workload feature-aware prefetcher (WoFP) to achieve high-performance heterogeneous memory processing. To evaluate the effect of WoFP on OMeGa, we report the execution time of all *SpMM* operations with and without WoFP (denoted as OMeGa and OMeGa-w/o-WoFP, respectively) on top of our proposed EaTA thread allocation policy. Note that the reported execution time includes the overhead of thread allocation (EaTA), prefetching (WoFP), and *SpMM*. Figure 14 shows the efficiency of OMeGa and OMeGa-w/o-WoFP on all evaluated graph datasets. The results indicate that OMeGa achieves a 37.28% average performance improvement, with particularly significant gains on the *OR* graph, where our solution is 52% faster than OMeGa-w/o-WoFP. Additionally, we observed that the overhead introduced by EaTA and WoFP constitutes a minimal fraction of the total execution time, averaging less than 3.17% across all datasets. Given the significant performance improvements, these overheads are negligible and do not affect the overall efficiency of OMeGa. We further evaluated the impact of the parameters used in WoFP, as discussed in §IV-I.

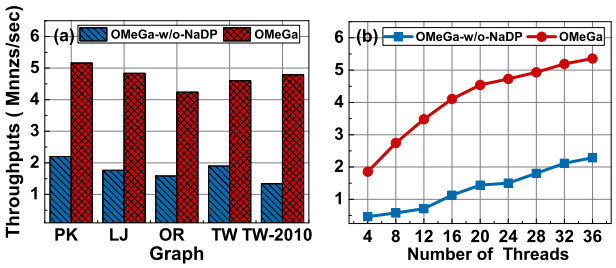

Fig. 16: Throughput of SpMM on five real-world graphs (a) and with different threads on *soc-LiveJournal* graph (b).

E. NUMA-aware Data Placement

Figure 15 exhibits the performance of NUMA-aware data placement (NaDP) for our proposed framework OMeGa. To access the effect of NaDP on improving the performance of PM for NUMA, we prepare a version of OMeGa without NaDP (using OS-provided Interleave NUMA allocation policy, known for its benefits in parallel computing [57]), denoted as OMeGa-w/o-NaDP. In Figure 15(a), we report the effect of NaDP for overall performance and compare it with the ideal baseline OMeGa-DRAM. As we reasoned in §III-D, the performance gap between DRAM and PM is further amplified under the NUMA architecture due to the slower PM accesses than DRAM. The results show that OMeGa-w/o-NaDP is on average 2.98× slower than OMeGa-DRAM. Hence, OMeGa takes into consideration both the hardware features and the running characteristics of the operator, aiming to minimize the adverse impact of NUMA to PM. As a result, OMeGa significantly outperforms OMeGa-w/o-NaDP on all these graphs, achieving an average speedup of 1.95×. Considering that parallel *SpMM* is directly influenced by the NUMA architecture, we further access the performance of NaDP on *SpMM* operation as shown in Figure 15(b). As an ideal baseline, although OMeGa-DRAM presents the best performance in most cases, it can not handle the billion-scale graph *Twitter-2010* due to memory limitation. Compared to OMeGa-w/o-NaDP, benefiting from NaDP, OMeGa can achieve an acceleration ranging from 2.42× to 3.59×, especially reducing the performance gap with OMeGa-DRAM to 40.17% on average. We also report the throughput of the *SpMM* operation

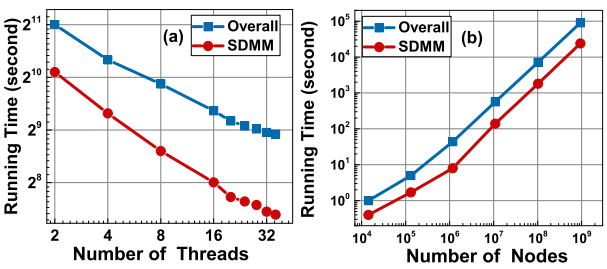

Fig. 17: The scalability performance of OMeGa with the increasing number of threads (a) and the size of a synthetic graph (b).

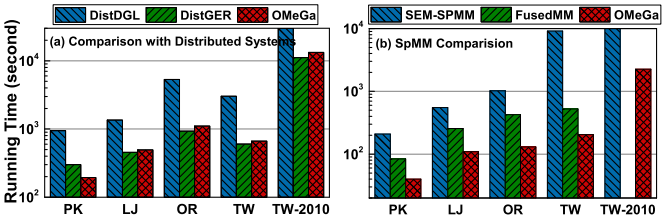

Fig. 18: Performance comparison of OMeGa with distributed graph embedding systems: DistGER [11] and DistDGL [12] (a), and with SpMM-optimized systems for *SpMM* operation execution: SEM-SpMM [69] and FusedMM [60] (b).

(million nnzs fetched per second) for OMeGa-w/o-NaDP and OMeGa on all graphs using 30 threads in Figure 16(a), as well as the throughput of the *SpMM* operation with different numbers of threads on the *soc-LiveJournal* graph in Figure 16(b). The results show that our proposed NaDP can better utilize the parallel capability of computational resources.

F. Scalability

Figure 17(a) shows the running time of overall performance (blue line) and *SpMM* (red line) for OMeGa on the *soc-LiveJournal* graph. (Due to space limitations, we omit results on other graphs, which exhibit similar trends). The results indicate that the running time decreases linearly with the number of threads. To further evaluate the scalability of OMeGa and its performance across diverse graph structures, we generate synthetic graphs using the RMAT generator [68]. RMAT enables control over both graph sparsity and density by adjusting parameters such as edge distribution and the average degree per node. The generated graphs exhibit varying structural properties, with node counts ranging from 10^4 to 10^9, reaching the billion-node scale. In Figure 17(b), we report the running times for overall performance and the *SpMM* operation on these synthetic graphs using 30 threads. The results confirm that OMeGa scales efficiently with graph size and performs robustly across both sparse and dense graphs.

G. Efficient Alternatives to Distributed Systems

To address the memory capacity limitations for large-scale graph embeddings, aside from the out-of-core systems, such as SSD-based Ginex, another conventional choice is to use distributed systems. We report the end-to-end running times of the recent distributed graph embedding systems DistGER [11] and DistDGL [12] on six real-world graphs, using a cluster of four machines (identical hardware to OMeGa but excluding PM), as shown in Figure 18(a). The reported end-to-end time includes both the graph reading and embedding generation procedures. OMeGa achieves comparable or superior performance in most cases. Specifically, it outperforms DistDGL on all datasets, with an average speedup of 4.31×, primarily due to the high overhead of graph sampling (accounts for approximately 80% of the runtime) and the synchronization overhead for gradient updates. While DistGER employs information-oriented random walks for efficient and scalable feature learning, outperforming DistDGL, OMeGa proves competitive by being 1.58× faster on the small graph *soc-Pokec* and delivering comparable performance on larger graphs. Due to the high financial costs and significant computational resources required by distributed setups, the heterogeneous memory-based OMeGa offers a cost-effective alternative to traditional distributed solutions.

H. OMeGa's SpMM w.r.t Competitors

Recognizing the critical role of *SpMM* in embedding generation, we benchmark OMeGa's *SpMM* performance against representative SpMM-optimized systems: the SSD-based SEM-SpMM [69] and the in-memory FusedMM [60]. In our evaluation, we measure the running time of a single *SpMM* operation across six real-world graphs, as shown in Figure 18(b). The results reveal that OMeGa consistently outperforms both systems, particularly on the larger graphs. SEM-SpMM addresses memory capacity limitations in large-scale graph *SpMM* operations by utilizing semi-external memory configurations. Compared to SEM-SpMM, OMeGa achieves an average speedup of 15.69×, thank to the superiority of heterogeneous memory architecture and the *SpMM* optimizations. While FusedMM consolidates *SpMM* operations into a single unified kernel for graph embedding, OMeGa delivers a speedup ranging from 2.11× to 3.26×. Notably, we were unable to run FusedMM on the billion-scale *Twitter-2010* graph due to memory limitations. A similar memory bottleneck is evident in GPU-based approaches, which prioritize efficiency optimizations. For example, the GPU-based Ge-SpMM [30] encounters out-of-memory issues with large graphs like *Twitter-2010* and *Com-Friendster*. In contrast, OMeGa successfully handles these billion-scale graphs, showcasing its large capacity and high performance via heterogeneous memory processing.

I. Graph Format and Parameter Sensitivity

We report the processing efficiency of CSDB (i.e. graph reading procedure) and compare with CSR [48] (default storage format of popular graph embedding approaches) in Figure 19(a). The results exhibit that CSDB not only carries out a light-weighted graph data representation, but also presents a 1.35× acceleration compared to CSR. In addition, we also investigate how different parameter choices in WoFP impact the performance of OMeGa, specifically focusing on prefetcher type selection (μ) and prefetching size (σ). We report the normalized execution time of OMeGa on *PK* graph, except for the tested parameters, all the other parameters are set

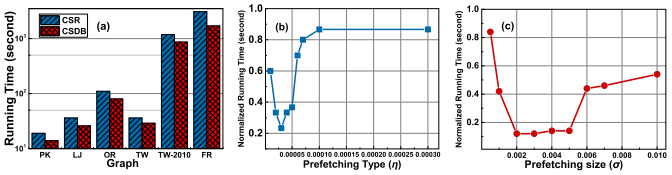

Fig. 19: The graph reading performance of OMeGa (a) and the parameter sensitive analysis for the prefetcher used in OMeGa (b).

as default in the experiments. η determines the prefetcher type selection in WoFP. As illustrated in Figure 19(b), when η is too small, most loads are directed to use frequency-based prefetcher, which incurs a relatively large overhead. However, as η increases, the preference shifts towards degree-based prefetcher. Nevertheless, ensuring the effectiveness of prefetching becomes challenging with higher η, leading to performance degradation. σ directly dictates the prefetching size (i.e. *top-M*). Similarly, Figure 19(c) demonstrates that as σ increases, the prefetcher becomes more active. However, an excessively large σ will escalate the prefetching costs.

V. Related Work

Typical graph embedding systems. Recently, several systems are developed from the CPU-GPU architecture perspective, such as Legion [70], MariusGNN [58], Seastar [71] and TGLite [72]. However, computing gaps between CPUs and GPUs and the limited memory of GPUs still plague the efficiency of graph embedding. Other approaches attempt to scale graph embeddings from a distributed perspective, such as AliGraph [73], ByteGNN [74], DistGNN [75], DistDGL [12], HET-KG [76], DistGER [11] and DistGER-Pipe [77]. Nonetheless, considering the high financial cost of consuming large amounts of computational resources, this is not consumer-friendly for the general public. There are also approaches attempting to address computational efficiency challenges with new hardware, such as GLIST [78], Ginex [14], SmartSAGE [79], ReGNN [80] and BeaconGNN [81]. While these solutions enable training on larger datasets or offer greater acceleration, their compatibility remains challenging.

Tackle memory-dependency challenges with heterogeneous memory. A few approaches are attempting to address the memory-dependency challenges via DRAM-PM-based heterogeneous memory (HM). PETPS [82] proposes the first HM-enabled parameter server that achieves both fast recovery and low storage costs for huge embedding models. Sentinel [25] automatically optimizes tensor management on HM systems to address challenges on tensor migration and allocation for DNN training. HNGraph [16] is an HM processing framework focused on traditional graph applications, aiming to reduce random accesses to both local and remote PM nodes. XPGraph [83] is an HM-based graph storage system designed to efficiently manage large-scale evolving graphs using XPLine-Friendly techniques. Given that target scenarios and graph applications are different from the above situations, and few approaches focus on HM-based graph embedding, this motivates us to develop a dedicated HM-based framework for efficient and scalable graph embedding.

***SpMM* acceleration in general-purpose scenarios.** Numerous domain-specific systems and accelerators have been proposed for *SpMM* [28]–[30], [60], [69], [84]–[86]. SPADE [28] is a hardware accelerator for *SpMM* and *SDDMM* that integrates processing elements with CPU cores to avoid data transfers, using a tile-based ISA for flexibility. DTC-SpMM [87] presents a framework incorporating efficient compression formats, reordering techniques, and pipeline optimizations to accelerate general *SpMM* on GPUs. The scalability of *SpMM* was investigated in Intel's PIUMA architecture, addressing challenges of large memory footprints, sparse computational patterns, and irregular memory accesses for graph analytics [88]. FusedMM [60] is an in-memory optimization framework that integrates multiple *SpMM* operations into a single, unified kernel, specifically designed to accelerate graph embedding tasks. SEM-SpMM [69] keeps sparse matrix on SSDs, dense matrices in memory, and incorporates numerous in-memory optimizations for *SpMM* at scale. OMeGa focuses on optimizing *SpMM* specifically for high-performance and large-scale graph embedding using heterogeneous memory processing, which is orthogonal to these techniques.

VI. Conclusions and Discussion

We proposed OMeGa, a novel heterogeneous memory-oriented graph embedding framework. OMeGa leverages an *entropy-aware thread allocation* (EaTA), achieving workload balancing and tail latency reduction among threads. It further proposes a *workload feature-aware prefetcher* (WoFP) to avoid unnecessary data movement during heterogeneous processing. OMeGa invents a *NUMA-aware data placement* (NaDP), minimizing NUMA's adverse impact on PM. Additionally, it incorporates a *compressed sparse degree-block* (CSDB) graph format and *asynchronous adaptive streaming loading* (ASL) for heterogeneous memory processing. Experimental results show OMeGa outperforms state-of-the-art systems in efficiency and scalability. Notably, OMeGa is highly flexible and adaptable to diverse storage hierarchies. Its EaTA and WoFP optimize SpMM parallel efficiency for graph embedding, applicable to any storage system. While NUMA architectures face remote-local access gaps, NaDP remains effective across alternative hierarchies. The rise of CXL enables the integration of PM into scalable memory architectures, further advancing high-performance graph embedding.

Acknowledgment

This work was supported by the National Key R&D Program of China (No. 2023YFB4502801), National Natural Science Foundation of China (Grant Nos. U22A2027 and 62402187), China Postdoctoral Science Foundation (Grant Nos. GZB20240243 and 2024M751009), and Postdoctoral Project of Hubei Province (Grant No. 2024HBBHCXA024). Siqiang Luo is supported by Singapore MOE AcRF Tier-2 (Grant No. T2EP20122-0003).

REFERENCES

[1] X. Wei, L. Xu, B. Cao, and P. S. Yu, "Cross View Link Prediction by Learning Noise-resilient Representation Consensus," in *Proceedings of the 26th International Conference on World Wide Web (WWW)*, 2017, pp. 1611–1619.

[2] S. Bhagat, G. Cormode, and S. Muthukrishnan, "Node Classification in Social Networks," in *Social Network Data Analytics*, C. C. Aggarwal, Ed. Springer, 2011, pp. 115–148.

[3] F. Nie, W. Zhu, and X. Li, "Unsupervised Large Graph Embedding," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 31, no. 1, 2017.

[4] C. Shi, B. Hu, W. X. Zhao, and S. Y. Philip, "Heterogeneous information network embedding for recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 2, pp. 357–370, 2018.

[5] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "WTF: The Who to Follow Service at Twitter," in *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, 2013, pp. 505–514.

[6] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018, pp. 839–848.

[7] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "LINE: Large-scale Information Network Embedding," in *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015, pp. 1067–1077.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014, pp. 701–710.

[9] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 855–864.

[10] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, "ProNE: Fast and Scalable Network Representation Learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 19, 2019, pp. 4278–4284. [Online]. Available: https://github.com/THUDM/ProNE

[11] P. Fang, A. Khan, S. Luo, F. Wang, D. Feng, Z. Li, W. Yin, and Y. Cao, "Distributed graph embedding with information-oriented random walks," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1643–1656, 2023. [Online]. Available: https://github.com/RocmFang/DistGER

[12] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2020, pp. 36–44. [Online]. Available: https://github.com/dmlc/dgl

[13] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–14.

[14] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, p. 2626–2639, 2022. [Online]. Available: https://github.com/SNU-ARC/Ginex

[15] J. H. Kim, Y. J. Moon, and S. H. Noh, "An experimental study on the effect of asymmetric memory latency of new memory on application performance," in *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 493–498.

[16] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "HNGraph: Parallel graph processing in hybrid memory based numa systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 388–397.

[17] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[18] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 875–890.

[19] K. Huang, D. Imai, T. Wang, and D. Xie, "Ssds striking back: The storage jungle and its implications to persistent indexes." in *The Conference on Innovative Data Systems Research (CIDR)*, vol. 22, 2022, pp. 1–8.

[20] K. Wu, J. Ren, I. Peng, and D. Li, "ArchTM:Architecture-Aware, high performance transaction for persistent memory," in *19th USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 141–153.

[21] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014, pp. 153–165.

[22] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1077–1091.

[23] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 169–182.

[24] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to numa-aware persistent memory indexes," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 93–111.

[25] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 598–611.

[26] Y. Liu, Y. Ren, M. Liu, H. Li, H. Guo, X. Miao, X. Hu, and H. Chen, "Optimizing file systems on heterogeneous memory by integrating DRAM cache with virtual memory management," in *22nd USENIX Conference on File and Storage Technologies (FAST)*, 2024, pp. 71–87.

[27] C. Lameter, "An overview of non-uniform memory access," *Communications of the ACM*, vol. 56, no. 9, pp. 59–54, 2013.

[28] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, "SPADE: A flexible and scalable accelerator for spmm and sddmm," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–15.

[29] G. Gerogiannis, S. Aananthakrishnan, J. Torrellas, and I. Hur, "HotTiles: Accelerating spmm with heterogeneous accelerator architectures," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 1012–1028.

[30] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2020, pp. 1–12.

[31] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 26, 2013.

[33] J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J. Tang, "NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization," in *Proceedings of the 28th International Conference on World Wide Web (WWW)*, 2019, pp. 1509–1520.

[34] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick, "Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank," *Prococeding of VLDB Endowment*, vol. 13, no. 5, pp. 670–683, 2020.

[35] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec," in *Proceedings of the 11th ACM International Conference on Web Search and Data Mining (WSDM)*, 2018, pp. 459–467.

[36] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang, "Community Preserving Network Embedding," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 31, no. 1, 2017.

[37] P. Fang, F. Wang, Z. Shi, H. Jiang, D. Feng, and L. Yang, "HuGE: An entropy-driven approach to efficient and scalable graph embeddings," in *IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2045–2050.

[38] P. Fang, F. Wang, Z. Shi, H. Jiang, D. Feng, X. Xu, and W. Yin, "How to realize efficient and scalable graph embeddings via an entropy-driven mechanism," *IEEE Transactions on Big Data*, vol. 9, no. 1, pp. 358–371, 2022.

[39] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[40] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[41] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xie, and M. Guo, "GraphGAN: Graph Representation Learning With Generative Adversarial Nets," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, vol. 32, no. 1, 2018.

[42] S. Yang, M. Zhang, W. Dong, and D. Li, "Betty: Enabling large-scale gnn training with batch-level graph partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 103–117.

[43] D. Zhu, Z. Zhang, P. Cui, and W. Zhu, "Robust graph convolutional networks against adversarial attacks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2019, pp. 1399–1407.

[44] Y. Yin and Z. Wei, "Scalable graph embeddings via sparse transpose proximities," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2019, p. 1429–1437.

[45] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

[46] J. R. Lee, S. O. Gharan, and L. Trevisan, "Multiway spectral partitioning and higher-order cheeger inequalities," *Journal of the ACM*, vol. 61, no. 6, pp. 1–30, 2014.

[47] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware (MOD)*, 2020, pp. 1–8.

[48] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2014, pp. 769–780.

[49] G. Huang, G. Dai, Y. Wang, Y. Ding, and Y. Xie, "Efficient sparse matrix kernels based on adaptive workload-balancing and parallel-reduction," *arXiv preprint arXiv:2106.16064*, 2021.

[50] J. Korner and K. Marton, "Random access communication and graph entropy," *IEEE Transactions on Information Theory*, vol. 34, no. 2, pp. 312–314, 1988.

[51] R.-H. Li, J. X. Yu, and J. Liu, "Link prediction: the power of maximal entropy random walk," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM)*, 2011, pp. 1147–1156.

[52] "Non-uniform memory access control," NUMACTL, https://github.com/numactl/numactl.

[53] "Flexible i/o teste," FIO, https://github.com/axboe/fio.

[54] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015, pp. 183–193.

[55] "Intel memory latency checker," MLC, https://www.intel.com/content/www/us/en/developer/articles/tool/intelrmemory-latency-checker.html.

[56] "Intel vtune profiler," VTune, https://software.intel.com/en-us/vtune.

[57] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1304–1318, 2020.

[58] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, "Mariusgnn: Resource-efficient out-of-core training of graph neural networks," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023, p. 144–161. [Online]. Available: https://github.com/marius-team/marius

[59] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication for billion-node graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1470–1483, 2016. [Online]. Available: https://github.com/flashxio/FlashX

[60] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266. [Online]. Available: https://github.com/HipGraph/FusedMM

[61] "Our code and datasets," https://github.com/RocmFang/OMeGa, 2025.

[62] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference and International Workshop Present Day Trends of Innovations*, vol. 1, no. 6, 2012.

[63] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2006, pp. 44–54.

[64] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012, pp. 1–8.

[65] R. Zafarani and H. Liu, "Social computing data repository at asu," in *http://socialcomputing.asu.edu*, 2009.

[66] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010, pp. 591–600.

[67] "Com-friendster dataset," FR, https://snap.stanford.edu/data/com-Friendster.html.

[68] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the SIAM International Conference on Data Mining (ICDM)*, 2004, pp. 442–446.

[69] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication for billion-node graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1470–1483, 2017.

[70] J. Sun, L. Su, Z. Shi, W. Shen, Z. Wang, L. Wang, J. Zhang, Y. Li, W. Yu, J. Zhou *et al.*, "Legion: Automatically pushing the envelope of multi-gpu system for billion-scale gnn training," in *USENIX Annual Technical Conference (USENIX ATC)*, 2023, pp. 165–179.

[71] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu, "Seastar: vertex-centric programming for graph neural networks," in *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, 2021, pp. 359–375.

[72] Y. Wang and C. Mendis, "TGLite: A lightweight programming framework for continuous-time temporal graph neural networks," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 1183–1199.

[73] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "AliGraph: A comprehensive graph neural network platform," *Proceedings of VLDB Endowment*, p. 2094–2105, 2019.

[74] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang, "ByteGNN: Efficient Graph Neural Network Training at Large Scale," *Proceedings of VLDB Endowment*, vol. 15, no. 6, pp. 1228–1242, 2022.

[75] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "DistGNN: Scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–14.

[76] S. Dong, X. Miao, P. Liu, X. Wang, B. Cui, and J. Li, "HET-KG: Communication-efficient knowledge graph embedding training via hotness-aware cache," in *IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1754–1766.

[77] P. Fang, Z. Li, A. Khan, S. Luo, F. Wang, Z. Shi, and D. Feng, "Information-oriented random walks and pipeline optimization for distributed graph embedding," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 1, pp. 408–422, 2025.

[78] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "Glist: Towards in-storage graph learning," in *USENIX Annual Technical Conference (USENIX ATC)*, 2021, pp. 225–238.

[79] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 932–945.

[80] C. Liu, H. Liu, H. Jin, X. Liao, Y. Zhang, Z. Duan, J. Xu, and H. Li, "ReGNN: a reram-based heterogeneous architecture for general graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 469–474.

[81] Y. Wang, X. Pan, Y. An, J. Zhang, and G. Reinman, "BeaconGNN: Large-scale gnn acceleration with out-of-order streaming in-storage computing," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 330–344.

[82] M. Xie, Y. Lu, Q. Wang, Y. Feng, J. Liu, K. Ren, and J. Shu, "PetPS: Supporting huge embedding models with persistent memory," *Proceedings of the VLDB Endowment*, vol. 16, no. 5, pp. 1013–1022, 2023.

[83] R. Wang, S. He, W. Zong, Y. Li, and Y. Xu, "XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs," in *55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1308–1325.

[84] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 600–614.

[85] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022, pp. 65–77.

[86] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 689–702.

[87] R. Fan, W. Wang, and X. Chu, "DTC-SpMM: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024, pp. 253–267.

[88] M. J. Adiletta, J. J. Tithi, E.-I. Farsarakis, G. Gerogiannis, R. Adolf, R. Benke, S. Kashyap, S. Hsia, K. Lakhotia, F. Petrini *et al.*, "Characterizing the scalability of graph convolutional networks on intel® piuma," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 168–177.