FlexGM: An Adaptive Runtime System to Accelerate Graph Matching Networks on GPUs

Yue Dai
Computer Science Department
University of Pittsburgh
Pittsburgh, PA
Email: yud42@pitt.edu

Xulong Tang
Computer Science Department
University of Pittsburgh
Pittsburgh, PA
Email: xulongtang@pitt.edu

Youtao Zhang
Computer Science Department
University of Pittsburgh
Pittsburgh, PA
Email: zhangyt@cs.pitt.edu

2023 IEEE 41st International Conference on Computer Design (ICCD) | 979-8-3503-4291-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCD58817.2023.00060

***Abstract*—GMNs (Graph Matching Networks) exploit recently developed GNNs (Graph Neural Networks) to analyze the similarity between two graphs. They are increasingly deployed in many application domains due to their improved inference accuracy. A GMN consists of two stages, i.e., node-embedding and node-matching stages. The node-matching stage matches node features from two graphs for similarity, which accounts for over 90% of the total execution time. However, it is challenging to accelerate GMNs on GPUs due to their diverse computing patterns for different graph inputs. For large graphs, the overhead comes mainly from the high computation overhead, which increases quadratically to the size of the graphs; for small graphs, the overhead comes from the low parallelism and resource utilization.**

In this paper, we propose FlexGM, a flexible runtime, to adaptively accelerate GMNs on GPUs. For large graphs, we exploit the massive computation redundancy in GMNs and develop a low-overhead deduplication module to mitigate the high computation overhead. For small graphs, we develop a unified matching module to optimize GPU hardware resource usage. An adaptive module manager is then developed to judiciously select beneficial optimization strategies. Experimental results show that the FlexGM system achieves 2.5× (up to 7.6×) average speedup over existing methods.

***Index Terms*—Graph matching networks, Graph neural networks, GPU runtime**

I. Introduction

Graph Similarity Computing plays a pivotal role in numerous application domains, such as bioinformatic analysis in medical science [1], friend cycle matching in social networks [2], and feature matching in the computer vision [3], [4]. Recently introduced Graph Matching Networks (GMNs) demonstrate outstanding inference accuracy on these tasks [3], [5]–[7]. GMNs comprise two stages: The first, the embedding stage, integrates subgraph information into node features, while the second, the matching stage, calculates similarities between node features for diverse matching purposes. Generally, the embedding stage is facilitated by Graph Neural Networks (GNNs), which take advantage of existing GNN optimizations [8]–[11]. However, the matching stage, which requires all-to-all node matching between pairs of various-sized graphs, comprises over 90% execution latencies across distinct models and datasets, identifying it as the primary bottleneck in most GMN applications. Despite this, many applications necessitate a rapid response time or high throughput. For instance, computer vision tasks require real-time graph matching with strict deadlines [12]–[14], and the graph searching requires handling millions of matching queries in a short time [14]–[16]. These GMN applications are typically deployed on servers equipped with GPUs. Therefore, developing a GMN-accelerating runtime system for the GPUs is crucial and highly beneficial.

Nevertheless, designing a runtime system for GMNs is challenging due to the diversity in computing patterns exhibited by graphs of varying sizes. The inefficiency comes from two aspects: First, the all-to-all node matching suffers quadratic computing complexity, thus introducing heavy overheads when the input graphs are large. However, there exists tremendous redundancy in GMN computing. In particular, nodes tend to have identical neighborhoods, resulting in the same features and matching results. Consequently, redundant computations are introduced by repeatedly calculating these identical matchings. Therefore, detecting and eliminating those redundant computations could be beneficial for handling large graph inputs. Nevertheless, when the graphs are small, the overhead of detecting and eliminating those redundant computations could outweigh the benefits. Secondly, the GPU hardware resources are underutilized when the input graphs are small. Specifically, as varying sizes of graph pairs pose challenges to batching, current GMN implementations tend to compute the similarities between different graph pairs serially. Therefore, when a single graph pair is not large enough to saturate the GPU hardware resources, the GPU resources will be underutilized and cannot contribute to shorter execution latency. One potential solution is to parallel the node matching across different-sized graphs using Sparse Matrix Multiplication (SpMM). However, for large graphs that fully utilize GPU resources, the overhead of managing the sparse data structure could offset any advantages. In summary, to optimize GMN computations on GPUs, it is essential to eliminate redundant computations and maximize parallelism during the node-matching stage. Moreover, it is equally important to carefully balance the advantages of such strategies against the potential overheads.

To this end, we introduce FlexGM, a runtime system to adaptively accelerates GMN computations on GPUs. Firstly, we present a graph deduplication module that reduces unnecessary computation in the node-matching stage. Secondly, we implement a unified matching module that parallels node

2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00060

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

matching across various graph pairs to harness the full potential of GPUs' parallel capacity. Lastly, to optimally balance the trade-off between the advantages and overheads of these enhancements, we propose an adaptive module manager that selectively activates optimization modules based on the input data, thus ensuring diverse inputs achieve their optimization targets. Our contributions can be summarized as follows:

- We conduct a thorough and quantitative analysis of the computational challenges faced by GMNs on GPUs and identify the optimization opportunities in diverse input workloads.
- We introduce FlexGM, a runtime system designed to enhance GMN execution on GPUs. FlexGM incorporates a graph deduplication module to mitigate redundancy, a unified matching module to optimize hardware utilization, and an adaptive module manager to dynamically activate beneficial optimizations.
- We implement and evaluate FlexGM utilizing widely adopted GMN models. The experimental results demonstrate that FlexGM achieves an average speedup of $2.5\times$ and peaks at $7.6\times$ over existing GMN implementations.

II. Background

A. Graph Matching Networks

Graph Neural Network (GNN) based models for graph similarity computation have recently emerged as a popular choice owing to their enhanced accuracy and scalability [3], [5]–[7]. These models are referred to as *Graph Matching Networks (GMNs)*. These GMNs typically encompass two stages: When provided with a pair of graphs (G_1, G_2), where G_1 is conventionally referred to as the target graph and G_2 as the query graph, all nodes from both graphs are subjected to two stages, namely, node embedding stage and node matching stage, these stages occur either at a layer-wise or model-wise level and help calculate the degree of similarity between the graphs, as shown in Figure 1.

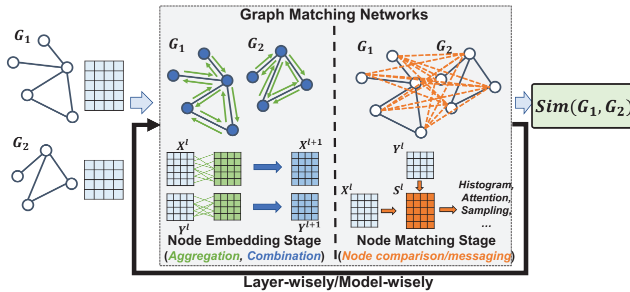

Fig. 1. Node embedding and node matching stages in GMN models.

Stage 1: Node Embedding. During the node embedding stage, GMNs employ a conventional GNN methodology to update node features. At this stage, every node gathers intra-graph messages from its adjacent nodes, subsequently merging these received messages with its own features through neural networks. The general process for the node embedding stage at layer l can be articulated as follows:

$$X^{l+1} = \sigma(COMB(AGGR(A, X^l, W_e^l), W_n^l)) \quad (1)$$

The equation incorporates $\sigma(\cdot)$ as the activation function, A as the graph's adjacency matrix, X^l as the layer l node feature, and W_e^l and W_n^l as the layer l weights. The input feature X^l is aggregated along edges, as denoted by the adjacency matrix A, in the $AGGR(\cdot)$ module. Following this, a combination module $COMB(\cdot)$ combines the aggregated messages with nodes' original features based on W_n^l.

Stage 2: Node Matching. During the node matching stage, GMNs compute the similarity between nodes from target and query graphs. Various functions like dot-product similarity, cosine similarity, and Euclidean similarity are utilized to compute similarities between the cross-graph node-pairs [5]–[7]. These similarity values can either be used directly for making predictions [5], [7] or indirectly facilitate further cross-graph communication [3], [6]. The matching procedure at layer l can be represented as:

$$S^l = \frac{X^l (Y^l)^T}{K} \quad (2)$$

The S^l is the similarity matrix between two node sets in layer l and K is the scale factors (i.e., $K = 1$ for dot-product similarity, $K = 2$ for euclidean similarity, and $K_{ij} = \|X_i^l\| \cdot \|Y_j^l\|$ for cosine similarity). For Euclidean similarity, the score S_{ij}^l will be further normalized by subtracting squared magnitudes of row vectors $S_{ij}^l = S_{ij}^l - (\|X_i^l\|^2 + \|Y_j^l\|^2)$ [6]. Recent GMNs adopt layer-wise node matching since it yields better accuracy [3], [4], [6], [7], [17].

B. GMN implementations on GPUs

GMNs are commonly executed on GPUs by leveraging existing GNN and deep learning frameworks [18]–[21]. In particular, the node embedding stages are implemented by GNN frameworks such as PyTorch-Geometric (PyG) [19], and the node matching stages are implemented by general-purpose deep learning frameworks like PyTorch [18]. Many studies focus on improving the performance of GNN operations on GPUs [8], [19], [20]. As a result, the computations in the node embedding stages are effectively parallelized across batches of input graphs, facilitating simultaneous processing of edges and nodes from various graph pairs and capitalizing on the parallel processing power of GPUs.

III. Motivations

To delve into potential challenges and opportunities associated with optimizing GMNs on the GPUs, we characterize computations in three widely used GMN models: Graph-Matching-Network (GMN-Li) [6], SimGNN [5] and Graph-Sim [7]. The models are implemented in PyTorch [18] and PyTorch-Geometric [19]. We investigate their performances on three representative datasets: AIDS [22] represents small-sized graphs, Github Stargazers (GITHUB) [23] represents middle-sized graphs, and REDDIT-BINARY (RD-B) [15] represents large-sized graphs. More details about GMN models, datasets, and experiment platforms are described in Section V.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

A. Bottleneck in GMNs

To recognize the bottleneck in GMN computing, we first analyze the latencies from different stages of the GMN model. Figure 2 illustrates the breakdown of GMN inference latencies. Noticeably, the node matching consumes over 90% total execution time on average. Hence, optimizing the node-matching stage is crucial for enhancing the efficiency of GMN execution. The substantial time consumption can be traced to two main issues from two kinds of distinct computing patterns: While dealing with large graphs, the quadratically increased node comparisons can cause heavy computation, resulting in significant latencies with massive redundancy; while handling small graphs, a single graph pair can hardly fully exploit the GPU parallel capability, leading to long latencies with low hardware utilization.

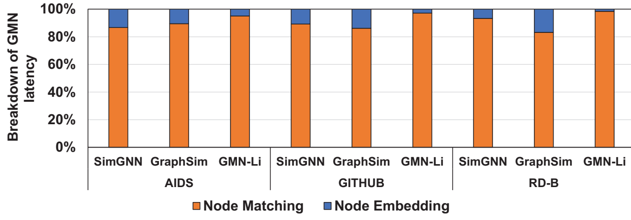

Fig. 2. Normalized latency breakdown in GMN models.

B. Redundancy in quadratically increased node comparisons

When computing node matching on large graph inputs, the GMNs face an overwhelming surge in computational demands. The volume of computations and associated memory accesses grow dramatically with the increase in graph sizes due to all-to-all node comparisons. Nevertheless, massive redundancy exists in these heavy burdens due to duplicate subgraphs.

Redundant matching. In GMNs, the feature of a node in layer l, denoted as X_i^l, represents the information from the l-hop subgraph surrounding $node_i$. As such, if another node, $node_j$, has an l-hop neighborhood forming a subgraph identical to that of $node_i$ (i.e., isomorphic neighborhood subgraphs), the features of $node_i$ and $node_j$ at layer l will be the same (i.e., $X_i^l = X_j^l$). As the example illustrated in Figure 3, $node_0$ and $node_1$ have the same features due to their identical neighborhoods. We refer to nodes with distinct features as unique nodes and nodes with features that replicate those of the unique nodes as duplicate nodes. The matching results from duplicate nodes to the other graphs are identical to their unique counterparts. For instance, suppose we carry out node matching at layer l. Assuming we have a unique $node_i$ with features X_i^l and a duplicate $node_j$ having identical features to $node_i$ (i.e., $X_i^l = X_j^l$) matched with the same graph (i.e., Y^l), the matching results for $node_i$ would be the same as the matching results of $node_j$, as shown below.

$$S_j^l = \frac{X_j^l(Y^l)^T}{K} = \frac{X_i^l(Y^l)^T}{K} = S_i^l \tag{3}$$

In this case, the matching results associated with the duplicate $node_j$ (i.e., S_j^l) can be obtained by reusing the matching results tied to the unique $node_i$ (i.e., S_i^l). We refer to the matching results from the unique nodes as unique matching and the results from the duplicate nodes as duplicate matching. Using the example illustrated in Figure 3, the matching results from $node_0$ to nodes in the other graphs S_0^1 are the same as those results computed by $node_1$ (i.e., S_1^1), then we can regard one of them as unique matching and reuse it for the other. Consequently, computations for duplicate matching are redundant and can be eliminated. We quantitatively investigate the number of duplicate nodes and corresponding redundant matching to quantify this potential redundancy. As depicted in Figure 4, duplicate nodes account for over 74% on average, leading to over 78% duplicate matching.

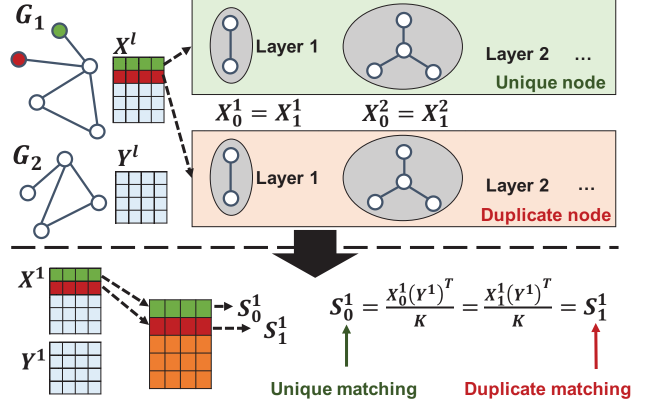

Fig. 3. An illustration of a duplicate node and its corresponding duplicate matching with their unique counterparts.

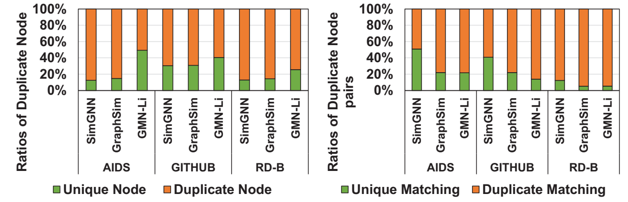

Fig. 4. Ratios of duplicate node (left) and consequent duplicate pairs (right).

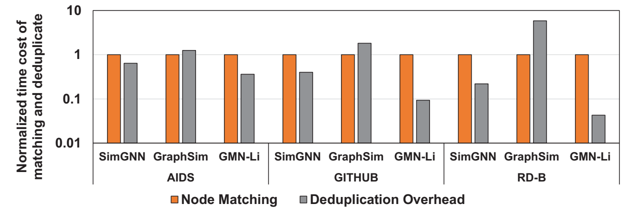

Fig. 5. Comparison between latencies of matching stage and deduplication overheads. All results are normalized to the latency of the matching stage.

Challenges and trade-off of deduplication. However, redundancy removal in GMN computing is costly. The process, which we call deduplication, requires comparing nodes within a graph and recognizing duplicates, often introducing considerable overheads due to numerous comparisons on lengthy node

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

features. To investigate potential overheads, we implement a naive solution for deduplication, which first identifies duplicate matching by comparing node features within the same graph and then computes unique matching results for all duplicate nodes. As illustrated in Figure 5, the average overhead of the deduplication process is 1.1× the latency of the matching stage with redundancies, indicating its unignorable costs. The overhead ratios fluctuate across models and datasets. For instance, in GMN-Li, the deduplication accounts average 0.17× of the original matching stage latencies. However, in GraphSim, the cost rises to 2.9×. Moreover, smaller graphs amplify the ratio because node matching is relatively faster. For instance, in RD-B, deduplication for GMN-Li takes only 0.04× latencies compared to node matching, yet in AIDS, the ratio increases 0.36×. This variability underscores the need for an adaptive approach that can flexibly handle redundant computation based on the specific context of the GMN model and dataset.

C. Poor parallelism across graph pairs

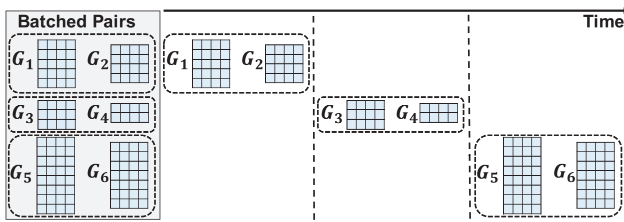

Fig. 6. An illustration of serial node matching in baselines.

When carrying out node matching on small graph inputs, the GMNs experience suboptimal hardware utilization. The amount of concurrent computations and related memory accesses is insufficient to fully engage the underlying resources, resulting in inefficient hardware utilization. However, it is challenging to parallel node matching across different graph pairs due to varying sizes of input graphs, which leads to different input sizes for Equation (2) and subsequent operations. To this end, when handling the node-matching stage of batched graph pairs, existing implementations of GMNs compute graph pairs in a pair-by-pair manner (i.e., serialized), and concurrent computations are only performed on node pairs from the same graph pairs, as shown in Figure 6. While nodes within large graph pairs can easily maximize the usage of the GPU's computing units and memory bandwidth, small graphs may underutilize these resources. To further explore this issue, we evaluated the utilization of Stream Multiprocessors and memory bandwidth during the node-matching stages in GMNs. As seen in Figure 7, GPU resources tend to be underutilized when handling small graphs in AIDS.

Challenges and trade-off of optimization. A viable method for parallelizing node comparisons across batched graph pairs is to employ sparse matrix multiplication (SpMM), which allows parallel comparison of node pairs within a batch using auxiliary data structures for specifying required comparisons. However, this introduces overheads, primarily

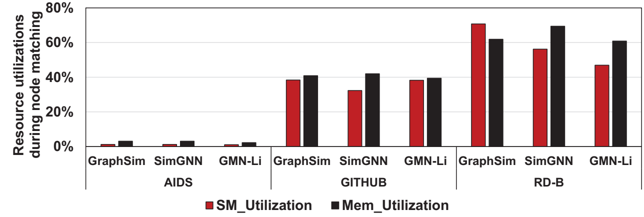

Fig. 7. Stream Multiprocessor (SM) and Memory (Mem) utilization of GMN models during the node matching.

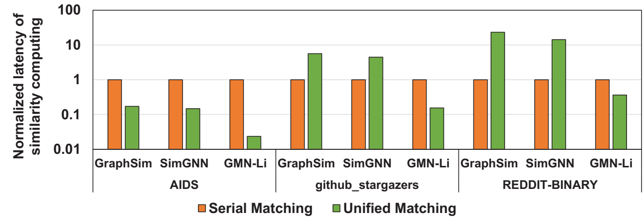

Fig. 8. Latencies of node pair similarity computing in baselines (serial matching) and unified matching normalized to the baselines.

due to the need for an additional data structure to indicate the node pairs requiring computation. If the graph pairs are already sufficiently large to maximize GPU resources, this method might yield little benefit and may even degrade performance. We refer to this approach as "unified matching" and discuss the design details in Section IV-C. As depicted in Figure 8, unified matching significantly accelerates the computation of node pair similarity in AIDS, delivering an 18× speedup on computing Equation (2). However, the solution's effectiveness diminishes in benchmarks with larger graph inputs, such as RD-B. Therefore, it is critical to devise a strategy that adaptively parallel computations across different pairs to optimally utilize the underlying GPU hardware resources.

IV. FlexGM

A. Overview of FlexGM

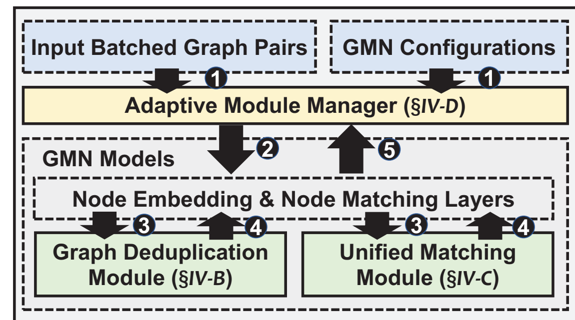

Fig. 9. The workflow overview of FlexGM.

To tackle the challenges in GMN computing, we introduce FlexGM, an end-to-end runtime system aimed at boosting the performance of GMNs on GPUs. FlexGM is composed of three key components: Firstly, the Graph Deduplication

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

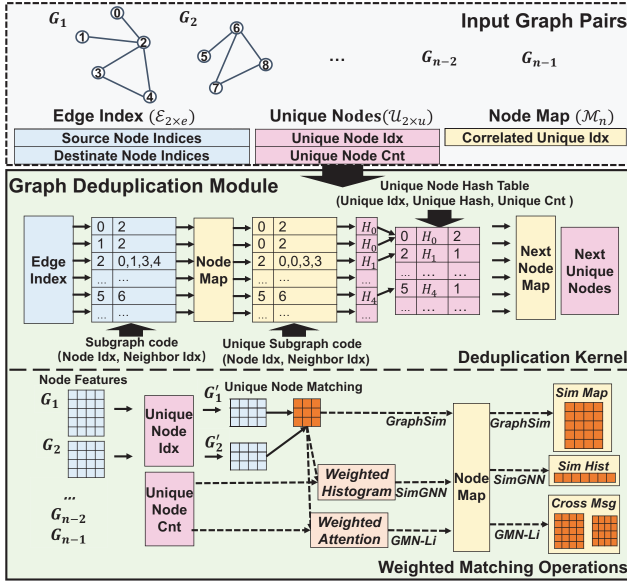

Fig. 10. The illustrated workflow of the Graph Deduplication Module.

Module (§IV-B) eliminates redundant computations in GMNs. Secondly, the Unified Matching Module (§IV-C) parallels node similarity computing across varying-sized graph pairs. Thirdly, the Adaptive Module Manager (§IV-D) dynamically activates the modules above to balance the trade-off between their overheads and benefits. As illustrated in Figure 9, the workflow in FlexGM consists of the following steps: Initially, the adaptive module manager gathers input (e.g., graph sizes) and model information (e.g., node feature size)(❶), determines which modules to activate, and configures the GMN layers(❷). During GMN computing, the layers dispatch the workload to the activated modules for optimized execution(❸). The activated modules then perform the execution and return results that align with the native GMN models' format(❹). Periodically, the adaptive module manager gathers runtime information (e.g., layer latencies) from GMN models(❺), adjusting its decisions and parameters.

B. Graph Deduplication Module

The graph deduplication module eliminates redundant computations with two functionalities: Firstly, it incorporates a deduplication kernel that efficiently identifies unique nodes and maps duplicate nodes to their unique equivalents. Secondly, it adopts a set of weighted matching operations to execute node-matching schemes within the GMN models without referring to the duplicate nodes. The detailed workflow of the graph deduplication module is depicted in Figure 10.

Deduplication Kernel. The deduplication kernel adopts a customized CUDA kernel that harnesses the computing power of GPUs to efficiently identify duplicate nodes in parallel. The kernel leverages a unique neighbor encoding scheme to simplify the equality check. Moreover, it adopts a hash-table-based approach to avoid massive comparisons between nodes. In addition to the edge index that is commonly used by GNN libraries to represent the graph topology [19], [20], the module takes two extra tensors as inputs: The *Unique Nodes* tensor, denoted as $\mathcal{U}_{2\times u}$, serves to represent unique node information. Its ith entry $\mathcal{U}[i]$ consists of the node index of the ith unique node and the number of duplicates of the ith unique node (including itself). The *Node Map* tensor $\mathcal{M}_n$ is a one-dimensional tensor that stores mappings from each node to its unique node. Its ith element $\mathcal{M}[i]$ denotes that $node_i$ is a duplicate of $node_{\mathcal{M}[i]}$. During the deduplication process, nodes are dispatched to different threads and processed parallelly as follows:

Firstly, each node is represented by an integer tensor denoted as *Subgraph Code*. The first element of this tensor is the node's index, while the subsequent elements are the indices of the node's neighbors. For example, as depicted in Figure 10, $node_1$ has one neighbor $node_2$. Hence, its subgraph code is a two-element tensor $(1, 2)$. Since real-world graphs typically have nodes with a limited number of neighbors, using subgraph codes instead of node features can considerably reduce overheads associated with the following steps.

Secondly, the nodes' subgraph codes are transformed into their unique representation, denoted as the *Unique Subgraph Code* in Figure 10. Specifically, each thread replaces the indices within the subgraph code with their respective unique counterparts from the node map and sorts these unique indices. The sorting ensures that neighbors in different permutations are represented equivalently. Referring to the previous example, the unique equivalent for $node_1$ is $node_0$, while $node_2$ is a unique node. Hence, the unique subgraph code of $node_1$ is transformed from $(1, 2)$ to $(0, 2)$.

Next, each thread hashes the *Unique Subgraph Code* of the node and uses the resulting hash value to locate an entry in the *Unique Node Hash Table*. If the entry is unoccupied, the node is unique. In such a scenario, the thread records the node's index, hash value, and the pointer to its unique subgraph code in the entry. However, if the entry is occupied and the unique subgraph code of the entry matches the node's code, it is a duplicate node. In this case, the thread increments the unique count in this entry by one and assigns its unique node index to the node's unique index in the node map. Using the previous example, the thread hashes the unique subgraph code of $node_1$ into H_0. Upon checking, the thread finds that the entry associated with H_0 is already occupied by $node_0$, and they have the same unique subgraph code. Therefore, $node_1$ is identified as a duplicate node of $node_0$. The thread then assigns the unique index of 0 to $node_1$ (i.e., $\mathcal{M}[1]$) and increments the unique count in that entry (i.e., $\mathcal{U}[0][1]$) by one.

Weighted Matching Operations. To realize the node-matching stage without relying on duplicate nodes, we use a set of weighted matching operations to modify GMN layers. The weighted matching operations accept the *unique nodes* ($\mathcal{U}_{2\times u}$) alongside the node features as inputs and compute node-matching stages in the following steps: Firstly, we employ the unique node index to select and match unique nodes. As depicted in Figure 10, we select the unique nodes from the graph pair (G_1, G_2), yielding graph pairs $(G_1^{'}, G_2^{'})$ that

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

consist exclusively of unique nodes, and conduct matching between them. Secondly, we utilize the unique node count to weigh the subsequent matching mechanisms in different GMN models. Intuitively, the matching results from the unique node are scaled up (i.e., multiplied) by these counts, then participate in the following computing as results from a set of duplicate nodes. For instance, for SimGNN, which necessitates a histogram operation to count similarities within different ranges [5], the unique node counts are multiplied by the count, thereby ensuring that the original counts on duplicate nodes are correctly computed by counting the weighted counts on unique nodes. For GMN-Li, which requires an attention operation to weigh and aggregate cross-graph information [6], we multiply the attention value by the unique node counts, ensuring that the messages from multiple duplicate nodes are substituted by scaled messages from unique nodes. Lastly, we recover the matching results for each node using the *node map* (i.e., $\mathcal{M}_n$).

C. Unified Matching Module

The unified matching module dissects the node-matching process from different graph pairs into fine-grained node pairs and computes the similarities between these node pairs in parallel through a customized CUDA kernel. We illustrate the workflow of the Unified Matching Module in Figure 11. First, the graph pairs are decomposed into node pairs, and the matching tasks from different graph pairs are unified into a batch-wise matching procedure. We employ an auxiliary 2-D tensor called *Match Index* to indicate which node pair should be compared, whose each entry (i, j) signifies the computation of similarity S_{ij} between $node_i$ and $node_j$. Second, all node pairs are partitioned into equally-sized *Matching Groups*. Each group is assigned a GPU warp to ensure a balanced workload distribution. We leverage the shared memory to cache the match indices in each block, as the threads frequently reuse them to access the node features. Lastly, the threads within a warp employ the match index (i, j) to access the features of $node_i$ (i.e., X_i) and $node_j$ (i.e., X_j). In particular, each thread is assigned a specific feature from the node pairs, and consecutive threads access and process consecutive node features of the pair. This approach allows the memory access of node features within a warp to be aligned into fewer memory transaction requests. The partial results produced by the different threads are aggregated using the $__shfl_down_sync()$ operation, which directly communicates between threads and accumulates the partial results into the final similarity score.

D. Adaptive Module Manager

The adaptive module manager dynamically activates beneficial modules based on cost models and uses runtime feedback to validate these decisions. For each batch, the module manager collects input information(e.g., graph sizes, node feature sizes, etc.), estimates the potential benefits of the modules based on cost models, and enables beneficial modules before inference. For the graph deduplication module, the cost model is as Equation (4),

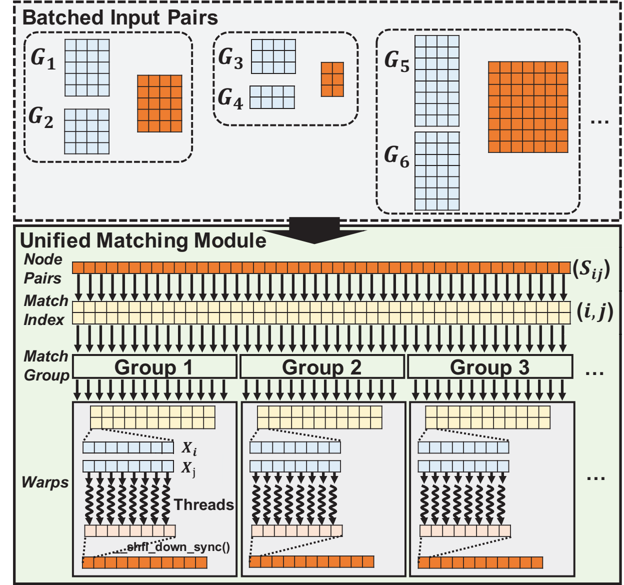

Fig. 11. The detailed workflow of the Unified Matching Module.

$$NB_{dup} = \frac{|M| \times \gamma \times \alpha}{|V|}, \mathcal{I}_{dup} = \begin{cases} 1, & \text{if } NB_{dup} \geq \theta_{dup} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

$|M|$ denotes the number of matching pairs per batch, γ and α represent matching complexity and duplicate matching ratio, respectively. We define *matching complexity* as the approximate ratio of matching stage latency to the overall latencies. Removing duplicates can be more beneficial with higher matching numbers, complexity, and duplicate rates, thus we multiply them to get the benefit factors. We use the number of nodes per batch, $|V|$, to approximate overheads and divide the benefit factors by this approximation to obtain NB_{dup}, representing the potential benefit matrix for enabling the graph deduplication module. If the matrix is higher than the preset threshold θ_{dup}, the graph deduplication module will be activated. For the unified matching module, the cost model can be represented as Equation (5),

$$NB_{um} = \frac{P \times B}{|M||F|}, \mathcal{I}_{um} = \begin{cases} 1, & \text{if } NB_{um} \geq \theta_{um} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

We approximate the intensity of node matching computing per graph pair by multiplying their average number of node pairs $\frac{|M|}{B}$ (i.e., total node pairs divided by batch size) with the size of node features $|F|$. The results are further discounted by dividing the GPU hardware's theoretical FLOPS, denoted as P, then reversed to obtain the final metric NB_{um}. Intuitively, it represents how many graphs pairs the underlying GPUs can handle concurrently. If NB_{um} is higher than the preset threshold θ_{um}, the module will be enabled.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

TABLE I
DETAILS OF GMNS MODELS.

Model	Layers (Type[hidden_sizes])	Similarity
GMN-Li [6]	MLP[1,64], 5*{MGNN[64,64,64], MATCHING[64,64], MLP(64*3,64,64)} READOUT[64,128,128]	Euclidean
GraphSim [7]	3*(GCN[1,64], SIM[64,1],SAMPLE[M,16×16]), CNN[3,16,32,64,128] MLP([128*3,128,64,32,16,1])	Cosine
SimGNN [5]	3*(GCN[1,64]), SIM[64,1] + HIST[M,16], READOUT[64,128,16], NTN[128,16],MLP([32,16,8,4,1])	Dot-product

TABLE II
DETAILS OF DATASETS.

Datasets	Ave. # of Nodes	Ave. # of Edges	# of Graph Pairs
AIDS [22]	15.69	16.20	200
GITHUB [23]	113.79	234.64	1273
RD-B [15]	429.63	497.75	200
RD-5K [15]	508.52	594.87	500
RD-12K [15]	391.41	456.89	1193

During GMN computing, the manager periodically validates its decisions and adjusts its parameters. Specifically, it periodically activates disabled modules and monitors their latencies to check if there is a better choice. If the current mode is incorrect, the manager will use the current matrix value to overwrite the current threshold so that the manager will make a different choice on similar workloads next time. The manager periodically samples runtime information and uses running averages to update parameters, such as γ and α in Equation (4), which are not directly detectable from input data. Initially, these values are set based on offline profiling averages, then updated with running averages of runtime latencies and duplicate matching ratios sampled during runtime.

V. EVALUATION

A. Experiment setup

Models. We evaluate FlexGM using three recent GMNs: 1) GMN-Li [6] and 2) GraphSim [7] conducts layer-wise node matching, and 3) SimGNN [5] employs node matching in the last-layer. The model details are described in Table I. The layer configurations are presented as Operations([hidden sizes]). The special GNN used in [6] is referred to as MGNN.

Datasets. We employ five widely-used real-world graph classification datasets for our evaluation, as detailed in Table II. Specifically, AIDS [22] comprises small-sized graphs representing molecular compounds sourced from the Antiviral Screen Database of Active Compounds. Github Stargazers(GITHUB) [23] encompasses medium-sized graphs, where nodes symbolize authors and edges indicate their relationships. REDDIT-BINARY(RD-B), REDDIT-MULTI-5K(RD-5K), and REDDIT-MULTI-12K(RD-12K) [15] consist of large-sized graphs, where nodes represent users, and edges denote relationships between them. Following classification task settings in GMN-Li [6], we create similar/dissimilar graph pairs by randomly substituting edges in the base graphs. We set the batch size to 32 for the datasets with smaller graphs (i.e., AIDS, GITHUB) and 8 for those larger graphs (i.e., RD-B, RD-5K, and RD-12K) to avoid out-of-memory.

Platforms and implementations. We conduct experiments on a server with a 64-Core AMD EPYC 7742 CPU and an NVIDIA A100 GPU. The following GMN implementations are compared in the experiments:

- **PyG (Baseline).** The PyG implements GMN models with PyTorch [18] and PyTorch-Geometric [19]. Specifically, we implement the node embedding stages within GMNs by the PyTorch-Geometric and use PyTorch to realize the operations in the node matching stages.
- **FlexGM.** FlexGM uses C++/CUDA for the backend and Python for the front end. The node embedding stage remains the baseline implementation, and the node matching stages are enhanced by our designs.

To investigate the benefits of each module, we also compare the following modes of FlexGM. The adaptive module manager is disabled in these modes.

- **FlexGM_Dedup.** This version of FlexGM always activates the graph deduplication module while keeping the unified matching module down.
- **FlexGM_UM.** This version of FlexGM always activates the unified matching module while keeping the graph deduplication module down.
- **FlexGM_Full.** This version of FlexGM always activates both the unified matching module and the graph deduplication module.

B. Overall Speedup

As shown in Figure 12, FlexGM achieves 2.6× speedups over the baseline on average. In benchmarks with heavy node-matching computing on large graphs, such as GMN-Li on RD-5K, the speedup can be up to 7.6×. Regarding GMN models, the average speedup on GMN-Li, GraphSim, and SimGNN are 4.7× 1.8× and 1.3×, respectively. There are more significant speedups on GMN-Li, which conducts heavier node-matching computing. Specifically, GMN-Li computes similarities and uses them for cross-graph message passing in each of its five layers, leading to higher node-matching latency ratios that are effectively shortened. Regarding the datasets, FlexGM achieves higher speedup on datasets containing larger graphs, such as RD-B, RD-5K, and RD-12K. The primary reason is massive duplicate subgraphs in these larger graphs. This phenomenon leads to considerable benefits from the deduplication module, resulting in a 3.5× average speedup in RD-B, RD-5K, and RD-12K. While the deduplication is not beneficial in the smaller graphs, unified matching can also effectively speed up the GMNs, resulting in a 1.3× speedup in AIDS. The results demonstrate that FlexGM can effectively speed up different GMNs on diverse workloads.

C. Optimization Analysis

Graph Deduplication Module. As shown in Figure 12, the graph deduplication module (i.e., FlexGM_Dedup) is highly effective when dealing with larger graphs. Specifically, 3.1×, 3.3×, and 2.5× speedups are observed in the RD-B, RD-5K, and RD-12K datasets. These improvements can be attributed

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

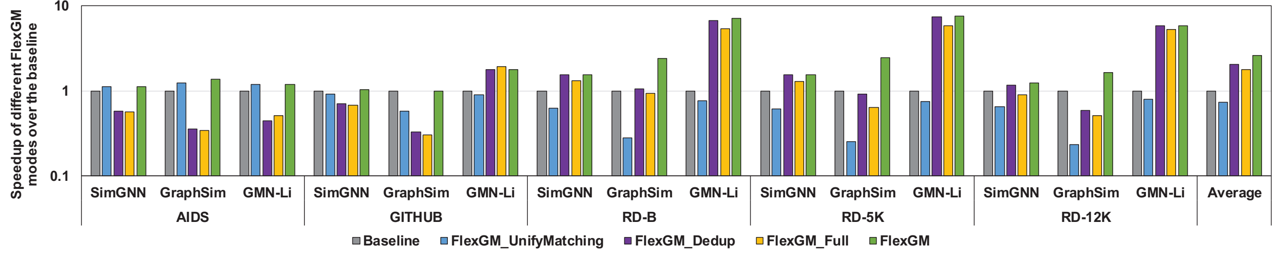

Fig. 12. The speedup of FlexGM_UM, FlexGM_Dedup, FlexGM_Full and FlexGM over the baseline.

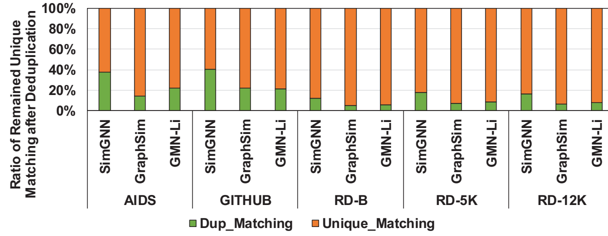

Fig. 13. The ratio of detected unique and duplicate matching.

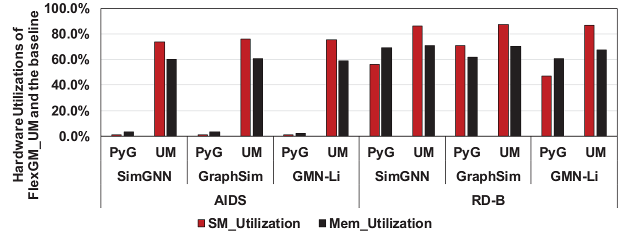

Fig. 14. The hardware utilizations of FlexGM_UM and baseline.

to higher ratios of duplicates. As depicted in Figure 13, the deduplication module eliminates over 90% matching in the RD-B, RD-5K, and RD-12K dataset, demonstrating that larger real-world graphs are more likely to contain recurring subgraphs. The graph deduplication module, therefore, plays a critical role in improving GMN performances on large graphs.

Unified Matching Module. The FlexGM_UM efficiently enhances the performance of GMNs on small graphs. As shown in Figure 12, in AIDS, it achieves 1.4×, 1.6×, and 1.2× speedups over the baseline for SimGNN, GraphSim, and GMN-Li, respectively. As shown in Figure 14, in AIDs, the underlying hardware resources are better exploited than the baseline. However, in RD-B, where the module is not as beneficial, the hardware resources are already exploited well.

Adaptive Module Manager. It can be observed from Figure 12 that simply enabling both modules statically in FlexGM_Full does not consistently yield performance improvements; in certain instances, it may even result in increased latencies. However, incorporating the adaptive module manager, FlexGM surpasses the performance of all statically configured modes. Notably, in datasets with smaller graphs, such as AIDS, FlexGM achieves comparable performance enhancements as FlexGM_UM, while in datasets containing larger graphs, like RD-B, it achieves the performance gains observed in FlexGM_Dedup. These findings underscore the ability of the adaptive module manager to judiciously select and enable modules that contribute positively to performance.

VI. Related works

Various software and GPU runtime optimizations have been introduced to enhance the performance of GNNs and traditional Graph Matching algorithms [9]–[11], [24], [25]. GNNAdvisor [8] utilizes a group-based workload management system to balance workload and a node reordering scheme to optimize data locality. QGTC [10] constructs a tensor-core-based framework supporting arbitrary bit-width quantized GNNs. HAG [9] proposes a hierarchical aggregation method to eliminate redundant computations during node aggregation. However, these techniques primarily enhance the node-embedding stage in GMNs, overlooking inefficiencies in the node-matching stage. Conversely, optimizations in conventional subgraph matching computation, like CECI [24], aren't readily transferable to GMNs. This is attributed to the differences in their computing flows: traditional algorithms focus on search-based matching, while GMNs are built on top of the GNN-based parallel similarity computations. CEGMA [14] is the most relevant to our study. It improves GMN computing by memorizing matching results, integrating node embedding and matching stages in GMNs, and designing a dedicated hardware accelerator. However, CEGMA relies on extra hardware designs to mitigate overheads and maximize the efficiency of the suggested dataflows, which hampers their application on general computing platforms. To conclude, while substantial efforts have been made to accelerate GNNs, no prior work targets optimizing GMN computing on GPUs, leaving a gap between general computing platforms and emerging GMN algorithms.

VII. Conclusion

In this study, we introduce FlexGM, a runtime system, to tackle the computational challenges inherent in Graph Matching Networks (GMNs) on GPUs. FlexGM speeds up GMN execution by adaptively eliminating redundant computation and exploiting underlying GPU resources. The experiment results show that FlexGM can achieve 2.5× on average and up to 7.6× speedup over the prior implementations.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGEMENT

The authors sincerely thank the anonymous ICCD reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, #2154973, #1725657, #1910413, and #2312157.

REFERENCES

[1] K. M. Borgwardt, H.-P. Kriegel, S. Vishwanathan, and N. N. Schraudolph, "Graph kernels for disease outcome prediction from protein-protein interaction networks," in *Biocomputing 2007*. World Scientific, 2007, pp. 4–15.

[2] C. Fabiana, M. Garetto, and E. Leonardi, "De-anonymizing scale-free social networks by percolation graph matching," in *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015, pp. 1571–1579.

[3] P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich, "Superglue: Learning feature matching with graph neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 4938–4947.

[4] W. Li, X. Liu, and Y. Yuan, "Sigma: Semantic-complete graph matching for domain adaptive object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 5291–5300.

[5] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, 2019, pp. 384–392.

[6] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[7] Y. Bai, H. Ding, K. Gu, Y. Sun, and W. Wang, "Learning-based efficient graph similarity computation via multi-scale convolutional set matching," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 3219–3226.

[8] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus," in *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 2021.

[9] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation for graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 997–1005.

[10] Y. Wang, B. Feng, and Y. Ding, "Qgtc: accelerating quantized graph neural networks via gpu tensor core," in *Proceedings of the 27th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2022, pp. 107–119.

[11] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 401–415.

[12] Y. Jin, D. Mishkin, A. Mishchuk, J. Matas, P. Fua, K. M. Yi, and E. Trulls, "Image matching across wide baselines: From paper to practice," *International Journal of Computer Vision*, vol. 129, no. 2, pp. 517–547, 2021.

[13] J. Ma, X. Jiang, A. Fan, J. Jiang, and J. Yan, "Image matching from handcrafted to deep features: A survey," *International Journal of Computer Vision*, vol. 129, no. 1, pp. 23–79, 2021.

[14] Y. Dai, Y. Zhang, and X. Tang, "Cegma: Coordinated elastic graph matching acceleration for graph matching networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 584–597.

[15] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, 2015, pp. 1365–1374.

[16] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[17] I. Papakis, A. Sarkar, and A. Karpatne, "Gcnnmatch: Graph convolutional neural networks for multi-object tracking via sinkhorn normalization," *arXiv preprint arXiv:2010.00067*, 2020.

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

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

[20] R. Wang, J. Yan, and X. Yang, "Learning combinatorial embedding networks for deep graph matching," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 3056–3065.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.

[22] K. Riesen and H. Bunke, "Iam graph database repository for graph based pattern recognition and machine learning," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 2008, pp. 287–297.

[23] B. Rozemberczki, O. Kiss, and R. Sarkar, "An api oriented open-source python framework for unsupervised learning on graphs," *arXiv preprint arXiv:2003.04819*, vol. 10, no. 3340531.3412757, 2020.

[24] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1447–1462.

[25] B. Li, J. Yin, A. Holey, Y. Zhang, J. Yang, and X. Tang, "Trans-fw: Short circuiting page table walk in multi-gpu systems via remote forwarding," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 456–470.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:41:48 UTC from IEEE Xplore. Restrictions apply.