

# Efficient Graph Embedding at Scale: Optimizing CPU-GPU-SSD Integration

Zhonggen Li  
Zhejiang University  
Hangzhou, China  
zgli@zju.edu.cn

Xiangyu Ke  
Zhejiang University  
Hangzhou, China  
xiangyu.ke@zju.edu.cn

Yifan Zhu  
Zhejiang University  
Hangzhou, China  
xtf\_z@zju.edu.cn

Yunjun Gao  
Zhejiang University  
Hangzhou, China  
gaoyj@zju.edu.cn

Feifei Li  
Alibaba Group  
Hangzhou, China  
lifeifei@alibaba-inc.com

## Abstract

Graph embeddings provide continuous vector representations of nodes in a graph, which are widely applicable in community detection, recommendations, and various scientific fields. However, existing graph embedding systems either face scalability challenges due to the high cost of RAM and multiple GPUs, or rely on disk storage at the expense of I/O efficiency.

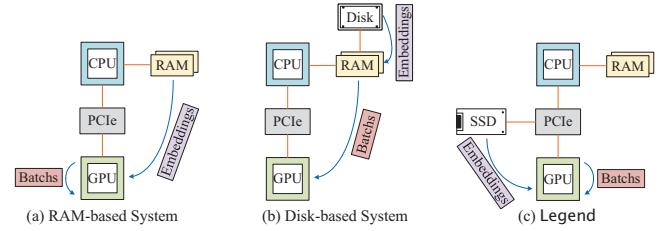
In this paper, we propose Legend, a lightweight heterogeneous system for graph embedding that systematically redefines data management across CPU, GPU, and NVMe SSD resources. Legend is built on a foundation of efficient data placement and retrieval strategies tailored to the unique strengths of each hardware. Key innovations include a prefetch-friendly embedding loading strategy, enabling GPUs to directly prefetch data from SSDs with minimal I/O overhead, and a high-throughput GPU-SSD direct access driver optimized for graph embedding tasks. Furthermore, we propose a customized parallel execution strategy to maximize GPU utilization, ensuring efficient handling of billion-scale datasets. Extensive experiments demonstrate that Legend achieves up to 4.8 $\times$  speedup compared to state-of-the-art systems. Moreover, Legend exhibits comparable performance on a single GPU to that of the state-of-the-art system using 4 GPUs on the billion-scale dataset.

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/ZJU-DAILY/Legend>.

## 1 Introduction

Graphs are fundamental data structures used to represent relationships across diverse domains, including social network users [9, 18, 54], proteins [1, 24], genes [12, 43], among many others. The recent advancements in graph machine learning have unlocked their potential to address a wide range of analytical tasks, such as link prediction [56, 63, 66], node classification [40, 49, 58], and beyond [7, 22, 41]. These capabilities rely on graph embeddings—continuous



**Figure 1: Comparison of system architectures.**

vector representations of nodes that encode the structural and semantic properties of graphs. These embeddings serve as the foundation for downstream applications in recommendation systems [27, 50, 57], finance [39, 47], drug discovery [15, 51, 61], and many others [4, 8, 17].

To meet the computational demands of graph embedding learning for large-scale graphs, a variety of systems have been developed, such as DGL-KE by Amazon [60], PyTorch Big Graph (PBG) by Meta [20], Marius [30], and GE<sup>2</sup> [59]. These systems harness the computational power of GPUs to accelerate training and enable scalable graph processing. However, as real-world graphs often contain millions or more nodes, these systems require tens or thousands of gigabytes of memory to maintain the embedding vectors and optimizer states, making it impractical to keep all these data in GPU memory [30, 59]. Consequently, two primary solutions have emerged. **(i) RAM-based systems** (e.g., DGL-KE and GE<sup>2</sup>) store embeddings and optimizer states in CPU memory (RAM), transferring data to GPU memory via PCIe as needed. **(ii) Disk-based systems** (such as PBG and Marius) store embeddings on disk, loading partitions into RAM and subsequently transferring them to GPU memory. Figures 1(a) and (b) illustrate the architecture of these systems. However, as shown in Table 1, the limitations of RAM capacity and its associated costs hinder the scalability of large graphs, increasing the storage expenses for RAM-based systems by 16 $\times$ . In contrast, disk-based methods suffer from limited data transfer bandwidth, approximately 8 $\times$  slower, and suboptimal GPU utilization, averaging only 58%.

In addition to the high cost and inefficiency associated with embedding storage, existing graph embedding systems face two critical computational limitations: **(i) CPU Dependence**. Systems like Marius rely on the CPU for batch construction, negative sampling, and embedding updates. This reliance fails to utilize the powerful

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.  
doi:XX.XX/XXX.XX

**Table 1: Statistics of cost and efficiency on dataset Twitter (Edges: 1.3B, Nodes: 41.6M).**

Systems	Storage cost	Computing cost	Communication bandwidth	Computing throughput	Batch time	Total time
GE <sup>2</sup>	2.02 \$/GB	33200\$ (4 A100)	10.05 GB/s (CPU-GPU)	$6.75 \times 10^6$ edges/s	18.5 ms	32 min
Marius	0.08 \$/GB	8300\$ (1 A100)	0.38 GB/s (Disk-CPU)	$1.49 \times 10^6$ edges/s	315.6 ms	146 min
<b>Legend</b>	<b>0.13 \$/GB</b>	<b>8300\$ (1 A100)</b>	<b>3.06 GB/s (SSD-GPU)</b>	<b><math>7.18 \times 10^6</math> edges/s</b>	<b>12.0 ms</b>	<b>30 min</b>

computational capabilities of GPU and increases communication overhead between CPU and GPU, resulting in processing delays of up to 26× per batch (Table 1). **(ii) Suboptimal GPU Utilization.** Most systems prioritize architectural design much over training optimization. Our experiments reveal that gradient computation on the GPU accounts for more than 80% of the batch processing time, representing a significant bottleneck in graph embedding workflow.

Recently, NVMe SSDs have gained traction due to their favorable balance of cost and performance [11, 14, 34]. Building on this, we propose a novel architecture (Figure 1(c)) that *replaces traditional storage mediums (RAM or disk) with NVMe SSD* and *employ GPU-SSD direct access* to achieve cost-effective and efficient storage. Furthermore, we accelerate the training by *strategically allocating tasks and optimizing GPU computations*. While prior research has explored GPU-SSD direct access for various applications [2, 16, 33], adapting this architecture for graph embedding systems introduces three unique challenges:

- **Task Mapping.** Existing workflows for graph embedding tasks are not well-suited for the CPU-GPU-SSD architecture, resulting in suboptimal performance. For instance, systems like Marius, which treat SSDs as traditional disk storage, place excessive computational burdens on the CPU. The CPU must handle data transfers from the SSD to RAM and subsequently to the GPU, significantly impacting efficiency. Alternatively, workflows like GE<sup>2</sup>, which treat SSDs as RAM, also degrade performance because the bandwidth between SSD and GPU is more than three times lower than that between RAM and GPU.
- **I/O Bottlenecks.** Directly loading embeddings from SSD to GPU, as depicted in Figure 1(c), eliminates the overhead of CPU-mediated data transfers. However, our experiments indicate that I/O operation still accounts for more than 25% of the training pipeline time. The inherently lower data transfer bandwidth of SSDs compared to RAM remains a significant impediment to high training efficiency.
- **Computational Intensity.** After embeddings are loaded into the GPU, tasks such as batch construction and training rely on intensive operations like exponentiation and multiplication. These operations fail to fully exploit GPU resources, making gradient computation a dominant computational bottleneck.

In this paper, we propose Legend, an efficient and scalable graph embedding system that systematically integrates CPU, GPU, and SSD, leveraging their unique strengths. We propose the following techniques to address the challenges above.

**Storage Arrangement and Task Allocation.** To efficiently map the graph embedding tasks, we carefully design the workflow to enable seamless integration of the SSD (§ 3). The embeddings and optimizer states are stored in SSD, while the graph data is kept in

RAM. This distribution allows for efficient memory utilization by offloading large, less frequently accessed data to the SSD, reducing the storage burden on host memory. At the same time, storing graph data in RAM ensures faster access to frequently used information, facilitating high-speed computation. Additionally, we allocate control tasks to the CPU and offload computational tasks to the GPU, maximizing the unique strengths of each hardware component.

**I/O Optimizations.** To optimize I/O efficiency, we propose two techniques. **(i) Embedding prefetching.** Prefetching is a widely used approach to reduce I/O overhead [53, 65]. For massive graph embedding learning, embeddings are usually partitioned and loaded from storage to GPU in an I/O-optimized order [20, 60]. Computing a prefetch-friendly order while minimizing I/O times is NP-hard as proved in Theorem 2. To address this issue, we propose an efficient order-generating algorithm (§ 4). It employs a column separation covering strategy, generating a partition swapping order that achieves comparable I/O times to the state-of-the-art algorithm while supporting prefetching [30]. **(ii) Customized GPU-SSD direct access mechanism.** Simply employing existing GPU-SSD direct access driver leads to redundant overhead of lock and doorbell ringing, which is not efficient in graph embedding [26, 33, 37]. To tackle this problem, we reconsider the NVMe queue management in the context of graph embedding (§ 5). In Legend, each thread within the block can perform enqueue and dequeue operations simultaneously by precomputing the positions of queue entries, ensuring high parallelism. Besides, we design a novel doorbell ringing and completion queue polling strategy to reduce GPU resource occupation and GPU-SSD communication overhead.

**Computation Optimizations.** To efficiently calculate the gradients, we design a parallel strategy tailored to graph embedding learning workloads that fully leverages Tensor cores, registers, and shared memory in GPU. This strategy reduces heavy memory access while ensuring high parallelism. Additionally, we identify redundant calculations and reuse intermediate results to further reduce computational costs (§ 6). These optimizations address the neglected issue of gradient computation in existing graph embedding systems, achieving higher GPU utilization.

Comprehensive experiments demonstrate the efficiency and scalability of Legend. It achieves a speedup of up to 4.8× compared to the state-of-the-art graph embedding systems. Legend is also lightweight and exhibits comparable performance on a single GPU to GE<sup>2</sup> using 4 GPUs on the Twitter dataset (§ 7).

In summary, our key contributions are as follows:

- We design a workflow for graph embedding in the CPU-GPU-NVMe SSD heterogeneous systems, considering the respective characteristics of each hardware (§ 3).
- We prove the NP-hardness of identifying a prefetch-friendly iteration order and propose a heuristic algorithm (§ 4). And

- we devise a customized GPU-SSD direct access mechanism (§ 5) to achieve efficient I/O during embedding training.
- We optimize the gradient computation on the GPU by devising a specific parallel strategy and exploiting the computing resources to enhance GPU utilization (§ 6).
  - We conduct comprehensive evaluations demonstrating that Legend outperforms existing systems, achieving up to 4.8× speedup for massive graph embedding (§ 7).

The rest of this paper is organized as follows. Section 2 briefly introduces the background of graph embedding, GPU architectures, and data access mechanism in NVMe SSD. Section 3 presents the workflow design in Legend. Section 4 describes the iteration order. Section 5 and Section 6 illustrate the optimization on SSD and GPU, respectively. Section 7 exhibits the experimental results. Section 8 reviews the related studies. We conclude the paper in Section 9.

## 2 Preliminaries

In this section, we first provide an overview of graph embedding learning. Then we offer a concise description of GPU architecture, following the data access mechanism of NVMe SSD.

### 2.1 Graph Embedding Learning

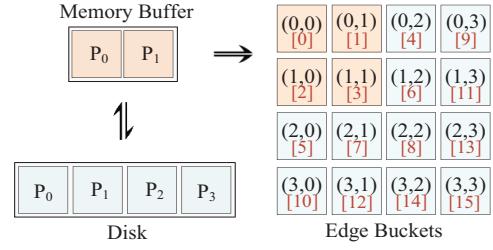
Following PBG [20], Marius [30] and GE<sup>2</sup> [59], we focus on the multi-relation graphs denoted by  $G = (V, R, E)$ , where  $V$  represents the set of nodes (entities),  $R$  is the set of edge (relation) types and  $E$  is the set of edges. Each edge  $e \in E$  is a triplet denoted as  $e = (s, r, d)$ , where  $s$  is the source node,  $r$  is the relation type, and  $d$  is the destination node. The triplet  $(s, r, d)$  signifies that entity  $s$  has a relationship  $r$  with entity  $d$ , indicating the presence of an edge between  $s$  and  $d$ . Although Legend primarily targets multi-relation graphs, it’s also capable of handling graphs without relation types.

An embedding is a vector  $\theta$  of fixed dimension. During graph embedding learning, the elements in the embedding vectors of each node and relation type are iteratively updated based on their previous values. Specifically, graph embedding learning uses a score function  $f(\theta_s, \theta_r, \theta_d)$ , where  $\theta_s, \theta_r$  and  $\theta_d$  represent the embedding vectors of  $s, r, d$  in the triplet  $e = (s, r, d)$ . The goal of graph embedding learning is to maximize  $f(\theta_s, \theta_r, \theta_d)$  for  $(s, r, d) \in E$  and minimize  $f(\theta_{s'}, \theta_{r'}, \theta_{d'})$  for  $(s', r', d') \notin E$ , where  $e' = (s', r', d')$  is referred to as a negative edge. This objective is achieved using the contrastive loss function, as shown in Equation 1.

$$\mathcal{L} = - \sum_{(s, r, d) \in E} (f(\theta_s, \theta_r, \theta_d) - \log(\sum_{(s', r', d') \notin E} e^{f(\theta_{s'}, \theta_{r'}, \theta_{d'})})) \quad (1)$$

Embedding models update the embeddings through mini-batch stochastic gradient descent (SGD). Existing systems employ Adagrad as the optimizer for parameter updates. For each positive edge (edges from  $E$ ), several negative edges are sampled using negative sampling algorithms. A mini-batch is composed of embeddings corresponding to both positive edges and negative edges. During each epoch, all edges in  $E$  are calculated once as positive edges.

As the number of nodes in a graph can easily reach hundreds of millions, the limited GPU memory cannot accommodate such large-scale embedding data. To enable scalable training, we adopt a partition-based scheme similar to PBG. As illustrated in Figure 2, PBG divides the node embeddings into several equal-sized partitions ( $\{P_0, P_1, P_2, P_3\}$ ) based on the node IDs, and stores them



**Figure 2: Partition-based training scheme.**  $P_i$  denotes the node partition and  $[j]$  denotes the calculating order.

on disk. In practical implementations, optimizer states are stored alongside the node embeddings, although they are omitted in Figure 2 for simplicity. Correspondingly, the edges are grouped into several buckets, where the bucket ID  $(i, j)$  indicates that the source nodes of these edges are located in node partition  $P_i$ , and the destination nodes are located in node partition  $P_j$ .

During training, the edge buckets are processed in a specific order, such as the order denoted by "[k]" in Figure 2. To retrieve the embeddings related to these edges, the corresponding node partitions are required to be loaded into the memory buffer. For example, the memory buffer in Figure 2 contains  $P_0$  and  $P_1$ , supporting the training of edge buckets  $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ . As the edge buckets are processed in order, the node partitions in the memory buffer are continuously updated. The node partitions in the memory buffer at any given time are referred to as the *buffer state*. For instance, the current buffer state in Figure 2 is  $\{P_0, P_1\}$ . During each training epoch, every edge bucket is iterated once.

It is important to note that the order in which node partitions are loaded and edge buckets are processed significantly affects the I/O times between the disk and memory buffer. For example, if the edge buckets are iterated in the order  $\{(0, 0), (1, 3), (1, 0)\}$ , the I/O time is 4, as  $P_0$  is loaded twice and other partitions are loaded once. In contrast, iterating in the order  $\{(0, 0), (1, 0), (1, 3)\}$  reduces the I/O time to 3, since  $P_0$  is loaded only once.

### 2.2 GPU Architecture

Modern GPUs feature two types of computing cores: CUDA cores and Tensor cores. CUDA cores serve as the primary computational units for general-purpose tasks, while Tensor cores are specialized for efficient matrix multiplication, enabling the multiplication of fixed-size matrices within a single clock cycle [62]. The GPU’s memory hierarchy consists of global memory, shared memory, and registers. Global memory provides the largest capacity but has relatively lower I/O bandwidth. Shared memory, accessible by all threads within each thread block, offers higher bandwidth. Registers provide the fastest data access among these memory structures, which are private to individual threads once declared [5, 32].

### 2.3 Data Access Mechanism of NVMe SSD

The NVMe SSD facilitates data transmission by leveraging queue pairs, which consist of submission queues (SQs) and completion queues (CQs). Multiple queue pairs in an NVMe SSD enable parallel responses to requests, ensuring high throughput [26, 37]. When a CPU or GPU requests data, it first constructs NVMe commands following the NVMe protocol. These commands specify the request

type (read/write), page size, request address, data placement address, and other parameters. Subsequently, it places the command at the end of an SQ. Afterward, it signals to the NVMe controller by writing the updated tail pointer into the doorbell register of the NVMe SSD via PCIe, indicating that new commands have been added to the SQ. The NVMe controller processes the commands, transfers the requested data to the host memory, and places completion entries into the CQ. Finally, the CPU or GPU retrieves the completion entries from the CQ and informs the NVMe controller by writing the new head pointer to the doorbell register, signifying that the new entries in the CQ have been processed [37].

### 3 Workflow in Legend

In this section, we introduce the workflow of Legend, focusing on storage arrangement, task assignment across different hardware components, and the overall data flow among each hardware.

Following the partition schema used in PBG [20], Marius [30] and GE<sup>2</sup> [59], Legend divides the graph’s nodes into  $n$  equal-sized partitions based on their IDs ( $n = 4$  in Figure 3). As a result, the node embeddings are split into  $n$  corresponding partitions and the edges are distributed into buckets. For example, the edge bucket (1, 2) in Figure 3 indicates that the source nodes of the edges in this edge bucket belong to node partition 1, while the destination nodes are from node partition 2. Next, we will introduce how Legend maps storage and tasks to the architecture of CPU-GPU-SSD.

**Storage Arrangement.** In Legend, node embeddings and optimizer states are stored in the NVMe SSD, which occupies the majority of memory space during the graph embedding learning process. To maximize the bandwidth and make full use of the high parallelism of NVMe SSD, the embeddings and optimizer states of each partition are stored in consecutive memory addresses. This allows embedding and optimizer states of a partition to be loaded simultaneously with a single kernel in the GPU. The edges, on the other hand, are stored in RAM, which requires significantly less space compared to embeddings and optimizer states. Storing the edges in RAM rather than NVMe SSD offers two key advantages. First, since the CPU controls the graph embedding learning process, it can effectively track the GPU’s progress—specifically, which edge bucket is currently being processed. As a result, the CPU can transfer new edge buckets to the GPU on time and instruct the GPU to fetch the required embedding data from NVMe SSD. Second, although the theoretical bandwidth of RAM and SSD to GPU is the same, the actual bandwidth between RAM and GPU is more than 3 times higher than that between one NVMe SSD and GPU in our experiments (Table 1) due to the hardware restriction. Thus, storing edge buckets in RAM allows for efficient and synchronous transfers from the CPU to the GPU, reducing the GPU’s idle time. For multi-relation graphs, the number of relation types is typically small, necessitating frequent synchronous updates [30]. Consequently, we store relation embeddings (denoted as Rel. Embs.) and optimizer states (denoted as Rel. Stas.) in the global memory of the GPU, following the design of existing systems [30, 59, 60].

**Tasks Mapping.** Considering the powerful ability of the CPU to handle complex logic and control tasks, the CPU is responsible for

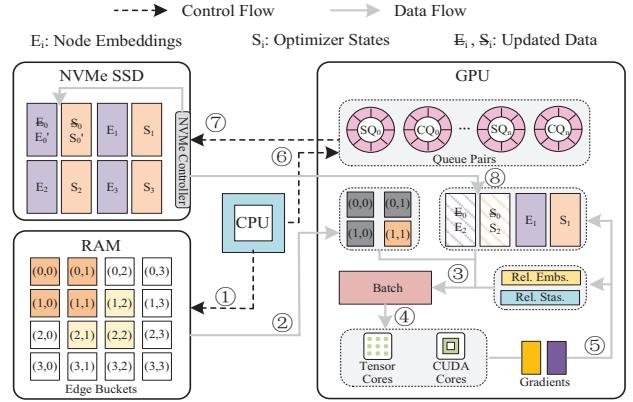
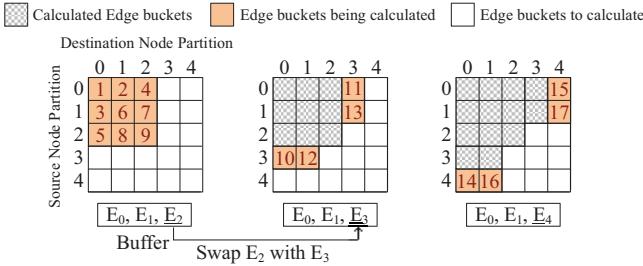


Figure 3: Workflow of Legend.

moving data and sending commands to the GPU and SSD in Legend, coordinating and controlling the processes of tasks on various hardware components. Meanwhile, the GPU takes on all computing tasks. Based on this strategy, the CPU first transfers edge buckets from RAM to GPU and the GPU subsequently constructs batches as well as computes gradients. Once the CPU detects that the edges on the GPU are going to be used up, it instructs the GPU to fetch the next embedding partition from the SSD. Afterwards, the CPU transfers new edge buckets to the GPU and a new round of process begins in the same way.

Specifically, assume that the nodes are divided into four partitions and that the GPU global memory can accommodate two partitions at a time, i.e., the buffer capacity is 2. Initially, the embeddings and optimizer states of partition 0 and partition 1 reside in the GPU global memory as shown in Figure 3. With partition 0 and partition 1, the GPU conducts the computation of 4 edge buckets, namely  $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ . To do this, as depicted in Figure 3, the CPU ① fetches edge buckets  $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$  from RAM and ② transfers them to the GPU global memory. The GPU then ③ fetches a fixed number of edges (i.e. positive edges) from the edge buckets, samples negative edges for each positive edge, and retrieves the corresponding embeddings and optimizer states from  $\{\{E_0, E_1\}, \{S_0, S_1\}\}$  to construct a batch. Next, the gradients of this batch ④ are calculated using Tensor cores and CUDA cores, which will be detailed in Section 6. The embeddings and the optimizer states in the global memory are ⑤ updated by the GPU with the computed gradients. The advantages of sampling negative edges and constructing batches on the GPU are threefold. First, the corresponding embeddings of the trained edge buckets are stored in the GPU rather than in RAM, minimizing the need for frequent communication between the CPU and GPU. Second, embeddings are updated synchronously, avoiding the staleness issues encountered in some graph embedding systems, such as Marius. Third, both negative edge sampling and embedding retrieval are parallelizable tasks, making them well-suited for execution on the GPU.

When Legend finishes the calculation of all four edge buckets in Figure 3, it has to exchange an embedding partition in the GPU buffer ( $E_0$  and  $S_0$ ) with another partition in SSD ( $E_2$  and  $S_2$ ). However, the GPU has no computational tasks during data exchange, resulting in low utilization. Consequently, necessary data is prefetched



**Figure 4: Partition loading order in Marius. The numbers inside edge buckets denote their calculated order.**

by Legend at an appropriate time before being used, which will be introduced in Section 4. The CPU ⑥ launches a data access kernel at the appropriate time to have the GPU offload  $E_0$  and  $S_0$  to the NVMe SSD, and to load  $E_2$  and  $S_2$  into global memory, which is the key operation of prefetching. The GPU data access kernel employs several thread blocks each with hundreds of threads to simultaneously construct NVMe commands and enqueue them into the submission queues. Subsequently, each submission queue ⑦ has a dedicated thread to ring the doorbells located in the controller of NVMe SSD, informing the NVMe SSD that there are new data access requests to process. The NVMe controller ⑧ retrieves the data and transfers the data to the required addresses in the GPU global memory using DMA (details will be introduced in Section 5). Meanwhile, the GPU calculates gradients for the remaining edge buckets during data exchange. The data transfer overhead can be hidden in the computation of the remaining edge buckets, achieving overall performance improvement. Furthermore, GPU waiting time is eliminated by prefetching the data that needs to be calculated, thereby increasing GPU utilization.

## 4 Edge Bucket Ordering

In this section, we illustrate our proposed prefetch-friendly partition loading order and the edge bucket iteration order.

As discussed in Section 3, when the GPU completes the training of all edge buckets associated with the current partitions in its global memory, it has to wait for the transfer of the next partitions, leading to reduced GPU utilization. If embeddings and optimizer states of the next partition are prefetched into the GPU’s global memory before being used, the GPU can perform calculations for the subsequent batch without incurring waiting time. Although a prefetching technique is implemented in *Marius* [30], the loading order of partitions is not conducive to effective prefetching.

We illustrate the issue with *Marius*’s loading order through an example. As depicted in Figure 4, the memory buffer is initialized with partitions  $\{E_0, E_1, E_2\}$ . *Marius* computes the edge buckets in the order of  $\{(0, 0), (0, 1), (1, 0), (0, 2), (2, 0), (1, 1), (1, 2), (2, 1), (2, 2)\}$ . Next, partition  $E_2$  is evicted and  $E_3$  will be swapped in. However,  $E_3$  cannot be prefetched at this time because there is no remaining edge bucket in the GPU memory to calculate while fetching  $E_3$ .

To achieve prefetching, the edge bucket iteration order of the initial buffer state (the first subfigure in Figure 4) can be adjusted to  $\{(0, 2), (2, 0), (1, 2), (2, 1), (2, 2)\}, \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ . This exchange prioritizes the computation of edge buckets related to the partition  $E_2$  before swapping it out for  $E_3$ . Thus, we can exchange  $E_2$  with  $E_3$  during calculating the remaining edge buckets

$\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ . However, this adjustment is only applicable in the initial buffer state. In the subsequent buffer state, as depicted in the second subfigure in Figure 4, the edge buckets are all related to  $E_3$ , hindering the eviction of  $E_3$  and the prefetching of the next partition. The underlying issue is that the partition swapped in during the previous buffer state is immediately evicted in the next buffer state, leaving insufficient time for prefetching. Although *GE<sup>2</sup>* [59] employs another strategy to reduce the I/O times, it also fails to support effective prefetching.

From the preceding discussion, we observe that prefetching can be achieved if the recently loaded partition is not immediately swapped out of the GPU memory buffer. In our problem setting, we assume the number of partitions is  $n (n > 3)$  and the buffer capacity in the GPU’s global memory is **fixed at 3** for simplification. This configuration enables training on datasets of any size, as the value of  $n$  is arbitrary. To identify a loading order that supports prefetching, we first define the concept of a *Prefetching Supported Order*.

**DEFINITION 1 (PREFETCHING SUPPORTED ORDER).** *A Prefetching Supported Order is a partition loading order such that there is at least one edge bucket not related to the partition scheduled for eviction in each buffer state.*

According to Definition 1, in each buffer state, we can first compute the edge buckets related to the partition that will be evicted and simultaneously load a new partition during the computing of the edge buckets unrelated to this partition. The loading orders in *Marius* and *GE<sup>2</sup>* do not qualify as a *Prefetching Supported Order*, as they have no edge bucket unrelated to the partition that will be evicted in most buffer states. To effectively identify the *Prefetching Supported Order*, we explore its properties in Theorem 1.

**THEOREM 1.** *For the buffer capacity of 3, a partition loading order is classified as a Prefetching Supported Order if it satisfies two properties: (1) The partition that has just been swapped in each buffer state will not be immediately evicted in the subsequent buffer state. (2) Any two partitions may appear concurrently in multiple buffer states, but only in consecutive buffer states.*

**PROOF.** Without loss of generality, we consider the consecutive buffer states  $\{E_i, E_j, E_k\}, \{E_i, E_j, \underline{E_l}\}, \{E_i, E_m, E_l\}$ , where the **blue** color and the underline denote the loaded partition in the current buffer state and the partition to be evicted, respectively.  $E_i$  and  $E_j$  appear concurrently in the first two consecutive buffer states. For the buffer state  $\{E_i, E_j, \underline{E_k}\}$ , the edge buckets related to  $E_i$  and  $E_k$  have been calculated during loading  $E_j$ .  $E_k$  is the partition in the current buffer state that is going to be evicted. The edge buckets  $\{(i, j), (j, i)\}$  that is not related to  $E_k$  must not have been calculated due to the property (2). Otherwise, if the property (2) is not satisfied,  $E_i$  and  $E_j$  can appear concurrently in a previous buffer state. Their corresponding edge buckets  $(i, j)$  and  $(j, i)$  have already been calculated at that buffer state. Therefore we cannot exchange  $E_k$  with  $E_l$  while calculating other edge buckets at buffer state  $\{E_i, E_j, \underline{E_k}\}$ , leading to the failure of prefetching. Similarity, for next buffer state  $\{E_i, E_j, \underline{E_l}\}$ ,  $E_j$  is going to be evicted. Edge buckets  $\{(i, l), (l, i)\}$  are not related to  $E_j$  and have not been calculated. We can evict  $E_j$  and prefetch the next partition while calculating  $\{(i, l), (l, i)\}$ .  $\square$

---

**Algorithm 1:** Node partition loading order

---

**Input:** Node partitions  $n$ , buffer capacity 3  
**Output:** Buffer states in order.

```

1  $EdgeBuckets \leftarrow \{0\}_{n \times n}$ ,  $BufStates \leftarrow \{\}$ ,  $CurCol \leftarrow 0$ ;
2  $BufStates.append(\{0, 1, 2\})$ ,  $Buffer \leftarrow BufStates[-1]$ ;
3 for  $i$  in range(3,  $n$ ) do
4    $Buffer \leftarrow Buffer - \{i - 2\} + \{i\}$ ;
5    $BufStates.append(Buffer)$ ;
6 Set the corresponding edge buckets of  $BufStates$  to 1;
7 while sum( $EdgeBuckets$ ) <  $n^2$  do
8    $ToEvict \leftarrow -1$ ,  $ToLoad \leftarrow -1$ ;
9   if sum( $EdgeBuckets[CurCol]$ ) =  $n$  then
10    |  $Buffer \leftarrow Buffer - \{CurCol\} + \{CurCol + 1\}$ ;
11    |  $BufStates.append(Buffer)$ ;
12    | Set the corresponding edge buckets of  $Buffer$  to 1;
13    | if sum( $EdgeBuckets[CurCol][max(Buffer) + 1:n]$ ) <
14      |  $n - max(Buffer)$  then
15      |   |  $ToEvict \leftarrow max(Buffer)$ ;
16    | else
17    |   |  $ToEvict \leftarrow min(Buffer)$ ;
18    |   |  $CurCol \leftarrow CurCol + 1$ ;
19   | else
20   |   |  $ToEvict \leftarrow id \in BufStates[-1] \cap BufStates[-2]$  and
21     |     |  $id \neq CurCol$ ;
22   |   |  $BeginID \leftarrow (Buffer - \{ToEvict, CurCol\}) + CurCol + 2$ ;
23   |   |  $ToLoad \leftarrow$  the id that covers the most edge buckets from
24     |     |  $BeginID$  to  $BeginID - 1$ ;
25   |   |  $Buffer \leftarrow Buffer - \{ToEvict\} + \{ToLoad\}$ ;
26   |   |  $BufStates.append(Buffer)$ ;
27   | Set the corresponding edge buckets of  $BufStates$  to 1;
28 return  $BufStates$ 

```

---

It is important to note that the impact of the property (2) in Theorem 1 is minimal in practical applications. Without considering property (2), only 4 out of 36 I/O times fail to support prefetching for 12 partitions. Therefore, we exclude property (2) from algorithm design. Our goal is to find an order that satisfies property (1) while minimizing the I/O times. We adopt the same swapping strategy as Marius [30], which allows only one partition to be swapped in each buffer state. Generating an order that satisfies property (1) is straightforward. However, identifying an order that meets property (1) while minimizing I/O times is NP-hard as proved in Theorem 2.

**THEOREM 2.** *With  $n$  partitions and a buffer capacity of 3, the problem of identifying an order that meets property (1) while minimizing I/O times is NP-hard.*

**PROOF.** We demonstrate that the problem is NP-hard via a reduction from the covering design problem, a well-known NP-hard problem. Specifically, an instance of the covering design problem with parameters  $(n, 3, 2)$ , which seeks the minimum number of 3-element subsets (blocks) covering all  $C_n^2$  pairs, is mapped to our problem as follows. The buffer is first initialized with a buffer state, corresponding to an initial block in the covering design problem. A node partition in the buffer is subsequently swapped with a partition out of the buffer. Each exchange operation above generates a new block in the covering design problem. The requirement that all pairs of node partitions must coexist in the buffer is equivalent to ensuring all  $C_n^2$  pairs are covered by the sequence of blocks. A

covering design problem solution with  $m$  blocks implies a valid exchange sequence of  $m - 1$  steps, as each block requires one exchange. Conversely, an exchange sequence of length  $k$  produces  $k+1$  blocks covering all pairs. To address the constraint that is introduced in the property (1) of Theorem 1, intermediate blocks may be inserted (e.g., exchanging two elements sequentially), which only polynomially inflates the sequence length and preserves the reduction's validity. Since the covering design problem is NP-hard, our problem is also NP-hard.  $\square$

The NP-hardness of this problem motivates us to devise an efficient algorithm. We propose a column separation covering strategy to generate a satisfactory order within one second. Figure 5 depicts an example of our proposed node partition loading order and edge bucket iteration order with 6 node partitions.

The key idea of the loading order is to *sequentially cover each column of edge buckets, greedily maximizing coverage in each column*. Specifically, we initially cover edge buckets in the first column by swapping in each node partition in order of their ID. For example, as shown in Figure 5, we cover edge buckets  $\{(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)\}$  in column 0 by sequentially swapping in node partitions  $E_0, E_1, E_2, E_3, E_4, E_5$ . For subsequent columns, we swap in node partitions starting from the maximal ID in the current buffer state. If all edge buckets from the maximal ID to  $n$  are covered, we then switch to the minimal ID. For instance, after transitioning to column 1 with the buffer state  $\{E_1, E_5, E_4\}$ , we start with the node partition having the minimal ID to swap in, which is  $E_3$ . Since only one edge bucket  $(3, 1)$  in column 1 is not calculated, we move to column 2 after the buffer state  $\{E_1, E_5, E_3\}$  by loading  $E_2$ . For column 2, we again start with the minimal ID, which is 4, to swap in. The procedures are formalized in Algorithm 1.

In Algorithm 1, we first generate the buffer states related to node partition 0 (lines 3-6). In the while loop within lines 8-24, if all edge buckets in the current column  $CurCol$  have been covered, we advance to the next column by swapping node partition  $E_{CurCol}$  with  $E_{CurCol+1}$  and mark the corresponding edge buckets as covered (lines 11-13). Then we select the maximal ID in the current buffer as the node partition to evict, provided that subsequent IDs have corresponding edge buckets that remain uncovered. Otherwise, we opt for the minimal ID (lines 13-17). If the edge buckets in the current column have not been fully accessed, we evict a node partition that was not just swapped in the last buffer state (line 19). Finally, we greedily select a node partition that covers the most edge buckets from the  $BeginID$  to swap in (lines 20-24).

Algorithm 2 generates the edge bucket iteration order according to the output of Algorithm 1. It first covers the edge buckets related to the node partition scheduled for eviction in the next buffer state (lines 7-13). Subsequently, it calculates the edge buckets related to both the node partition that will be evicted and the one that was just swapped in (lines 14-19).

Although prefetching hides I/O overhead in the computation, it also raises the problem of whether the I/O overhead can be completely covered. To this end, Theorem 3 discusses this problem and proves that it has to do with the dataset characteristics.

**THEOREM 3.** *Using the loading order generated by Algorithm 2, the I/O overhead can be completely covered by the computation when  $\frac{|E|}{|V|^2} \geq \frac{96d^2}{M\ell(w+r)}$ , where  $|E|$  and  $|V|$  are the number of edges and nodes,  $d$*

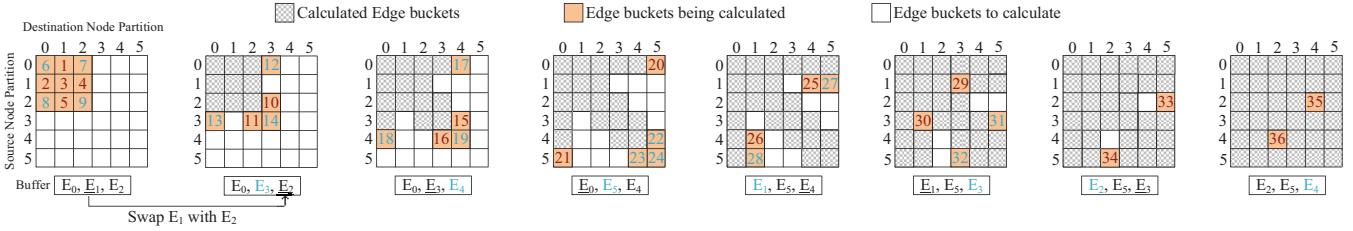


Figure 5: Order for prefetching in Legend. The numbers inside edge buckets denote their calculated order. The blue color indicates the edge buckets that can be calculated while prefetching the next partition.  $E_i$  is the prefetched partition.

---

**Algorithm 2:** Edge buckets iterating order

---

**Input:** Buffer states  $BuFS$ , node partitions  $n$   
**Output:** Edge buckets iterating order.

- 1  $EdgeBuckets \leftarrow \{0\}_{n \times n}$ ,  $BuFS \leftarrow \{\}$ ;
- 2  $IterOrder \leftarrow \{(0, 1), (1, 1), (1, 0), (1, 2), (2, 1)\}$ ;
- 3 Set the corresponding edge buckets of  $IterOrder$  to 1;
- 4  $LoadedPar \leftarrow 3$ ;
- 5 **for**  $i$  in range( $\text{len}(BuFS)-1$ ) **do**
- 6    $ToEvict \leftarrow BuFS[i] - BuFS[i+1]$ ;
- 7   **for**  $k \in BuFS[i] - \{LoadedPar\}$  **do**
- 8     **if**  $EdgeBuckets[ToEvict][k] = 0$  **then**
- 9        $EdgeBuckets[ToEvict][k] = 1$ ;
- 10       $IterOrder.append((ToEvict, k))$ ;
- 11     **if**  $EdgeBuckets[k][ToEvict] = 0$  **then**
- 12        $EdgeBuckets[k][ToEvict] = 1$ ;
- 13        $IterOrder.append((k, ToEvict))$ ;
- 14     **if**  $EdgeBuckets[ToEvict][LoadedPar] = 0$  **then**
- 15        $EdgeBuckets[ToEvict][LoadedPar] = 1$ ;
- 16        $IterOrder.append((ToEvict, LoadedPar))$ ;
- 17     **if**  $EdgeBuckets[LoadedPar][ToEvict] = 0$  **then**
- 18        $EdgeBuckets[LoadedPar][ToEvict] = 1$ ;
- 19        $IterOrder.append((LoadedPar, ToEvict))$ ;
- 20     $LoadedPar \leftarrow BuFS[i+1] - BuFS[i]$ ;
- 21 **return**  $IterOrder$

---

is the embedding dimension,  $M$  is the buffer size in the global memory of GPU,  $t$  is the average computing time of an edge,  $w$  and  $r$  are the writing and reading throughput between the GPU and NVMe SSD.

**PROOF.** Suppose the node embeddings are divided into  $n$  partitions. For each edge bucket, the average number of edges is  $\frac{|E|}{n^2}$ . Consequently, the average time to calculate an edge bucket is  $t * \frac{|E|}{n^2}$ . An exchange of a partition including writing the old partition into the NVMe SSD and loading the new one into the GPU buffer. Each partition contains embeddings and optimizer states, whose total size is  $2 * P$ . As a result, an exchange of a partition requires time of  $\frac{2*P}{w+r}$ . Following the order output by Algorithm 2 ensures that there are at least 2 edge buckets for computing during partition exchange. So if the inequality  $2t * \frac{|E|}{n^2} \geq 2 * \frac{P}{w+r}$ , the I/O overhead can be completely covered by the calculation of edge buckets. As the buffer size is  $M$  and it can contain 3 partitions in our hypothesis,  $P$  can be calculated as  $\frac{M}{6}$  and the minimum  $n$  can be calculated as  $\frac{|V|+d+4*2}{M/3}$ , where 4 denotes the bytes of a float type. Substituting  $P$  and  $n$  into the inequality yields  $\frac{|E|}{|V|^2} \geq \frac{96d^2}{MT(w+r)}$ .  $\square$

Theorem 3 displays the condition that I/O overhead can be completely covered by the computation using our prefetching strategy.

Legend has the metrics of  $t \approx 10^{-7}$  s,  $w \approx 2G/s$  and  $r \approx 3G/s$  in our experimental setting. With  $M = 15G$  and  $d = 100$ , the I/O overhead can be completely covered by the computation if  $\frac{|E|}{|V|^2} \geq 10^{-7}$ .

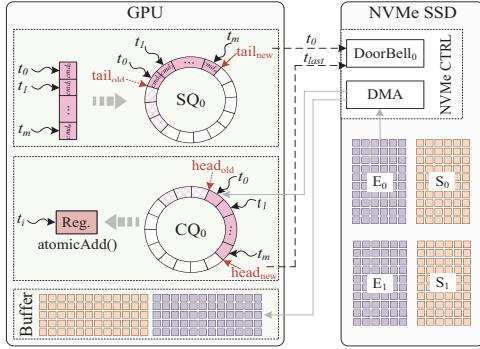
## 5 Optimizations on GPU Direct Access to SSD

In this section, we introduce the optimization for the GPU-SSD direct access mechanism, specifically designed for graph embedding workloads, enhancing the bandwidth between GPU and SSD.

Typically, access to NVMe SSD relies on the kernel I/O stacks of the operating system, which involve context switching, data copying, interrupts, resource synchronization, etc [55]. As the latency of storage devices decreases, the CPU software stack becomes a bottleneck for I/O access [37]. Consequently, research has shifted towards offloading I/O tasks from the CPU and reconstructing the user-level I/O stack on the GPU, aiming to reduce stacks' overhead and enhance throughput by leveraging the massive parallelism of GPU threads. Among these, BaM achieves the state-of-the-art performance [37]. However, BaM is designed to handle general workloads across various scenarios, incorporating complex mechanisms including parallel queue management strategies, atomic operations, caching strategies, etc. Moreover, BaM employs numerous thread blocks to achieve high throughput between the GPU and NVMe SSD, which consumes valuable GPU resources and hinders the simultaneous execution of the data access kernel and computing kernel. To implement a lightweight yet high-throughput NVMe SSD access kernel, we analyze the specific workload of graph embedding learning and optimize the direct access mechanism.

In the context of graph embedding learning, embeddings and optimizer states are loaded from NVMe SSD to the GPU buffer only after the GPU has completed the computation of the edge buckets related to the node partitions in the current buffer state. The data loading times are determined once Algorithm 2 provides the order. Additionally, the size of the embedding and optimizer states for each node partition is fixed, allowing for sequential access page by page. Such a workload leads to opportunities to reduce the complexity of the queue management mechanism.

To avoid building complex I/O stacks from scratch, similar to BaM, we implement the GPU-SSD direct access driver based on an open-source codebase [26]. We will only introduce our contributions below. We utilize several thread blocks, with each thread block managing one NVMe queue pair. All threads within a thread block enqueue and dequeue on the corresponding queue pair. This thread block allocation strategy simplifies the management of queue



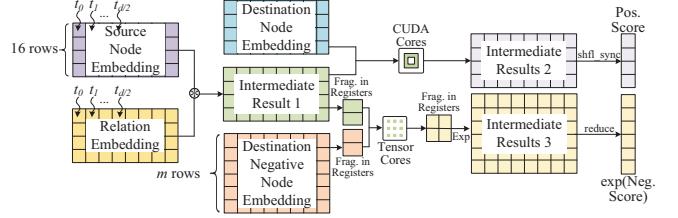
**Figure 6: Procedure of the GPU direct access to NVMe SSD.**

pairs, as synchronization among threads within the same block is straightforward and has low overhead.

Figure 6 depicts the optimized procedure for GPU direct access to NVMe SSD within a thread block. The key idea is to *utilize the regular embedding access characteristics to precompute the positions of queue entries and minimize the doorbell ringing time, avoiding the use of locks and atomic operations, as well as reducing the overhead of doorbell ringing*. During the commands construction phase, threads in a block construct read/write commands in parallel, requesting consecutive NVMe addresses. Each thread then inserts its command into the position  $tail+i$  of the corresponding submission queue (SQ), where  $tail$  denotes the current head pointer and  $i$  denotes the thread ID. This enqueue process is parallelized among threads since each thread has a unique and determined position in the SQ. The fixed size of embeddings simplifies the enqueue process, allowing parallel operation without the complex data structures and atomic operations in BaM for correct enqueueing. The doorbell registers in the NVMe controller are write-only, necessitating serial writing from threads. Furthermore, the writing overhead of doorbell registers is high because they are located in the NVMe SSD and the writing needs to be performed through PCIe. As a result, we assign a single thread to ring the doorbell after all the threads in a block have completed the enqueue process rather than ringing the doorbell multiple times.

Once the doorbell rings, the NVMe controller fetches commands from the SQ in the GPU’s global memory. The NVMe controller analyzes these commands, retrieves data from the NVMe SSD, and transfers the data to the specified addresses in the GPU buffer according to the commands via Direct Memory Access (DMA). Following this, the NVMe controller inserts an entry into the completion queue (CQ) corresponding to the entry in the SQ.

We employ a polling strategy, which is a common approach in existing NVMe SSD software stacks, to wait for the completion of data access by polling the CQ. During the polling phase, each thread within a thread block checks the position of  $head+i$  in the CQ, where  $head$  is the head pointer of the CQ and  $i$  denotes the thread ID. We maintain a counter for each CQ in GPU’s shared memory, which is initialized with 0. When a thread detects that an entry has been inserted by the NVMe controller, it atomically adds 1 to the counter. The last thread to increment this counter updates the head pointer of CQ and rings the doorbell to transfer the updated position of the latest head pointer. The atomic operation has a low cost because the counter is located in shared memory and



**Figure 7: Optimized procedure of the forward phase.**

the number of threads within a block is limited. This polling strategy can not only fully utilize thread parallelism, but also reduce the overhead of doorbell ringing.

As the size of required embeddings and optimizer states is determined, threads in a block repeat the data access procedures synchronously until all the pages of embedding and optimizer states in NVMe SSD are loaded.

## 6 Optimizations on GPU

In this section, we describe our optimization of the gradient computation on the GPU, which fully exploits resources on the GPU and significantly enhances GPU utilization.

As mentioned above, existing systems often overlook GPU computation optimization, leading to underutilization of the GPU. In graph embedding systems, the time cost can be divided into three parts: CPU processing, CPU-GPU communication, and GPU computing [59]. As some tasks on the CPU are offloaded to the GPU and CPU-GPU communication is optimized, the overhead of GPU computing becomes more pronounced. For instance, both Marius [30] and GE<sup>2</sup> [59] have similar GPU computing overhead because they use the same GPU computing engine. However, GE<sup>2</sup> offloads some tasks to the GPU and reduces the CPU-GPU communication cost by customized loading order, which results in the GPU computing becoming the most time-consuming part (more than 1/3). In Legend, the communication overhead is further reduced due to the node partition loading order that minimizes the I/O times as well as supports prefetching, which makes GPU computing the primary bottleneck of graph embedding learning. To optimize the computation, our key idea is to *implement a parallel pattern for graph embedding on the GPU to reduce memory access, maximize the reuse of the intermediate results, and leverage various GPU resources including Tensor cores and registers to enhance GPU utilization*.

As illustrated in Figure 7, we horizontally split a batch into several chunks, distributing each chunk across thread blocks in the GPU for simultaneous computation. Each thread block contains several warps, with  $\lceil d/64 \rceil$  warp(s) collaborating to process one row, where  $d$  represents the embedding dimension. This design allows the  $\lceil d/64 \rceil$  warp(s) to calculate the first half of the elements in a row before processing the last half. It leads to only one access to each element for the cross-calculation between the first and last half elements in some embedding models such as ComplEx [44]. It is also suitable for other embedding models without cross-calculation. Each thread block handles 16 rows of embeddings in a batch, as subsequent calculations utilize Tensor cores, which necessitate an input size of  $16 \times 8$  in each thread block.

Following the calculation of  $\theta_s \otimes \theta_r$ , we obtain Intermediate Results 1. To minimize the access times of global memory, we first use

these results stored in registers to compute the positive scores before writing them to global memory. We employ a similar parallel strategy to calculate  $(\theta_s \otimes \theta_r) \oplus \theta_d$ , yielding Intermediate Result 2 in Figure 7, where  $\otimes$  and  $\oplus$  are defined by the adopted embedding model. Notably, Intermediate Result 2 remains stored in registers and is distributed among threads. To calculate the positive scores from Intermediate Result 2, we implement a two-phase reduction strategy, which first reduces elements within threads in each warp using the inter-thread data exchange function `_shfl_sync()`, and second reduces the elements within warps in each row. The two-phase reduction strategy leverages shared memory only in the second phase, thereby reducing memory access overhead. During the calculation of positive scores, global memory access only happens during the data loading at the beginning and the positive scores writing at the end. Consequently, we improve the efficiency of positive score calculation by optimizing computation and memory access.

For the calculation of negative scores, we design an optimized kernel specifically for multiplication-based embedding models such as ComplEx [44] and DistMult [52], whose  $\oplus$  is a multiplication operation in  $(\theta_s \otimes \theta_r) \oplus \theta'_d$ . Normally, a source node embedding is required to perform element-wise multiplication with a group of negative node embeddings, followed by a reduction of elements in each row to calculate the negative scores. Given the substantial number of multiplication operations, we utilize Tensor cores, which can execute fix-size matrix multiplications within a single clock cycle. We adopt the TF32 data type for matrix multiplication in Tensor cores, requiring input matrices to be sized  $16 \times 8$ . As shown in Figure 7, in each thread block, the Intermediate Result 1 contains exactly 16 rows, facilitating horizontal iteration. In a thread block, we employ multiple warps to iterate over the negative node embeddings horizontally, with each warp handling 16 rows of the negative embeddings. Each warp loads a fragment of embeddings into registers and feeds them into the Tensor cores to get the multiplication results. Considering that we need to use the exponent results of the negative scores in the loss and gradients calculation, we perform the exponent operation in advance in registers before writing the results to global memory (Intermediate Result 3 in Figure 7). Finally, we employ the reduction API in libtorch to reduce the elements in Intermediate Result 3.

During the backward phase, we reuse the Intermediate Result 1, 2, and 3 to eliminate redundant calculations. We also apply the same parallel strategy as in the forward phase to compute the gradients in the backward phase. The parallel strategy, memory access strategy, and the intermediate results reusing perform collaboratively to enhance GPU utilization and enable high-performance gradient computation on large datasets.

## 7 Experiments

In this section, we evaluate the performance of our proposed Legend and conduct a comparative evaluation with state-of-the-art graph embedding learning systems.

### 7.1 Experiment Settings

**Datasets.** For comprehensive evaluations, we use 4 datasets with varying volumes, previously employed in related works [30, 59, 60].

**Table 2: Details of Datasets.**

Graphs	V	E	R	Dim.	E. Size
FB15k ( <i>FB</i> )	15k	592k	1345	100	13MB
LiveJournal ( <i>LJ</i> )	4.8M	68M	-	100	3.8GB
Twitter ( <i>TW</i> )	41.6M	1.46B	-	100	32GB
Freebase86M ( <i>FM</i> )	86.1M	304.7M	14824	100	68GB

Table 2 summarizes their properties, where *FB* and *FM* are multi-type knowledge graphs, while *LJ* and *TW* are social networks without relation types. In Table 2, *Dim.* denotes the embedding dimension and *E. Size* indicates the storage requirements for embedding and optimizer states. Each dataset is divided into training, test, and validation subsets for embedding training and evaluation.

**Embedding models.** Following Marius and GE<sup>2</sup>, on dataset *LJ* and *TW* we employ the popular model Dot [21] as they lack relation types. On *FB* and *FM*, we utilize ComplEx [44].

**Baseline systems.** We compare Legend with two state-of-the-art graph embedding learning systems, i.e., Marius [30] and GE<sup>2</sup> [59], which are disk-based and RAM-based systems respectively. Among the two methods, Marius supports only a single GPU, while GE<sup>2</sup> can leverage multiple GPUs. We exclude DGL-KE [60] and PBG [20] from the comparison, as Marius and GE<sup>2</sup> have been demonstrated to outperform them. To ensure a fair comparison, we maintain identical hyperparameters across the three graph embedding learning systems, including a learning rate of 0.1, a batch size of  $10^5, 10^3$  negative samples per positive edge, 10 epochs for *TW* and *FM*, 30 epochs for *FB* and *LJ*, etc. Legend and Marius use 8 node partitions with a buffer capacity of 3 (12GB of the GPU global memory) for *TW* and 12 node partitions with a buffer capacity of 3 (17GB of the GPU global memory) for *FM*. GE<sup>2</sup> uses 16 node partitions and a buffer capacity of 4 on *TW* and *FM* because it only supports the number of partitions of  $4^L$  and a fixed buffer capacity of 4. This comparison is fair as the restricted support for flexible partition numbers is exactly the limitation of GE<sup>2</sup>. We also apply the order in GE<sup>2</sup> with 16 partitions to Legend, as referenced in Table 7.

**Metrics.** We employ Mean Reciprocal Rank (MRR) and Hits@*k* as the quality metrics, which are widely used to evaluate the embeddings [19, 30, 36, 59]. Higher MMR and Hits@*k* values indicate better embedding quality. Consistent with GE<sup>2</sup>, we utilize part of test edges ( $10^6$ ) to compute MRR and Hit@*k*, as using the entire test set would be time-prohibitive.

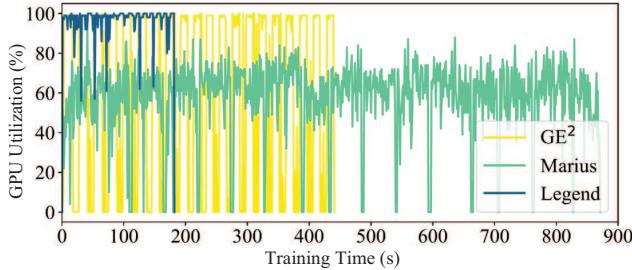
**Platforms.** All experiments are conducted on an Ubuntu 20.04 server, featuring an Intel Xeon Silver 4216 CPU@2.10GHz with 64 cores, an Nvidia A100 GPU (40G), and a Samsung 980 NVMe SSD (1T). We implement Legend in C++/CUDA under Nvidia CUDA 11.1 and LibTorch 1.7.1. Legend can be easily integrated into Pytorch by pybind but the host running Pytorch needs to be equipped with an NVMe SSD and a GPU supporting GPUDirect RDMA.

### 7.2 Comparisons with Existing Systems

Firstly, we evaluate the overall performance of the compared systems. The training performance and overhead for the three systems using a single GPU are reported in Table 3. The time reported in Table 3 is the average epoch duration. Notably, we omit the execution time of Marius on *FB* due to a floating point exception

**Table 3: Results of the systems using a single GPU.**

Graphs	Model	System	MRR	Hit@10	Time (s)
FB	ComplEx	Marius	<b>0.582</b>	0.802	-
		GE <sup>2</sup>	0.573	0.801	0.17 (2.4×)
		<b>Legend</b>	0.581	<b>0.806</b>	<b>0.07</b>
LJ	Dot	Marius	0.746	0.879	12.2 (1.7×)
		GE <sup>2</sup>	0.733	<b>0.880</b>	13.6 (1.9×)
		<b>Legend</b>	<b>0.747</b>	0.879	<b>7.1</b>
TW	Dot	Marius	<b>0.414</b>	<b>0.576</b>	872.7 (4.8×)
		GE <sup>2</sup>	0.312	0.487	439.3 (2.4×)
		<b>Legend</b>	0.398	0.555	<b>181.0</b>
FM	ComplEx	Marius	0.725	0.762	409.7 (1.7×)
		GE <sup>2</sup>	0.731	0.772	315.5 (1.3×)
		<b>Legend</b>	<b>0.764</b>	<b>0.829</b>	<b>243.8</b>



**Figure 8: GPU utilization of Legend, GE<sup>2</sup> and Marius on TW.**

with the batch size of  $10^5$  while reporting the performance with the batch size of 50000. On average, Legend achieves a speedup of  $2.7\times$  over Marius and  $2.0\times$  over GE<sup>2</sup> while maintaining similar embedding quality. In optimal scenarios, Legend achieves a remarkable speedup of  $4.8\times$  over Marius on *TW* and  $2.4\times$  speedup over GE<sup>2</sup> on *TW* and *FB*. It's worth noting that GE<sup>2</sup> stores embeddings and optimizer states in RAM, while Legend stores them in the NVMe SSD. Legend exhibits excellent scalability and efficiency on the four datasets with various volumes. This is attributed to the optimization of each hardware component and the workflow that orchestrates each hardware in the heterogeneous system by making full use of their unique characteristics. Although the systems load all data into the GPU memory without I/O overhead during embedding learning on datasets *FB* and *LJ*, Legend still demonstrates superior training speed. This indicates that the workflow and optimizations on GPU contribute to accelerating the training process except for the I/O optimization. As introduced in Section 3, Legend loads entire node partitions into the global memory of GPU and constructs batches on the GPU, ensuring that updated embeddings from the last batch can be used immediately, avoiding the problem of staleness present in Marius. As a result, Legend achieves better performance on *FM* in MRR and Hit@10.

On *FM*, the speedup of Legend is relatively insignificant, which is due to the graph properties. The number of edges in *FM* is relatively small compared to the number of vertices, where  $\frac{|E|}{|V|^2} \approx 4 \times 10^{-8} < 10^{-7}$ . According to Theorem 3, the I/O overhead between the GPU and NVMe SSD cannot be entirely covered by computation. Furthermore, the I/O times can reach 36 even though the node partition ordering algorithm is applied, exacerbating the I/O

**Table 4: Performance on various number of GPUs on TW.**

Systems	GPU(s)	MRR	Hit@10	Time (s)
GE <sup>2</sup>	1	0.312	0.487	439.3 (2.43×)
	2	0.299	0.473	315.2 (1.74×)
	4	0.284	0.456	192.5 (1.06×)
<b>Legend</b>	<b>1</b>	<b>0.398</b>	<b>0.555</b>	<b>181.0</b>

overhead. In contrast, the training speed of Legend is more significant on *TW*. Using the records in Table 2,  $\frac{|E|}{|V|^2} \approx 8 \times 10^{-7} > 10^{-7}$  on *TW*, which indicates the I/O overhead can be covered by computation. Consequently, this alleviates the bandwidth constraints between the GPU and NVMe SSD, leading to improved performance.

To further validate the GPU utilization improvements from our proposed techniques, we assess GPU utilization on dataset *TW*. Figure 8 depicts the variation in GPU utilization across the three systems over time. The average GPU utilization of Legend is 96.79%, compared to 57.63% for Marius and 59.85% for GE<sup>2</sup>, even with prefetching enabled. As shown in Figure 8, GPU utilization periodically drops to zero for Marius and GE<sup>2</sup>, indicating that the GPU is idle during data loading from the disk or RAM. In contrast, GPU utilization of Legend remains consistently above 55%, exceeding 90% for most of the duration. This enhanced utilization can be attributed to three key factors. First, we offload batch construction and negative sampling to the GPU, which improves the batch constructing speed. Second, we prefetch the node embeddings and optimize the bandwidth between the GPU and NVMe SSD, which reduces the data transfer overhead and the GPU idle time. Third, we further optimize the training on the GPU by customized parallel strategy and data reuse to maximize the resource utilization of the GPU.

Please note that the partition order in Legend doesn't support prefetching across multiple GPUs, and accessing data from a single NVMe SSD to multiple GPUs adversely affects the throughput. We leave the support to multi-GPU for future work. Nevertheless, we compare Legend on a single GPU with GE<sup>2</sup> on multi-GPU using *TW*. Table 4 presents the results. Legend exhibits superior performance compared to GE<sup>2</sup>. Particularly, when GE<sup>2</sup> employs 4 GPUs, Legend still shows comparable performance. Note that as the number of GPUs increases, the time overhead of GE<sup>2</sup> does not decrease proportionally. This phenomenon arises from the limited I/O bandwidth between host and device memory, which constrains data transfer rates to multiple GPUs. This issue can be mitigated by employing NVMe SSD as the primary data storage device. Since the NVMe SSD is much cheaper than RAM, it is feasible to allocate one NVMe SSD per GPU, thereby eliminating competition for limited bandwidth, which represents a promising direction for future research.

### 7.3 Evaluations of the Workflow

To demonstrate the superiority of our proposed workflow, we first evaluate the average batch time across the three systems. Batch time encompasses the entire process for a batch, from batch construction to embedding updates. The results are presented in Table 5. On dataset *TW* and *FM*, Marius constructs batches on the CPU and subsequently transfers them to the GPU, resulting in considerable communication overhead. In contrast, both Legend and GE<sup>2</sup>

offload the tasks of batch construction and negative sampling to the GPU, which achieves significant speedups.

However, Batch time only partially reflects the advantages of our workflow. To conduct a comprehensive evaluation, we omit all optimization modules that can be removed, including the modules of GPU optimization, edge bucket iteration order, and prefetching mechanism. The remaining components can completely reflect the performance of the workflow. The epoch times on *FB*, *LJ*, *TW* and *FM* are 0.12s, 13.06s, 291.89s and 331.40s, respectively. Compared with the epoch time in Table 3, it still exhibits superiority over Marius and GE<sup>2</sup> on most datasets.

#### 7.4 Prefetch-friendly Order

Prefetching is one of the key strategies that alleviates the limited bandwidth between the NVMe SSD and GPU. To evaluate the effectiveness of prefetching, we compare the performance of Legend with and without prefetching on *TW* and *FM*. The results are reported in Table 6. Legend benefits more from prefetching on *TW* than on *FM*, which can be attributed to the properties of the graphs. As calculated in subsection 7.2,  $\frac{|E|}{|V|^2} \approx 4 \times 10^{-8}$  for *FM* while  $\frac{|E|}{|V|^2} \approx 8 \times 10^{-7}$  for *TW*. The sparsity of *FM* results in an incomplete covering of I/O overhead, leading to reduced benefits from prefetching. Nonetheless, prefetching remains effective on *FM*, demonstrating the scalability of the prefetching strategy in Legend.

Furthermore, we apply the order used in Marius named BETA and GE<sup>2</sup> named COVER to Legend to demonstrate the effectiveness of our prefetch-friendly order. Using the same settings as in subsection 7.2, BETA and COVER divide the node embeddings into 8 and 16 partitions for *TW*, and into 12 and 16 partitions for *FM*. BETA has the buffer capacity of 3, while COVER has a buffer capacity of 4. The results are summarized in Table 7. Recall that the prefetch-friendly order generating algorithm aims to generate an order that supports prefetching while minimizing I/O times. A comparison among BETA, COVER and Legend without prefetching reveals the comparable I/O overhead between Legend(w/o prefetching) and BETA, which highlights the I/O efficiency of our proposed order. Although BETA has I/O times close to the theoretical lower bound, its design is not conducive to prefetching, as discussed in Section 4. In contrast, the ordering algorithm used in Legend exhibits similar I/O overhead while supporting effective prefetching. Additionally, COVER used in GE<sup>2</sup> has higher I/O overhead when applied to Legend. This is because it is specifically designed for training with multiple GPUs, which is not optimized for single GPU scenarios.

In Table 8, we summarize the I/O times and calculate the communication volume for the three ordering algorithms. Since COVER can only accommodate partition numbers of  $4^L$ , we report its metrics when the number of partitions is 16. As shown in Table 8, BETA and Legend have similar I/O times and communication volumes within the evaluated partitions. COVER is adopted by GE<sup>2</sup> to overcome the issue of I/O overhead within multiple GPUs. It is not optimized for a single GPU. In contrast, the communication volume remains unchanged with the increasing number of GPUs. Devising a prefetching-friendly and low-overhead ordering algorithm that supports multiple GPUs like COVER is left to future work.

**Table 5: Average batch time of the compared systems.**

Systems	<i>FB</i>	<i>LJ</i>	<i>TW</i>	<i>FM</i>
Marius	-	19.1ms	315.6ms	326.4ms
GE <sup>2</sup>	32.8ms	18.3ms	18.5ms	38.2ms
<b>Legend</b>	<b>12.1ms</b>	<b>12.1ms</b>	<b>12.0ms</b>	<b>13.8ms</b>

**Table 6: Epoch time of Legend with and without prefetching.**

Graphs	w/o Prefetching	Prefetching	Speedup
<i>TW</i>	235.0s	181.0s	29.83%
<i>FM</i>	271.2s	243.8s	11.24%

**Table 7: Epoch time of replacing the edge buckets iterating order in Legend with BETA and COVER.**

Graphs	BETA	COVER	Legend(w/o Prefetching)	Legend
<i>TW</i>	233.6s	276.6s	235.0s	<b>181.0s</b>
<i>FM</i>	273.8s	314.2s	271.2s	<b>243.8s</b>

**Table 8: I/O times and communication volume of different ordering algorithms with various numbers of partitions. *S* denotes the size of node embeddings and optimizer states.**

Par.	I/O times			Communication volume		
	BETA	COVER	Legend	BETA	COVER	Legend
6	8	-	8	1.33S	-	1.33S
8	15	-	16	1.88S	-	2S
10	24	-	24	2.4S	-	2.4S
12	34	-	36	2.83S	-	3S
14	48	-	50	3.43S	-	3.57S
16	63	80	66	3.94S	5S	4.13S

#### 7.5 GPU Direct Access to NVMe SSD

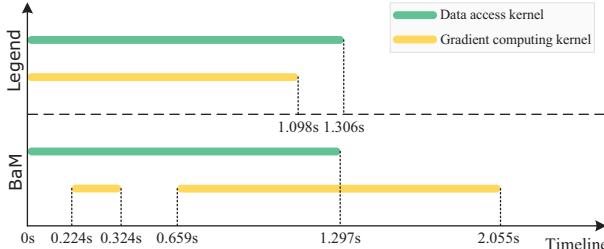
As discussed in Section 5, we aim to design a GPU direct access strategy to NVMe SSD that achieves high performance as well as supports the simultaneous execution of data access and gradient calculation kernels. To this end, we separately evaluate the bandwidth of read/write and the ability to simultaneously execute together with the calculation kernel.

We first compare the bandwidth of Legend with the state-of-the-art GPU direct access method named BaM. Moreover, we also evaluate the bandwidth between the CPU and GPU achieved by GE<sup>2</sup>. For our evaluation, the test data volume is set to 4GB. We employ 4096 thread blocks for BaM, each containing 32 threads, while for Legend, we employ 8 thread blocks, each containing 512 threads. We also evaluate BaM with the same settings as Legend, referred to as BaM (light). The results are presented in Table 9. Legend achieves comparable I/O bandwidth to BaM. Especially, the writing bandwidth of Legend outperforms BaM due to the high parallel queue management mechanism and the low-cost doorbell ringing strategy. Under the same settings, Legend achieves higher I/O bandwidth than BaM (light). For GE<sup>2</sup>, the bandwidth between the CPU and GPU is over 3 times higher than that between the GPU and NVMe SSD. This gap can be mitigated by carefully prefetching data, as demonstrated in our previous experiments.

To evaluate the capability of the data access kernel to execute concurrently with the gradient computing kernel, we run both kernels simultaneously by using CUDA streams. For the gradient computing kernel, we fix the batch size at  $10^5$  and execute the batch

**Table 9: Bandwidth of GPU direct access to SSD (GB/s). GE<sup>2</sup> is reported as the bandwidth between CPU and GPU.**

Access type	BaM	BaM (light)	Legend	GE <sup>2</sup> (CPU-GPU)
Read	3.20	2.59	3.19	10.05
Write	1.64	2.05	2.24	11.93



**Figure 9: Timeline of simultaneous execution of kernels.**

computation 100 times. The execution timelines are depicted in Figure 9. In Legend, both kernels can be executed concurrently with minimal performance degradation. In contrast, the data access kernel in BaM occupies a significant amount of resources, seriously affecting the execution of the gradient computing kernel. By considering the specific workload of graph embedding learning, we have simplified the complexity of the GPU direct access strategy, resulting in a lightweight but high-performance data access kernel.

## 7.6 Optimizations on the GPU

In this subsection, we evaluate the optimization techniques applied to the GPU, as proposed in Section 6. To achieve this, we measure the average gradient calculation time per batch, which includes both forward and backward computations. The results are reported in Table 10. As Marius exists a floating point exception with batch size  $10^5$  on *FB*, we omit its result on *FB*. Marius and GE<sup>2</sup> exhibit similar performance across the four datasets, as they utilize the same training engine. The training overhead for both systems on *FB* and *FM* is greater compared to *LJ* and *TW*. This discrepancy arises because these datasets are different types of graphs and employ distinct embedding models, as discussed in subsection 7.1. Specifically, *FB* and *FM* are knowledge graphs with multiple types of edges, which use ComplEx model. They calculate embeddings of edges while *LJ* and *TW* don't. In Legend, we combine the gradient calculation process for edge embeddings with that of node embeddings, eliminating redundant calculation. As a result, the calculation overhead remains consistent across the 4 datasets.

On *LJ* and *TW*, Legend achieves a speedup ratio of  $1.5\times$ - $1.6\times$ , while the speedup exceeds  $2\times$  on *FB* and *FM*. This indicates that the parallel strategy and intermediate results reuse techniques proposed in Section 6 are effective. Furthermore, during the training process, the GPU utilization for Legend remains above 98% for 81.22% of the time, above 99% for 59% of the time, and reaches 100% for 31.49% of the time. In contrast, the proportions of time during which GPU utilization exceeds 98%, 99%, and 100% for GE<sup>2</sup> are 51.48%, 43.28%, and 3.64%, respectively. Thus, the optimization of training on the GPU significantly improves GPU utilization, leading to a substantial reduction in calculation overhead.

**Table 10: Average gradient calculation time of a batch.**

Systems	<i>FB</i>	<i>LJ</i>	<i>TW</i>	<i>FM</i>
Marius	-	15.6ms	16.0ms	27.3ms
GE <sup>2</sup>	24.1ms	16.6ms	16.3ms	25.4ms
<b>Legend</b>	<b>10.2ms</b>	<b>10.5ms</b>	<b>10.5ms</b>	<b>10.4ms</b>

## 8 Related Work

**Graph embedding learning algorithms.** Extensive studies have been conducted to enhance the quality of embeddings. For general graphs, existing algorithms typically sample edges based on random walks, and employ the idea of Word2Vec [29] to train the embedding by skip-gram model [13, 35, 38, 42, 45]. In the context of multi-relation graphs, all edges are used for embedding learning without sampling [20, 30, 59, 60]. Extensive multi-relation graph embedding models have been developed, categorized into two primary types: translational distance models and semantic matching models [46]. Translational distance models, such as TransE [3] and TransH [48], employ distance-based scoring functions to evaluate the plausibility of facts between entities. Semantic matching models, such as DistMult [52] and ComplEx [44], utilize similarity-based scoring functions to assess plausibility based on entities semantics. These multi-relation graph embedding models can be easily integrated into Legend.

**Graph embedding learning systems.** Significant efforts have been dedicated to developing efficient systems for graph embedding training. GraphVite [64] employs CPU to generate random walks and samples negative edges on GPUs to achieve efficient embedding learning on general graphs. HET [28] capitalizes on the skewed distributions of embeddings and proposes an embedding cache strategy to reduce the communication overhead. DistGER [10] proposes an efficient distributed graph embedding system. For massive knowledge graphs, PBG [20] proposes a batched negative sampling method to reduce memory access overhead. DGLKE [60] overlaps the gradient update with batch processing to reduce GPU idle time. Kochsieck et al. provide a comprehensive experimental study of the existing knowledge graph embedding training techniques [19]. Following PBG, Marius [30] proposes a partition loading order BETA to reduce the I/O times and pipelines the training procedure on the CPU and GPU. GE<sup>2</sup> [59] designs a general negative sampling execution model, and proposes a loading order to reduce I/O overhead between RAM and GPUs. Different from them, our proposed Legend employs GPU-SSD direct access and prefetching supported order to optimize the I/O efficiency, while utilizing a customized GPU kernel to optimize the computing efficiency.

**GPU direct access to NVMe SSD.** Recent research has studied the GPU-SSD direct access to meet the demand for low latency and large capacity. GPUDirect Storage (GDS) [31] is a library supporting data transmission between GPU and NVMe SSD through a bounce buffer in the CPU's memory and a direct memory access (DMA) engine. However, GDS is still restricted by the high overhead software stacks. To completely break free from the limitations of the Linux software stacks, BaM [37] proposes a queue management mechanism and caching strategy completely on the GPU to achieve high-throughput access to storage. There are also

various customized GPU direct access methods for specific applications such as DNN training [2], GNN training [33], vector retrieval [16], and data analysis in OLAP [23]. Although Legend also employs the GPU-SSD direct access technique, it customizes the queue management mechanism for the GPU-SSD direct access driver according to the graph embedding workload to achieve higher bandwidth.

## 9 Conclusion

We introduce Legend, a lightweight heterogeneous embedding system for massive graphs. Legend systematically integrates CPU, GPU, and NVMe SSD resources, which performs efficient and scalable embedding training. We carefully design the workflow to enable a seamless introduction of the NVMe SSD into the system and distribute tasks according to the unique characteristics of each hardware component. Meanwhile, we design an edge bucket iteration order that minimizes the I/O times between the GPU and NVMe SSD while supporting efficient prefetching, as well as a customized GPU-SSD direct access driver to significantly reduce I/O overhead. Last but not least, we propose an efficient parallel strategy for graph embedding learning workload to optimize the computation on the GPU, ensuring efficient handling of billion-scale datasets. Experimental results consistently demonstrate the superiority of Legend.

## References

- [1] Shubhangi Agarwal, Sourav Dutta, and Arnab Bhattacharya. 2020. Chisel: Graph similarity search using chi-squared statistics in large probabilistic graphs. *PVLDB* 13, 10 (2020), 1654–1668.
- [2] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch training of very deep neural networks. In *FAST*. 387–401.
- [3] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. *NeurIPS* 26 (2013).
- [4] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *TKDE* 30, 9 (2018), 1616–1637.
- [5] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *PVLDB* 17, 3 (2023), 441–454.
- [6] Alberto Caprara, Paolo Toth, and Matteo Fischetti. 2000. Algorithms for the set covering problem. *Annals of Operations Research* 98, 1 (2000), 353–371.
- [7] Fangshu Chen, Yufei Zhang, Lu Chen, Xiankai Meng, Yanqiang Qi, and Jiahui Wang. 2023. Dynamic traveling time forecasting based on spatial-temporal graph convolutional networks. *FCS* 17, 6 (2023), 176615.
- [8] Kewei Cheng, Xian Li, Yifan Ethan Xu, Xin Luna Dong, and Yizhou Sun. 2022. PGE: robust product graph embedding learning for error detection. *PVLDB* 15, 6 (2022), 1288–1296.
- [9] Sara Cohen. 2016. Data management for social networking. In *PODS*. 165–177.
- [10] Peng Fang, Arijit Khan, Siqiang Luo, Fang Wang, Dan Feng, Zhenli Li, Wei Yin, and Yuchao Cao. 2023. Distributed graph embedding with information-oriented random walks. *PVLDB* 16, 7 (2023), 1643–1656.
- [11] Xiaokun Fang, Feng Zhang, Junxiang Nong, Mingxing Zhang, Puyun Hu, Yunpeng Chai, and Xiaoyong Du. 2024. Enabling Efficient NVM-Based Text Analytics without Decompression. In *ICDE*. 3725–3738.
- [12] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *PVLDB* 13, 6 (2020), 854–867.
- [13] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *SIGKDD*. 855–864.
- [14] Gabriel Haas and Viktor Leis. 2023. What modern NVMe storage can do, and how to exploit it: high-performance I/O for high-performance storage engines. *PVLDB* 16, 9 (2023), 2090–2102.
- [15] Yuntian He, Yue Zhang, Saket Gurukar, and Srinivasan Parthasarathy. 2022. WebMILE: democratizing network representation learning at scale. *PVLDB* 15, 12 (2022).
- [16] Yuchen Huang, Xiaopeng Fan, Song Yan, and Chuliang Weng. 2024. Neos: A NVMe-GPUs Direct Vector Service Buffer in User Space. In *ICDE*. 3767–3781.
- [17] Ihab F Ilyas, JP Lacerda, Yunyao Li, Umar Farooq Minhas, Ali Mousavi, Jeffrey Pound, Theodoros Rekatsinas, and Chiraag Sumanth. 2023. Growing and Serving Large Open-domain Knowledge Graphs. In *SIGMOD*. 253–259.
- [18] Junghoon Kim, Tao Guo, Kaiyu Feng, Gao Cong, Arijit Khan, and Farhana M Choudhury. 2020. Densely connected user community and location cluster search in location-based social networks. In *SIGMOD*. 2199–2209.
- [19] Adrian Kochsieb and Rainer Gemulla. 2021. Parallel training of knowledge graph embedding models: a comparison of techniques. *PVLDB* 15, 3 (2021), 633–645.
- [20] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large scale graph embedding system. *MLSys* 1 (2019), 120–131.
- [21] Jure Leskovec. 2018. Tutorial: Representation Learning on Networks. <http://snap.stanford.edu/proj/embeddings-www>.
- [22] Cheng-Ti Li, Yu-Chi Tsai, and Jay Chieh-Liao. 2023. Graph neural networks for tabular data learning. In *ICDE*. 3589–3592.
- [23] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *PVLDB* 9, 14 (2016), 1647–1658.
- [24] Xiaodong Li, Reynold Cheng, Kevin Chen-Chuan Chang, Caihua Shan, Chenhao Ma, and Hongtai Cao. 2021. On analyzing graphs with motif-paths. *PVLDB* 14, 6 (2021).
- [25] Zoltán Ádám Mann. 2017. The top eight misconceptions about NP-hardness. *Computer* 50, 5 (2017), 72–79.
- [26] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvæle Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-Overhead Device Sharing through PCIe Networking. *TOCS* 38 (2021).
- [27] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A graph-based system approach to scaling large embedding model training. In *SIGMOD*. 470–480.
- [28] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *PVLDB* 15, 2 (2021), 312–320.
- [29] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR Workshop*.
- [30] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivararam Venkataraman. 2021. Marius: Learning massive graph embeddings on a single machine. In *OSDI*. 533–549.
- [31] Nvidia. 2019. GPUDirect Storage. <https://developer.nvidia.com/blog/gpudirect-storage>.
- [32] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.
- [33] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *PVLDB* 17, 6 (2024), 1227–1240.
- [34] Yeonghong Park, Sunhong Min, and Jae W Lee. 2022. Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. *PVLDB* 15, 11 (2022), 2626–2639.
- [35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *SIGKDD*. 701–710.
- [36] Jiezhang Qiu, Laxman Dhulipala, Jie Tang, Richard Peng, and Chi Wang. 2021. Lightne: A lightweight graph processing system for network embedding. In *SIGMOD*. 2281–2289.
- [37] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *ASPLoS*. 325–339.
- [38] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. 2017. struc2vec: Learning node representations from structural identity. In *SIGKDD*. 385–394.
- [39] Siddhartha Sahu, Amine Mhedbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB* 29 (2020), 595–618.
- [40] Zixing Song, Yifei Zhang, and Irwin King. 2022. Towards an optimal asymmetric graph structure for robust semi-supervised node classification. In *SIGKDD*. 1656–1665.
- [41] Chenchen Sun, Yan Ning, Derong Shen, and Tiezheng Nie. 2023. Graph Neural Network-Based Short-Term Load Forecasting with Temporal Convolution. *DSE* (2023), 1–20.
- [42] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW*. 1067–1077.
- [43] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *PVLDB* 8, 4 (2014), 449–460.

- [44] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *International conference on machine learning*. PMLR, 2071–2080.
- [45] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *SIGKDD*, 839–848.
- [46] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *TKDE* 29, 12 (2017), 2724–2743.
- [47] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *SIGMOD*, 2628–2638.
- [48] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *AAAI*, Vol. 28.
- [49] Zhihao Wen, Yuan Fang, and Zemin Liu. 2021. Meta-inductive node classification across graphs. In *SIGIR*, 1219–1228.
- [50] Lianghao Xia, Chao Huang, Yong Xu, Peng Dai, Mengyin Lu, and Liefeng Bo. 2021. Multi-behavior enhanced recommendation with cross-interaction collaborative relation modeling. In *ICDE*, 1931–1936.
- [51] Xiujuan Lei Xiaoxuan Zhang. 2025. Predicting miRNA-drug interactions via dual-channel network based on TCN and BiLSTM. *FCS* 19, 5 (2025), 195905.
- [52] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *ICLR*.
- [53] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *PVLDB* 13, 12 (2020), 1976–1989.
- [54] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous network embedding for massive graphs via reweighted personalized PageRank. *PVLDB* 13, 5 (2020), 670–683.
- [55] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *IEEE CLOUD*, 154–161.
- [56] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *PVLDB* 17, 6 (2024), 1241–1254.
- [57] Jun Zhang, Chen Gao, Depeng Jin, and Yong Li. 2021. Group-buying recommendation for social e-commerce. In *ICDE*, 1536–1547.
- [58] Lingling Zhang, Shaowei Wang, Jun Liu, Xiaojun Chang, Qika Lin, Yaqiang Wu, and Qinghua Zheng. 2022. MuL-GRN: Multi-level graph relation network for few-shot node classification. *TKDE* 35, 6 (2022), 6085–6098.
- [59] Chengguang Zheng, Guanxian Jiang, Xiao Yan, Peiqi Yin, Qihui Zhou, and James Cheng. 2024. GE2: A General and Efficient Knowledge Graph Embedding Learning System. *SIGMOD* 2, 3 (2024), 1–27.
- [60] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. Dgl-ke: Training knowledge graph embeddings at scale. In *SIGIR*, 739–748.
- [61] Zhiqiang Zhong and Davide Mottin. 2023. Knowledge-augmented Graph Machine Learning for Drug Discovery: From Precision to Interpretability. In *SIGKDD*, 5841–5842.
- [62] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *MICRO*, 359–371.
- [63] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: A comprehensive graph neural network platform. *PVLDB* 12, 12 (2019), 2094–2105.
- [64] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *WWW*, 2494–2504.
- [65] Farzaneh Zirak, Farhana Choudhury, and Renata Borovica-Gajic. 2024. SeLeP: Learning Based Semantic Prefetching for Exploratory Database Workloads. *PVLDB* 17, 8 (2024), 2064–2076.
- [66] Yuanhang Zou, Zhihao Ding, Jieming Shi, Shuting Guo, Chunchen Su, and Yafei Zhang. 2023. EmbedX: A Versatile, Efficient and Scalable Platform to Embed Both Graphs and High-Dimensional Sparse Data. *PVLDB* 16, 12 (2023), 3543–3556.