CEGMA: Coordinated Elastic Graph Matching Acceleration for Graph Matching Networks

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

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

Xulong Tang
Computer Science Department
University of Pittsburgh
Pittsburgh, PA
Email: tax6@pitt.edu

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070956

Abstract—**The recently proposed Graph Matching Network models (GMNs) effectively improve the inference accuracy of graph similarity analysis tasks. GMNs often take graph pairs as input, embed nodes features, and match nodes between graphs for similarity analysis. While GMNs deliver high inference accuracy, the all-to-all node matching stage in GMNs introduces quadratic computing complexity with excessive memory accesses, resulting in significant computing and memory overhead that cannot be handled by existing approaches. In this paper, we propose the Coordinated Elastic Graph Matching Accelerator (CEGMA), a software and hardware co-design accelerator to address the challenges of GMNs. Specifically, by exploiting duplicate subgraphs in the input graphs, we develop an elastic matching filter to significantly reduce the quadratic computing overhead. By exploring the substantial data reuses oriented from accessing node features, we propose a cross-graph coordinator that fuses cross-graph similarity computing with intra-graph computing to enhance data locality. Experimental results show that, on average, CEGMA achieves $353\times$ and $6.5\times$ speedups in GMN computing compared to state-of-the-art GPU implementation and GNN accelerators, respectively.**

Index Terms—**Graph Matching Networks, Graph Neural Networks, Accelerator**

I. Introduction

Graph similarity analysis (i.e., graph matching) is an important application that powers many application fields, including protein interaction analysis for disease prediction in medical science [6], friend cycle matching in social networks [10], [19], [30], program debugging and subroutine identification in compilers [2], [3], [7], [36], and binary function similarity measuring in computer security [24]. In particular, feature matching gains momentum in the computer vision domain, which puts strict requirements on matching accuracy and matching latency [11], [12], [23], [31], [35], [39]. Graph similarity analysis is known to be an NP-hard problem [26] and it is increasingly difficult to conduct efficient graph matching analysis when the graph size is large.

Recently, Graph Matching Network (GMN) models have been proposed to enhance the efficiency of similarity analysis [4], [5], [24]. Specifically, the inference of these models consists of two stages: (1) **node embedding** which employs Graph Neural Networks (GNNs) to embed intra-graph node feature information into node vectors; and (2) **node matching** which compares/matches the embedded node vectors from one graph to all nodes in the other graph for similarity analysis. While the node embedding stage can benefit from substantial prior optimizations on Graph Neural Networks (GNNs) [17], [21], [38], [41], the node matching stage dominates the GMN execution performance and involves significant memory accesses and computation that have been overlooked. Moreover, many applications relying on graph similarity analysis have strict requirements on fast and efficient GMNs. One particular example is the graph matching tasks in computer vision that require real-time graph matching and have strict deadlines [16], [18], [27], [28]. However, using GMNs to match large graphs from images potentially yields a longer response time than the real-time demands. In addition, searching a graph in large chemistry/biology/cybersecurity databases requires millions of matching queries [33], [40], [43]. While one pair might take milliseconds on GPUs, the searching process can take hours. In contrast, real-time code clone search applications require searching within a second [40]. None of the existing CPU and GPU GMN implementations and state-of-the-art GNN accelerators can meet the latency deadlines with massive matching requests. As such, there is a practical need to design a hardware accelerator for efficient GMN computing.

It is non-trivial to design a GMN accelerator. First, the computing overhead caused by all-to-all node comparison during the node matching stage incurs dense computation and excessive memory accesses. For example, matching two graphs with 100 nodes and 1,000 edges requires 10,000 matching between cross-graph node pairs, which introduces more than $10\times$ computation and $2\times$ memory accesses than the intra-graph edge processing. Numerous redundancy exists within these cross-graph node pairs since nodes within the same graph exhibit significantly redundant matching due to their duplicate neighborhoods. Second, prior GNN accelerators focus on single graph optimizations [8], [13], [14], [20], [22], [25], [42], and fail to exploit the tremendous data reuse across graphs. The nodes from one graph are substantially reused to compute similarities with nodes from the other graph. Such reuses typically have long reuse distances and cannot be captured by existing accelerators.

In this paper, we target to reduce the i) redundant computation and ii) redundant memory accesses in GMNs. First, we observe a large amount of redundant computations and memory accesses due to the duplicate node features. Specifically, in the node matching stage, if two nodes from the same graph have the same features, their matching results are identical since they are compared with the same set of nodes in the other graph. We refer to these nodes as *duplicate nodes*. Second, we observe redundant memory accesses due to unnecessary accesses to node features in the node matching stage. Our quantitative study (discussed later in Section III) shows that the features are reused substantially between the embedding stage and the matching stage. However, those feature reuses are not translated to data locality due to i) the large reuse distances of the node features and ii) the limited on-chip storage capacity. To this end, we propose CEGMA (Coordinated Elastic Graph Matching Accelerator), which reduces the redundant computation and memory accesses to enhance the GMN efficiency, thereby satisfying the aforementioned application needs. We summarize the major contributions as follows.

- We comprehensively and quantitatively investigate the challenges in GMN models. We observe substantial redundant computation and memory accesses during the node matching stage, which have been overlooked by existing GMN implementations.
- We propose a GMN accelerator that significantly reduces the redundancy in GMN. Specifically, we propose i) elastic matching filter (EMF) that memorizes and reuses unique matching results, thereby eliminating the redundant computation, and ii) cross-graph coordination (CGC) that coordinates the embedding and matching stages at a fine granularity, thus enhancing the node feature locality.
- We evaluated CEGMA using SimGNN, GraphSim, and GMN-Li. The results indicate that CEGMA outperforms the optimized GPU implementations by an average of $353\times$. It also outperforms the state-of-the-art GNN accelerators, HyGCN and AWB-GCN, by an average of $9.2\times$ and $6.5\times$, respectively.

II. Background

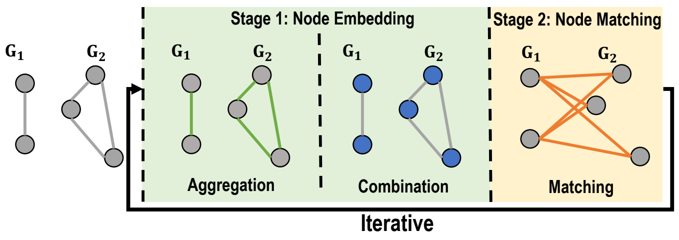


Fig. 1. Two stages in Graph Matching Networks.

In recent years, Graph Neural Network (GNN) based graph similarity computing models gain popularity due to their improved accuracy and scalability [4], [5], [24], [26], [35]. These models are referred to as *Graph Matching Networks (GMNs)*. GMNs often consist of two stages depicted in Figure 1: Given a pair of graphs (G_1, G_2)[1], all nodes from both graphs go through **node embedding** and **node matching** stages, either layer-wisely or model-wisely, to compute similarity. Specifically, in the node embedding stage, each node aggregates information from its neighbors during the aggregation phase (green edges in Figure 1) and combines the aggregated information messages with its own features during the combination phase (illustrated by blue nodes in Figure 1).[2] The matching stage performs all-to-all similarity computation between two graphs (orange edges in Figure 1).

Stage 1: Node Embedding. The GMNs use a traditional GNN to update the features of the nodes. Each node aggregates intra-graph messages from its immediate neighbors and combines the accumulated messages with its own features using neural networks. The embedding procedure at layer l can be generalized as:

$$X^{l+1} = \sigma(COMBINE(AGGREGATE(A, X^l, W_e^l), W_n^l)) \tag{1}$$

where $\sigma(\cdot)$ is the activation function, A is the adjacency matrix of the graph, X^l is the node feature at layer l, and W_e^l and W_n^l are the weights of layer l. The input feature X^l is first aggregated along edges indicated by adjacency matrix A in the $AGGREGATE(\cdot)$ module, then updated in the $COMBINE(\cdot)$ module. Since GNNs usually have several layers, nodes iteratively aggregate and combine information from their 1-hop neighbors per layer and embed K_hop subgraph information in the Kth layer accordingly [17], [21], [38], [41].

Stage 2: Node Matching. The GMNs conduct similarity computing between two nodes from target and query graphs. Different similarity calculation functions, including dot-product similarity, cosine similarity, and euclidean similarity, can be used to compute similarities between the cross-graph node-pairs [4], [5], [24]. The similarities are either directly used for prediction [4], [5] or indirectly used for further cross-graph communication [24], [35]. The matching procedure on layer l can be generalized as:

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

where S^l is the similarity matrix between two node sets in layer l (i.e., S_{ij}^l describes similarity between node i in X^l and node j in Y^l; X^l and Y^l are node features of layer l from G_1 and G_2), 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 $S_{ij}^l = S_{ij}^l - (\|X_i^l\|^2 + \|Y_j^l\|^2)$ [24]. Consequently, the node features X^l used by the *intra-graph* node embedding are also leveraged to the *cross-graph* node matching computation. Recent GMNs adopt layer-wise node matching since it yields better accuracy [5], [11], [12], [23], [24], [31], [35]. In layer-wise node matching, both equation (1) and equation (2) are calculated in each layer of the GMN

[1]We name G_1 as target graph, G_2 as query graph as convention.

[2]The combination phase can be conducted after matching stage in some GMN implementations [24].

computing during layer-wise as discussed above. In contrast, model-wise GMN only applies equation (2) at the last layer of GMN computing [4], which suffers from low matching accuracy as stated by prior works [5], [24].

III. Motivation

A. Why GMN Accelerator?

GMNs are important in many application domains as we mentioned in Section I [4], [5], [24], [26], [35], [43]. These applications require not only accurate but also fast and efficient similarity computing. However, GMNs suffer low computing efficiency due to quadratic increased computing complexity, which hardly meets the application requirements. We quantify the inference latency of GMN-Li [24] with different-sized input graphs. For each graph size, we generate random graphs following [24] and measure the average execution time per graph pair on Nvidia V100 GPU and state-of-the-art GNN accelerator (AWB-GCN [13]), as shown in Figure 2. With 1,000 nodes input graphs, the GMN-Li execution times are 33ms and 24ms on V100 and AWB-GCN per pair, and the execution times increase to 671ms and 514ms when the graph size grows to 5,000 nodes. These execution times are unacceptable for time-critical applications. For instance, autonomous vehicles ideally require $\sim$20ms for graph matching related tasks [15]. Moreover, searching a graph from an extensive database would require millions of matching queries [33], [40], [43]. For instance, searching a code clone within the BigCloneBench dataset [37] requires matching a code snippet with 60,000 candidates. While one pair might take milliseconds, the searching process can take hours. In contrast, real-time code clone search applications require searching within a second [40]. In a nutshell, there is a practical need for a GMN accelerator that meets both the accuracy and the latency requirements of graph matching applications.

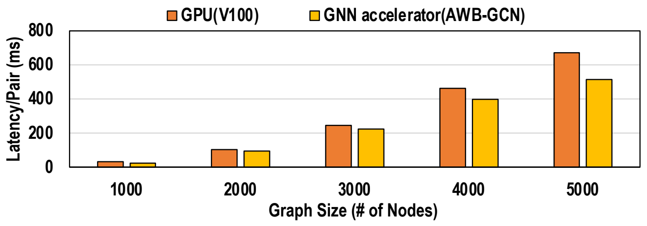


Fig. 2. Latency per pair with different sized graphs in Nvidia V100 GPU and AWB-GCN.

B. Quadratically Increased Node Comparisons

Given a pair of input graphs (G_1, G_2), finding their graph similarity demands all nodes in G_1 to be matched to the nodes in G_2, and vice versa, resulting in $\mathcal{O}(|V_1||V_2|)$ computing complexity, where $|V_1|$, $|V_2|$ are numbers of nodes in G_1 and G_2, respectively. As a comparison, the computing complexity of the aggregation stage depends on the total intra-graph edges, the complexity is $\mathcal{O}(|V_1|+|V_2|)$, much smaller than matching $(\mathcal{O}(|V_1||V_2|))$. To summarize, the node matching stage suffers from quadratic complexity growth to the input graph sizes. The *all-to-all* comparison nature leads to fast-growing computing overheads in graph similarity computing.

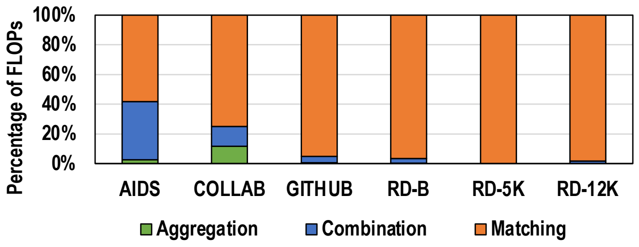


Fig. 3. Percentage of FLOPs within one GMN layer.

We quantify the percentages of FLOPs for intra-graph aggregation, combination, and cross-graph node matching stages on several commonly used datasets for graph matching tasks [26] in Figure 3 (details of the datasets are given in Table II Section V). We use the GMN layer defined in GraphSim [5], in which the node embedding follows a standard GCN setup, and the node matching computes the dot-product similarity between node features. We set the input and output feature sizes of the layer as 64. From the figure, the proportions of the FLOPs in cross-graph node matching are significantly larger than the FLOPs in intra-graph aggregation and combination (accounting for 58% to 99% of the FLOPs).

Therefore, it would not be surprising to observe larger computing overhead and more off-chip data accesses than the native GNN models.

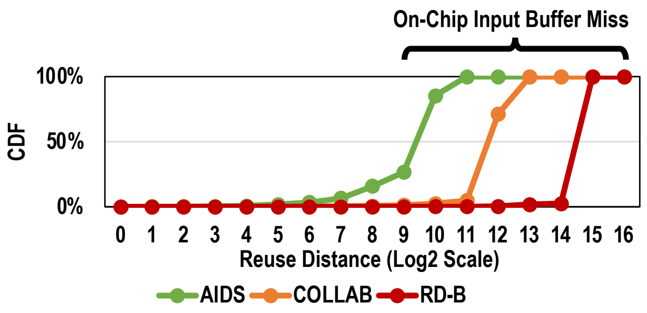


Fig. 4. Node reuse distances in GraphSim.

We next study the off-chip data accesses in GMNs. Due to the limited on-chip memory space and irregular memory access patterns to graphs, nodes are frequently read/written between the on-chip buffer and memory. To quantify the off-chip data accesses in GMNs, we profile the node-reusing distances in GraphSim [5] with three graph classification datasets: AIDS, COLLAB, and REDDIT-BINARY (RD-B). We set node feature dimension to 64 and batch size to 32, following typical settings in the literature. We define the reuse distance as the number of unique nodes between two references to the same node, and the node has to be reloaded from memory if the distance exceeds the input buffer's capacity (128KB [42]). Figure 4 summarizes the CDFs (Cumulative distribution function) of reuse distances for three datasets. From the figure, most of the revisits are missed. This is due to the small on-chip input buffer used. A naive solution would need to enlarge the input buffer. However, this is not feasible and not scalable in accelerator designs. For example, AIDS demand about $4\times$ buffer size to mitigate revisit misses, yet

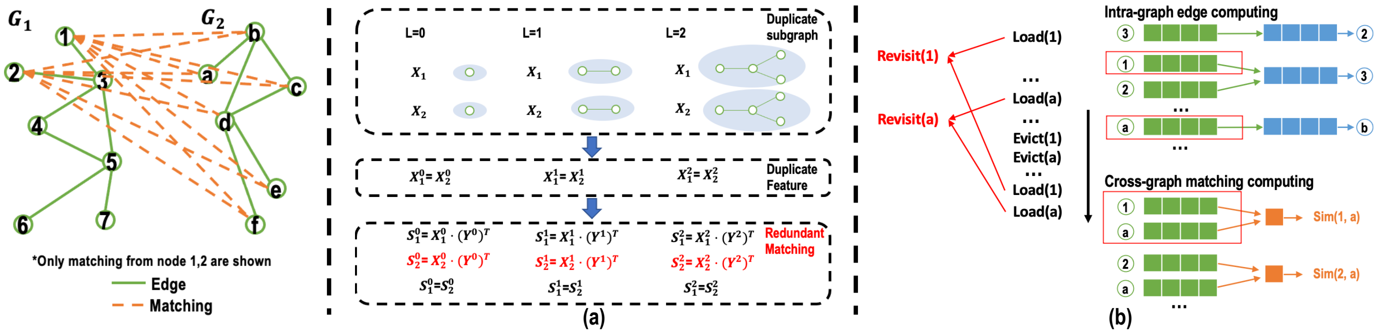


Fig. 5. Illustration of (a) Redundant Matching and (b) Redundant Revisiting in GMNs.

REDDIT-BINARY [43] dataset demands about $128\times$ buffer size. The demands can be further exacerbated for larger graphs.

C. Redundant Matching

Duplicate node features. In GMNs, the node feature in layer l (i.e., X_i^l) captures the l-hop subgraph information around node i. Therefore, if there exists another node j, whose l-hop neighbors form an isomorphic subgraph as that of i, both nodes will have the same feature in layer l (i.e., $X_i^l = X_j^l$). The isomorphic subgraphs are fairly common in real-world datasets, such as the same molecular within a macromolecule or the duplicate components within an object in point clouds.

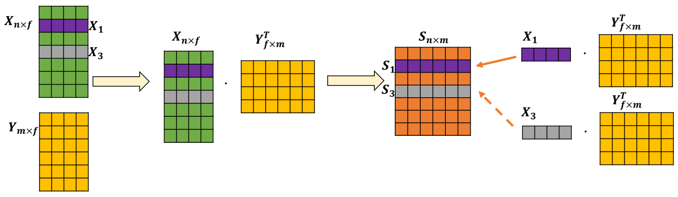


Fig. 6. Exactly same matching results caused by duplicate node features.

Redundant matching. In the following discussion, we assume that X^l and Y^l are node features from the target graph (i.e., G_1) and the query graph (i.e., G_2), respectively. As described in Equation (2), the i-th row in the similarity matrix S_l, which describes how similar node i compared with nodes in the other graph, is determined by i-th row in X^l and all the columns in $(Y^l)^T$, i.e., $S_i^l = X_i^l (Y^l)^T$. If there exist node i and node j in the target graph that have $X_i^l = X_j^l$, we have $S_i^l = S_j^l$ since $X_i^l (Y^l)^T = X_j^l (Y^l)^T$. The computation of the $j-$th row of the similarity matrix is redundant and thus can be optimized. In the example in Figure 6, $X_{n\times f}$ represents node features from an n-node target graph, and each node has f features; and $Y_{m\times f}$ represents node features from an m-node query graph and each node has f features. $S_{n\times m}$ represents their matching results in which row i describes similarities between node i and all the nodes in the target graph, and column j describes similarities between node j and all the nodes in the target graph. Assume for nodes 1 and 3, we have $X_1 = X_3$. We can derive $S_1 = S_3$. Therefore, we can skip computation operations and memory accesses for computing S_3 and copy from S_1 directly without jeopardizing accuracy. Note that the same principle is applicable when switching target and query graphs, in which case we find duplicates in two columns of the similarity matrix.

In this paper, we define **unique matching** as a matching from a node in the target graph to a node in query graphs whose computation has not been conducted before; and **redundant matching** as a matching between two nodes whose computation has already been conducted. Figure 5(a) illustrates an example of redundant matching. Assume we conduct similarity analysis for two graphs G_1 and G_2. For simplicity, we assume both graphs are unlabelled, and each node has the same initial feature set, e.g., $X_1^0 = X_2^0$. By conducting GNN embedding at layer l, GMNs collect information from neighbors within l-hop neighbors. When l=1, we have $X_1^1 = X_2^1$ since only $node_3$'s information is sent to $node_1$ and $node_2$, respectively. Similarly, we have $X_1^2 = X_2^2$ as both nodes share the same 2-hop neighbors. For the similarity matrix computation, their corresponding rows are exactly identical, as shown in the figure. Depending on the computation order, only the first computation of the two rows is necessary, while the other is redundant.

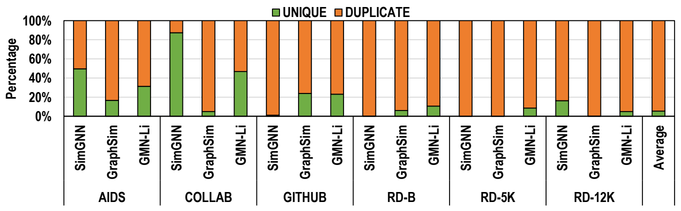
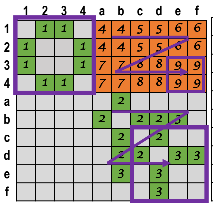


Fig. 7. Ratio between Redundant and Unique matching.

To quantify the potential redundant matching, we study three GMNs (SimGNN, GraphSim, and Graph-Matching-Network(GMN-Li) in Table I Section V). We train the models on the datasets mentioned in Section V and plot the ratios between redundant matching and unique matching in Figure 7. From the figure, we observe over 90% redundant matching on average. The scenarios are prevalent in datasets with large-sized graphs (i.e., graphs over 100 nodes), given that more subgraph patterns are replicated. Based on this observation, removing those redundant matching can effectively reduce the computation in the matching stage in GMNs.

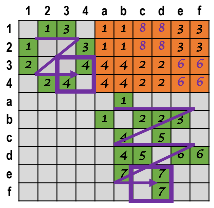
D. Redundant Node Revisiting

The node matching stage also suffers poor data locality due to excessive memory access. During the node embedding stage, the GMNs load and aggregate source-node features for each edge within the same graph. During the node matching stage, the same features are used to compute the similarity



Step	Input Nodes	Edges(src, dest)	Matching(X, Y)	Total Miss Count
1	1,2,3,4	A[1:4][1:4]	NONE	4
2	a,b,c,d	A[a:d][a:d]	NONE	8
3	b,d,e,f	A[e,f][a:d]→A[b,d][e,f]	NONE	10
4	1,2,a,b	NONE	S[1,2][a,b]	14
5	1,2,c,d	NONE	S[1,2][c,d]	16
6	1,2,e,f	NONE	S[1,2][e,f]	18
7	3,4,a,b	NONE	S[3,4][a,b]	22
8	3,4,c,d	NONE	S[3,4][c,d]	24
9	3,4,e,f	NONE	S[3,4][e,f]	26

(a)



Step	Input Nodes	Edges(A[src][dest])	Matching(S[X][Y])	Total Miss Count
1	1,2,a,b	A[1,2][1,2],A[a,b][a,b]	S[1,2][a,b]	4
2	3,4,c,d	A[3,4][1,2],A[c,d][a,b]	S[3,4][c,d]	8
3	1,2,e,f	A[1,2][3,4],A[e,f][a,b]	S[1,2][e,f]	12
4	3,4,a,b	A[3,4][3,4],A[a,b][c,d]	S[3,4][a,b]	16
5	3,4,c,d	A[a,b][c,d]	NONE	18
6	3,4,e,f	A[e,f][c,d]	S[3,4][e,f]	20
7	3,4,b,d	A[b,d][e,f]	NONE	22
8	1,2,c,d	NONE	S[1,2][c,d]	25

(b)

Fig. 8. Examples of (a) single intra-graph window and (b) double independent intra-graph window. The Input Nodes show on-chip input nodes with missed nodes highlighted. The Edges indicate node embedding, and Matching represents node matching pairs enclosed in the window step. The total miss count accumulates on-chip input buffer miss in terms of the node number.

between nodes. We use the same example pair (G_1, G_2) in Figure 5 to illustrate the redundancy in memory accesses. Given the case depicted in Figure 5(b), the features of $node_1$ in G_1 are first accessed during the node embedding stage to compute edge ($node_1$, $node_3$), and the features of $node_a$ from G_2 are read to compute edge ($node_a$, $node_b$). Due to the limited on-chip buffer capacity, these features are possibly evicted for other source nodes of edges and have to be loaded to the on-chip buffer again to compute $sim(node_1, node_a)$. These redundant revisits cause extra off-chip memory accesses and jeopardize latency and energy efficiency.

Existing works towards GNN acceleration focus on feature reusing within single graphs [8], [13], [14], [20], [25], [42]. When the approaches are employed for GMNs, the redundant revisit remains. We take the HyGCN [42] design as a representative example and show the limitation of these single-graph-based medthods. The edges within a graph can be represented by an adjacent matrix A, where $A(i, j)$ indicates an edge from i to j. Since real-world graphs are usually large and do not fit in the on-chip storage, existing GNN computing frameworks usually adopt a sliding window to process only part of the graph in each step. Specifically, a M×N window is placed on a graph's adjacency matrix and locates certain rows (a, a+M) and columns (b, b+N). The M and N are determined by the buffer size, and are usually much smaller than the size of A. Features of corresponding nodes are loaded into on-chip buffer for edge computation. The window slides column-wisely or row-wisely until all the edges are processed.

An example of adopting a single intra-graph sliding window for GMNs is shown in Figure 8(a). The workload within one GMN layer can be represented by a global adjacency matrix. The row index shows the source node of G_1 and the destination node of G_2, and the column index shows the source node of G_2 and the destination node of G_1. The green cells indicate intra-graph edges, and the orange cells indicate cross-graph node pairs to be matched. Assuming the input/output buffers hold four nodes in each, we can hold features of four nodes, and there is a 4×4 sliding window. First, the node embedding stage is executed for G_1(i.e., step 1) and G_2 (i.e., steps 2-3). Next, the node matching stage is executed (i.e., steps 4-9). During steps 1-3, the node embedding stage visits all the nodes, yet these nodes must be revisited in steps 4-9 for matching usage. The scheme causes 26 node misses in total, as depicted in Figure 8(a).

A possible optimization is to read features from both graphs and conduct *intra-graph* and *cross-graph* computing on-chip at the same time. However, problems remain due to *incomplete comparison* issue. Suppose we process two graphs in parallel and look for available matching pairs for possible feature reusing. The input buffer must be split because we need to handle inputs from both graphs. Here we follow a static partitioning that assigns 1/2 of buffer capacity to each graph. As shown in Figure 12(b), many nodes are evicted during window steps 1-3 without completely comparing with another graph. For instance, $node_1$ and $node_2$ are evicted after step 1 without matching with node c-f, then cause a miss at step 3. These incomplete comparisons result in a poor data locality. As shown in Figure 12(b), it causes 25 node misses in total. To this end, neither the existing accelerator nor simple optimizations can handle redundant node revisiting efficiently.

In summary, there exist substantial redundant computation and memory accesses in GMN computing. However, existing approaches overlook these redundancies and cannot mitigate them efficiently through simple design enhancement.

IV. Design

A. Overview of CEGMA

We propose a Coordinated Elastic Graph Matching Accelerator (CEGMA) for GMN to address the aforementioned challenges. The overall workflow of CEGMA is shown in Figure 9(a). First, Elastic Matching Filter (EMF) (❶) removes redundant matching by detecting and affiliating duplicate nodes. Next, Cross Graph Coordinator (CGC) (❷) removes redundant revisits by controlling a joint sliding window on the global adjacency matrix.

The overall architecture of CEGMA is presented in Figure 9(b). It consists of four components: SRAM buffers that hold on-chip data, a computing engine containing a MAC array, and two proposed components, EMF and CGC. The EMF (b)① handles duplicate node detection and affliction. The CGC (b)② solves the redundant node revisits.

B. Elastic Matching Filter

In the node matching stage, EMF filters all the redundant matching and only leaves unique matching. It prepares elastic matching metadata for layer l based on node features from the layer $l-1$ outputs. To facilitate description, we name nodes whose features are the first time seen as **Unique Node** and nodes whose features are seen before as **Duplicate Node**.

The overall algorithm of EMF is presented in Algorithm 1. It generates two sets that are cached on-chip for the subsequent layer usage: 1) The RecordSet that memorizes

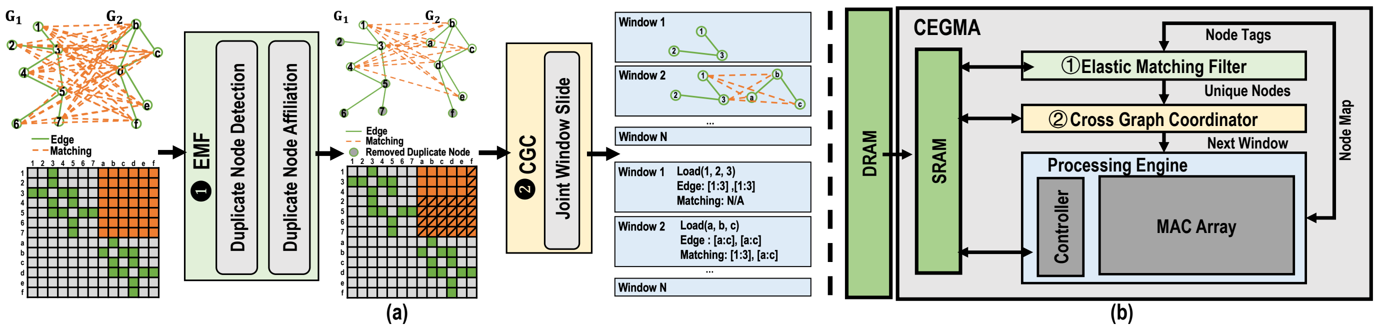


Fig. 9. Overview of the (a) architecture and (b) workflow in CEGMA.

Algorithm 1: Elastic Matching Filter

```
Output: $R_l$: RecordSet for non-duplicate nodes input to layer $l$; $M_l$: TagMap
        for duplicate node input to layer $l$
Input  : $X_{l-1}$: Node feature outputs from layer $l-1$
1 $R_l = \{\}, M_l = \{\}$;
2 for $x_i \in X_{l-1}$ do
3     $h_i$ = Hash($x_i$);
4     if $h_i \in R_l$ then
5         $M_l \leftarrow (i, match)$;
6     else
7         $R_l \leftarrow (i, h_i)$;
8 return $R_l, M_l$;
```

unique matching need to be computed, and 2) the TagMap that indicates redundant matching so they can be directly retrieved from previous results. As discussed in III-C, redundant matching appears in the entire row or column, which can be represented by the index of the duplicate node. We format entries in the sets as follows: i) Entry in the RecordSet is formatted as (*Unique_Node_idx*, *Unique_Node_tag*). The *Unique_Node_idx* specifies the index of a unique node and the *Unique_Node_tag* holds hash tag of node's feature. ii) Entries in TagMap affiliate duplicate nodes to unique nodes recorded in RecordSet. They are formatted as (*Node_idx*, *Unique_node_idx*). The *Duplicate_Node_idx* specifies the index of a duplicate node, and *Unique_node_idx* specifies the index of the unique node who holds the same results and has been recorded in the RecordSet.

We use tags to represent node features to save memorization and comparison overheads. Compared to the original features that usually consist of tens to hundreds of floating-point values, a single tag value costs much less on-chip memory and can simplify the comparison among. The output feature of node i is first hashed into a tag h_i, a 32-bit unsigned value, as depicted in line 3. We choose XXHash [9] to calculate the value. XXHash is a non-cryptographic hash algorithm that generates the hash value of input bit-streams in three steps: 1) It divides input bit-steam (B) into 4 independent sub-bit-streams $((b_1, b_2, b_3, b_4))$. 2) In each sub-bit-stream (i.e., b_k), it consumes 4 bytes to update state value (s_k), following the equation $s_k = rotateLeft(s_k + b_k * Prime2, 13) * Prime1)$. The $Prime1, Prime2$ in the equation are preset prime constants. The equation can be implemented in hardware using MAC and bit-shift operations. 3) In the end, it merges all four states (i.e., s_1, s_2, s_3, s_4) to generate a final hash value H. The conflict rate of XXhash is low. For instance, given 256 bytes of input bit-streams with 100 billion hashes, there are only 314 conflicts (i.e., the conflict rate is $\sim$0.0000314%) [9]. In all our experiments, we include the impact caused by hash conflicts. However, since the conflict rate is very low, we observe that the conflicts have little impact on the final matching accuracy (no conflict observed in our experiments). Due to the uniformity of hash functions, the generated tag h_i equals another tag h_k if their nodes have the same features $x_i = x_k$. Then we will have the same matching results for these two nodes in the next layer. As shown in line 4, we search tags in the RecordSet. If it is found, we get a duplicate node and memorize its index (i.e., i) and its counterpart unique node index (i.e., k) in TagMap. If it is not found, we record its index and tag in the RecordSet.

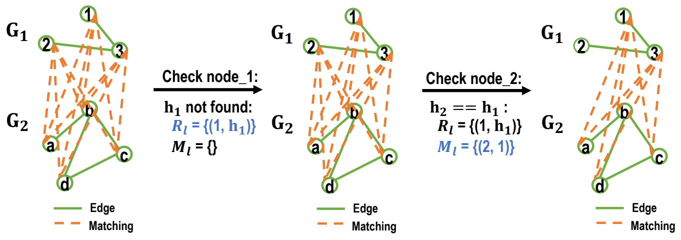


Fig. 10. An example of Elastic Matching Filter steps.

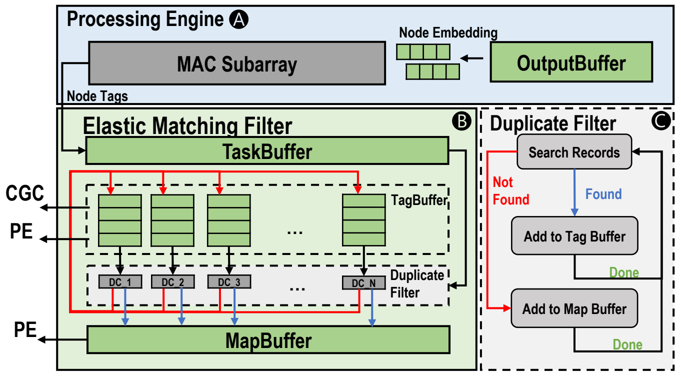
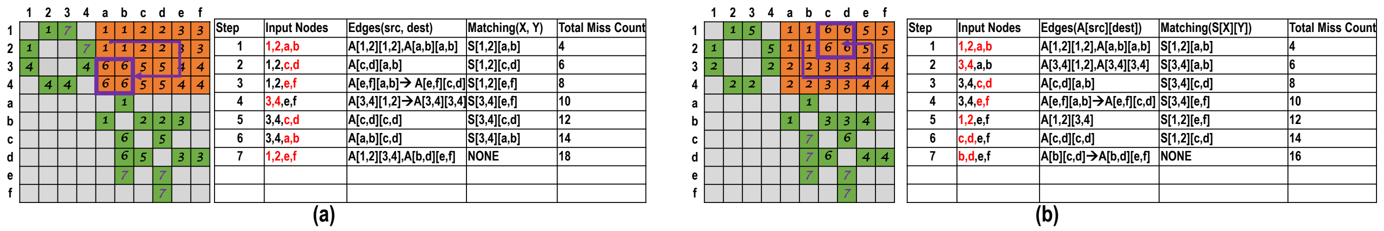


Fig. 11. Architecture of Elastic Matching Filter.

We illustrate an example in Figure 10, assuming there is a graph pair (G_1, G_2) such that $node_1$ and $node_2$ have the same features in the layer l. When the $node_1$ is checked, its tag h_1 is calculated and is not found in the RecordSet R_l. In this case, record (1, h_1) is added into R_l. When the $node_2$ is checked, its tag h_2 is calculated and is found duplicate with h_1 in the RecordSet entry (1, h_1). In this case, mapping (2, 1) is added into the TagMap M_l. The process continues until all nodes from the previous layer are digested.



(a)

Step	Input Nodes	Edges(src, dest)	Matching(X, Y)	Total Miss Count
1	1,2,a,b	A[1,2][1,2],A[a,b][a,b]	S[1,2][a,b]	4
2	1,2,c,d	A[c,d][a,b]	S[1,2][c,d]	6
3	1,2,e,f	A[e,f][a,b]→ A[e,f][c,d]	S[1,2][e,f]	8
4	3,4,e,f	A[3,4][1,2]→A[3,4][3,4]	S[3,4][e,f]	10
5	3,4,c,d	A[c,d][c,d]	S[3,4][c,d]	12
6	3,4,a,b	A[a,b][c,d]	S[3,4][a,b]	14
7	1,2,e,f	A[1,2][3,4],A[b,d][e,f]	NONE	18

(b)

Step	Input Nodes	Edges(A[src][dest])	Matching(S[X][Y])	Total Miss Count
1	1,2,a,b	A[1,2][1,2],A[a,b][a,b]	S[1,2][a,b]	4
2	3,4,a,b	A[3,4][1,2],A[3,4][3,4]	S[3,4][a,b]	6
3	3,4,c,d	A[c,d][a,b]	S[3,4][c,d]	8
4	3,4,e,f	A[e,f][a,b]→A[e,f][c,d]	S[3,4][e,f]	10
5	1,2,e,f	A[1,2][3,4]	S[1,2][e,f]	12
6	c,d,e,f	A[c,d][c,d]	S[1,2][c,d]	14
7	b,d,e,f	A[b][c,d]→A[b,d][e,f]	NONE	16

Fig. 12. Illustrated examples of (a) joint window slides row-wisely and (b) coordinated joint window using the same example and representation as Figure 8.

We present the detailed architecture design of EMF in Figure 11. The MAC subarray (A) computes the tags h_i for each node i. The subarray output 32-bits h_i at the end of each row. We pack the node index i with its tag h_i and push it to the TaskBuffer in the EMF. The EMF (B) and the processing engine (A) work in a "producer-consumer" pipelined fashion. Specifically, the processing engine calculates the hashtags for node indices and stores the information in the TaskBuffer. Then, the EMF retrieves the hashtag and node index from the TaskBuffer to eliminate the redundancy. The EMF(B) consists of four elements: The TaskBuffer holds 64-bits (i,h_i) entries that are produced by the processing engine and consumed by the duplicate filter. It is implemented by FIFO and holds 64-bits (i,j) entries for TagMap (i.e., M_l). The TagBuffer holds 64-bits (i,j) entries for RecordSet (i.e., R_l), it is implemented by a set of loopback FIFOs, and each FIFO holds a subset of R_l so that the lookup operations can be paralleled. DuplicateFilter handles main logic as lines 4-7 in Algorithm 1. It controls a set of duplicate comparators (DC) by a Finite State Machine(FSM), whose logic is depicted in (C). Each DC equips a 32-bit identity comparator and looks up tag h_i by comparing it with the last 32 bits of the entry popped from a subset of TagBuffer. If they are identical, FSM stops searching on all other DC and writes i with the first 32 bits of the found entry to the MapBuffer. Otherwise, if none of the DCs find an identical entry, the 64-bits task entry is inserted into the TagBuffer. Specifically, entries are added to the subsets in a round-robin manner so that each subset contains a balanced amount of entries. The generated TagBuffer and MapBuffer can be accessed by CGC and PE for computation in the following layer.

C. Cross Graph Coordinator

Cross Graph Coordinator (CGC) is designed to mitigate redundant revisits issues between node embedding and matching stages. It slides a single joint window on the matching matrix. On the one hand, the window jointly maps intra-graph edges and cross-graph matching pairs to facilitate inter-stage data reuse. On the other hand, the sliding path is heuristically determined to maximize inter-stage data locality.

We use the same example shown in Figure 8 to illustrate the scheme. Recall our discussion in Section III-D, adopting double independent windows is inadequate to reduce the node revisiting. The inefficiency comes from the incomplete comparison. To prevent incomplete comparisons in double independent windows, we adopt a joint sliding window. Instead of sliding two windows within each graph independently, we place one joint window sliding on the cross-graph adjacency area (i.e., the top-right area in the global adjacency matrix). The joint window maps both intra-graph edges and cross-graph matches. We present a row-wise sliding example in Figure 12(a). During steps 1-6, a joint window slides over the matching matrix. In step 7, the remaining edges are collected. The joint window has the following property: (1) It only goes row or column-wise so that one side of the window can always be fully reused to be compared with the other graph. As shown by steps 1-3 in the example, it goes through node a-f while keeping $node_1$ and $node_2$ unchanged. (2) It turns and starts at the closes start point of new rows/columns instead of jumping back to the zero-index point. For instance, step 3 finishes all the matching on $node_1$ and $node_2$. Instead of starting from $node_a$ and $node_b$ on $node_3$ and $node_4$ rows, the window keeps $node_e$ and $node_f$ and compares them with new rows. Therefore, except for the first step, we only update one side of the window and the other one is stationary. We can always have new coming nodes matched with on-chip stationary nodes while used for edge computation. In other words, we "pause" execution on one graph by freezing the side in one direction and "activate" execution on the other graph by sliding the window in the other direction.

We next address the question of when the joint window finishes one row/column. That is, to maximize data reuse, should it consume the remaining matching matrix row-wisely or column-wisely? As shown in Figure 12(a), sliding over the matching matrix always updates one side (e.g., two nodes in the example). In other words, it incurs the same number of missing node counts for steps that contain matching (e.g., Steps 1-6 in the example). Differences happen in the steps after the matching is completed (e.g., step 7)—the window slides over rows/columns during these remaining steps to schedule the remaining edges. Apparently, fewer remained edges lead to less extra cost at these steps. The intuition drives us to propose the coordinated joint window. It slides the same way as a joint window but selects its sliding direction based on a heuristic, Approximate Outlier Estimation (AOE), as presented in Algorithm 2.

The intuition behind AOE is: Keep nodes with fewer remaining edges. Whichever nodes have been kept stationary on-chip will have their matching completed; those nodes with remained edges need to be revisited after matching. To this end, for each of the two sides (input nodes from G_1 and G_2), AOE estimates the number of nodes with the least amount of remaining edges (remaining degree). We call these nodes *outliers*. Next, AOE keeps the side that contains more outliers stationary. As presented in the algorithm, lines 2-13 detect the

Algorithm 2: Approximate Outlier Estimation

```
Output: $g_m$: sliding direction (1—row-wise, 0—column-wise).
Input : $S_0$: on-chip node set represented by row index; $S_1$: on-chip node set
        represented by column index.
1  $n_0 = 0, n_1 = 0, Threshold = MAX\_INT$;
2  for $q \in S_0 \bigcup S_1$ do
3      if $q.remains < Threshold$ then
4          $Threshold = q.remains$;
5          if $q \in S_0$ then
6              $n_0 = 1, n_1 = 0$;
7          else
8              $n_1 = 1, n_0 = 0$;
9      else if $q.remains == Threshold$ then
10         if $q \in S_0$ then
11             $n_0 + = 1$;
12         else
13             $n_1 + = 1$;
14 if $n_0 > n_1$ then
15     return $g_m = 0$;
16 else
17     return $g_m = 1$;
```

number of outliers by counting outlier nodes whose remained edges are less than the threshold. The threshold is updated, and the counter is reset if a new minimum remaining degree is observed. Figure 12(b) illustrates an example of coordinated sliding window. In step 4, the window can go along rows 3, 4 or columns c, d. It evicts $node_c$ and $node_d$ instead of following its original column-wise sliding plan. Hence, $node_3$ and $node_4$ are matched with $node_a$ to $node_f$ and never used again. In contrast, if $node_c$ and $node_d$ are kept on-chip, edge (c, d), (d, c), (d, e), (e, f) will be revisited after step 6.

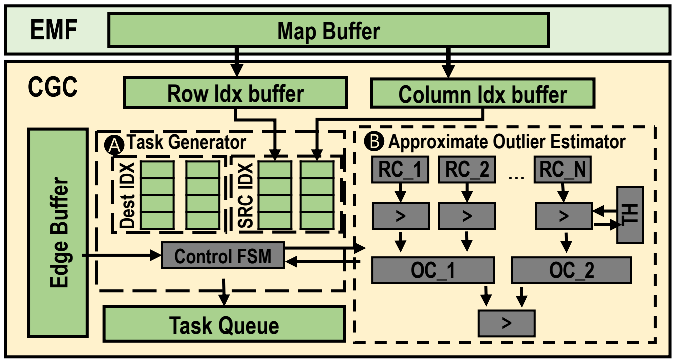


Fig. 13. Architecture of Cross Graph Coordinator.

We present architecture of CGC in Figure13. The entries in the MapBuffer are popped out to the Row_Idx Buffer and Column_Idx Buffer following the producer-consumer pattern. The Task Generator (A) keeps window sides information in the SRC_IDX cache and intra-graph destination node indices in Dest_IDX cache. The Control FSM operates as: i) if the window is located, it passes Dest_IDX and SRC_IDX to Task Queue; ii) if the window needs to be updated, it overwrites SRC_IDX caches based on direction; and iii) if direction needs to be decided, it collects row/columns from the edge buffer based on indices in the SRC_IDX, then sends them to the AOE (B). The AOE makes decision based on Algorithm 2. Row/columns from the edge buffer are fed into Remains Counters (RCs), they count the remaining edges from nodes, and their following comparator decides if nodes are outliers. The Outlier Counters (OCs) count number of outliers based on comparison results. Both RCs and OCs are implemented by 8-inputs parallel counters.

D. MainStream Execution in Processing Engine

We show the architecture of the Processing Engine and its dataflow in Figure 14. It consumes tasks from the Task Queue when finishing the tasks defined by the current window. Also, it utilizes node maps generated by the EMF to eliminate redundant matching.

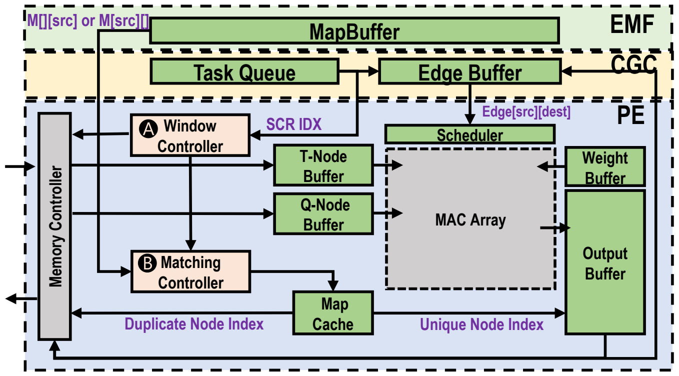


Fig. 14. Architecture of Processing Engine.

The input buffer are evenly partitioned into a T-node buffer and a Q-node buffer. The T-Node buffer stores input node features from the target graphs, and the Q-Node buffer stores input node features from the query graphs. The weight buffer stores the neural network parameters used by the node embedding stage, the Output buffer stores results, and the Map-Cache stores the TagMap entries for duplicate nodes.

The Window Controller (A) handles tasks from the CGC. When there is a task from the Task Queue. The corresponding edges (i.e., Edge[src][dest]) are fetched to the edge buffer and then utilized to control aggregation from the input nodes. Meanwhile, the window controller reads the input nodes index (SCR IDX). It checks missing nodes and marks one of the input buffers (i.e., T-node and Q-node buffers) as active input buffer, the other one as stationary input buffer. The memory controller load missing node features to the active input buffer. For matching, the features in active input buffer are streamed vertically across the MAC array, and the node features in stationary input buffer are streamed horizontally across the MAC array. For edge processing, only features from active input buffer are streamed vertically, while edge weights are streamed horizontally.

The Matching Controller (B) eliminates redundant matching computation using entries from the MapBuffer in EMF. It first collects entries from the MapBuffer into Map Cache based on the source node indices. The Map is collected and utilized in two types: (a) For GMNs that directly write similarities back to DRAM for later usage (e.g., SimGNN [4] and GraphSim [5]), matching controller looks for entries that contain input nodes as unique nodes. The entries are brought to the Map cache, and the memory controller uses their duplicate node indices to write back result to DRAM. We broadcast unique matching results to all redundant matching references in this case. (b) For GMNs that use matching results within each layer, such as Graph-Matching-Network (GMN-Li) [24], the matching controller collects entries that contain input nodes as duplicate nodes. The entry is used to access output results from their unique nodes. In this case, the unique nodes' results are cached on-chip and reused by duplicate nodes. The type is statically set before runtime.

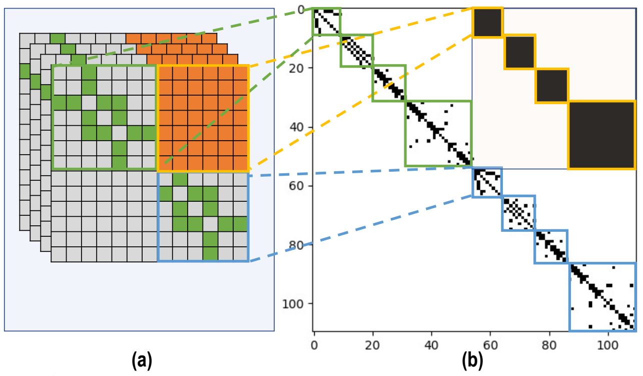


Fig. 15. An example of (a) graph batch with 4 input graph pairs and (b) corresponding global adjacency matrix A.

The CEGMA takes batched graph pairs to improve the hardware utilization. Specifically, a single global adjacency matrix is created for a batch of graph pairs, as shown in Figure 15. Given a batch of graph pairs (e.g., four graph pairs shown in Figure 15(a)), we combine edges in target graphs in the top-left area in Figure 15(b) (i.e., A[0:60][0:60]). Then we have edges in query graphs in the bottom-right area in Figure 15(b) (i.e., A[60:120][60:120]). Lastly, we keep cross-graph matching pairs in the top-right area in Figure 15(b)(i.e., A[0:60][60:120]).

V. Evaluation

A. Methodology

Models. We evaluate CEGMA using three recent GMNs: 1) GMN-Li [24], 2) GraphSim [5], which employs layer-wise cross-graph node matching, and 3) SimGNN [4], which employs a three-layer GCN and computes cross-graph similarity based on the third-layer output. We set the batch size as 32 for all the models. The model details are described in Table I. The layer configurations are presented as LayerType ([hidden sizes]). And the similarity describes which type of similarity (cosine/dot-product/euclidean) is computed. We name the special GNN used in [24] as MGNN, in which a special edge Multilayer-Perceptron (MLP) is used to generate intra-graph edge messages to be aggregated. And it uses another updating MLP to combine aggregated intra-graph messages from edge and cross-graph messages from matching into nodes.

Datasets. We use six commonly-used real-world graph classification datasets [26], as depicted in Table II. Specifically, AIDS [33] contains small-sized graphs of molecular compounds constructed from the AIDS Antiviral Screen Database of Active Compounds. Github Stargazers [34] (GITHUB) contains middle-sized graphs in which the nodes represent authors and edges represent their relationship. COLLAB [43] contains middle-sized ego-collaboration graphs collected from

TABLE I
Details of GMNs models.

Model	Layers (Type[hidden_sizes])	Similarity
GMN-Li	MLP[1,64], 5*(MGNN[64,64,64], MATCHING[64,64], MLP(64*3,64,64)) READOUT[64,128,128]	Euclidean
GraphSim	3*(GCN[1,64], SIM[64,1]), 3*(CNN[1,16,32,64,128]) MLP([128*3,128,64,32,16,1])	Cosine
SimGNN	3*(GCN[1,64]),SIM[64,1], 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	Graph Scale
AIDS	15.69	16.20	200	small-sized
COLLAB	74.49	2457.78	500	small-sized
Github-Stargazer(GITHUB)	113.79	234.64	1273	middle-sized
REDDIT-BINARY(RD-B)	429.63	497.75	200	middle-sized
REDDIT-MULTI-5K(RD-5K)	508.52	594.87	500	large-sized
REDDIT-MULTI-12K(RD-12K)	391.41	456.89	1193	large-sized

scientists in the fields. REDDIT-BINARY (RD-B), REDDIT-MULTI-5K (RD-5K), and REDDIT-MULTI-12K (RD-12K) [43] contain large-sized graphs in which nodes represent users and edges represent relationships among users. We follow the setting of the classification task in [24] to generate similar/dissimilar graph pairs by randomly substituting edges in original graphs. Specifically, given an original graph G_{orig}, we substitute $n_{positive} = 1$ edges to produce its similar counterpart G_{pos}, and $n_{negative} = 4$ edges to produce its dissimilar counterpart G_{neg}. We also follow [24] to split the dataset with the ratio in *train:val:test=8:1:1*. Then we use graph pairs in the test set for the evaluation. The numbers of graph pairs in these test sets are shown in Table II.

Simulation and platforms. We implement CEGMA in a cycle-accurate simulator from scratch to compare with the baseline and the state-of-the-art GNN accelerators. The simulator is trace-driven: We first run the GMNs on the CPU, and profile trace files include node features, adjacency matrices, weights, and operations within each layer of GMNs. Next, the simulator reads these files and then simulates the execution of CEGMA. Note that all the information in the trace file is application dependent, and they are not tied to particular platforms where the traces are obtained. Therefore, if the GNM is implemented in a different framework (e.g., TensorFlow), we can profile the same information and use it as input to the CEGMA simulator. The power and area of buffers in CEGMA are estimated using CACTI [29]. We synthesize GECMA on RTL and estimate its area cost on TSMC 14nm process. Other platforms are presented in Table III. We compared the CEGMA with the following different GMN implementations:

TABLE III
Hardware configuration of CEGMA and other platforms.

Platform	PyG-CPU	PyG-GPU	HyGCN	AWB-GCN
Compute Units	2.60 GHz @ 12 cores	1.25GHz @ 5120cores	1GHz @ 32SIMD 16cores and 32x128 systolic array	1GHz @ 4096 PEs
On-chip Memory	32MB	34MB	128KB (Input) + 24MB (Others)	128KB (Input) + 24MB (Others)
Off-chip Memory	119.21 GB/s DDR4	∼900GB/s HBM 2.0	256GB/s HBM 1.0	256GB/s HBM 1.0
Area	-	-	7.8 mm^2(12nm)	-
CEGMA				
Module	**Configuration**			
EMF	32-bits Identity Comparator $\times$ 1024			
CGC	8-input Parallel_Counter $\times$ 34, 8-bit Magnitude Comparator $\times$ 33			
PE	128$\times$ 32 MAC array			
On-chip Memory	128KB(TNode+QNode)+6.8MB(Others) SRAM			
Off-chip Memory	256GB/s HBM 1.0			
Frequency	1 GHz			
Area	6.3 mm^2 (14nm)			
Area-Distribution-EMF	0.18%(logic) + 6.66%(buffer)			
Area-Distribution-CGC	0.01%(logic) + 11.79%(buffer)			
Area-Distribution-PE	53.58%(logic) + 27.78%(buffer)			

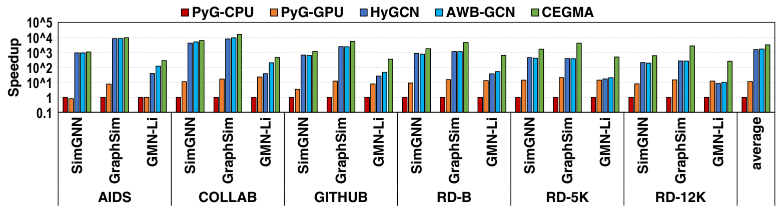


Fig. 16. End-to-End Speedup of prior GMN implementations and CEGMA over the baseline.

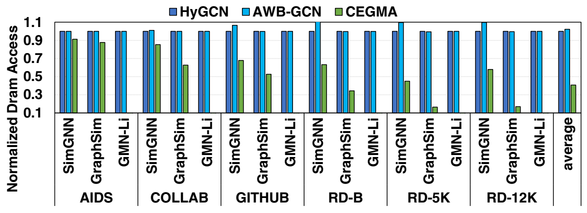


Fig. 17. DRAM access of CEGMA and GNN accelerators normalized by the DRAM access in HyGCN.

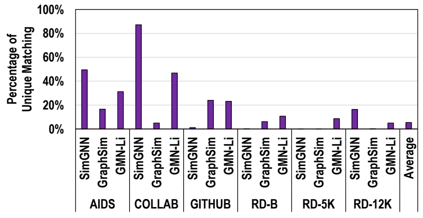


Fig. 18. Percentage of remained unique matching over original matching pairs.

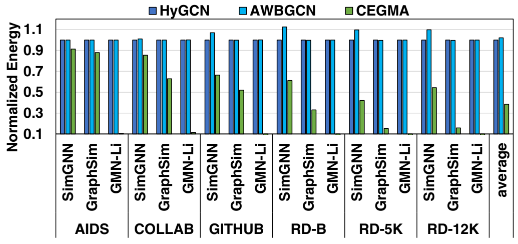


Fig. 19. Normalized energy consumption.

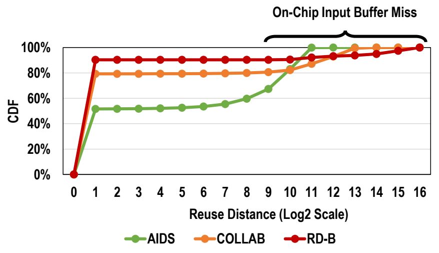


Fig. 20. Node reuse distances of GraphSim in CEGMA.

- **PyG-CPU (Baseline)**: The PyG-CPU runs GMN on a dual 12-core Skylake CPU (Intel Xeon Gold 6126 2.60 GHz). We implement GMNs using PyTorch and PyTorch-Geometric [32], and convert the model into TorchScript for better efficiency. We use the PyTorch version with MKL and OpenMP libraries to optimize parallelism on CPU.
- **PyG-GPU**: In PyG-GPU, we run GMN on NVIDIA V100 GPU using CUDA 10.1. We implement GMNs the same ways as PyG-CPU. This version leverages cuSPARSE and cuBLAS libraries to accelerate GMN computation on GPUs.
- **HyGCN**: HyGCN [42] is a GCN accelerator that accelerates the node embedding stage in GMN through sparsity elimination. We use the same configuration of HyGCN in [42]. To run GMN on HyGCN: 1) we keep its GCN optimizations for the node embedding stage; 2) for the node matching stage, it reads the input node features from the input buffer (if hit) or DRAM (if missing), computes the similarity using the general PE array within its combiner and writes back the matching results to memory. During the matching stage, it schedules window steps based on the similarity matrix (i.e., sliding window on S^l like what it does on A^l during node embedding). It runs at 1GHZ frequency.
- **AWB-GCN**: AWB-GCN [13] is an FPGA-based GCN accelerator. It balances the workload among PEs and reorders computation within GCN to reduce computations. For a fair comparison, we set its buffer sizes the same as buffer sizes in HyGCN. The steps to make GMN run on AWB-GCN are similar with HyGCN. The only difference is that we use all 4K PEs in the AWB-GCN to conduct similarity computation as AWB-GCN is a homogeneous accelerator. This differs from HyGCN, whose combiner only includes a subset of PEs. We also set 1GHZ frequency for AWB-GCN.

B. Overall Performance

We present the log-scale speedup of CEGMA over baseline (PyG-CPU) in Figure 16. From the figure, one can make the following observations. First, compared with PyG-CPU, PyG-GPU, HyGCN, and AWB-GCN, our proposed CEGMA achieves significant performance improvements, with an average of 3139×, 353×, 8.4×, and 6.5×, respectively. The major improvements come from the effective reduction in redundant matching and node revisiting. Second, different GMN models have different speedups. In particular, CEGMA performs better on those GMN models that require more similarity computation. For instance, the GMN-Li, CEGMA achieves 18.1× and 12.2× speedups over HyGCN and AWB-GCN, respectively. We also show the percentage of remaining unique matching after removing redundancy matching in Figure 18. As one can observe, CEGMA eliminates more than 90% matching computation on average. In contrast, for SimGNN, CEGMA achieves only 2.1× and 2.3× speedups over HyGCN and AWB-GCN, respectively. The main reason is that the SimGNN performs matching in its last layer, thereby having less optimization potential. Third, the speedup also varies across different datasets. CEGMA gets 3.1× and 1.5× speedups over HyGCN and AWB-GCN on AIDS, which contains relatively small-sized graphs (∼15.69 nodes per graph on average). When graph size grows, CEGMA yields higher speedups. For instance, in GITHUB (∼113.79 nodes per graph on average), the speedups are 5.8× and 3.9× over HyGCN and AWB-GCN, respectively. In contrast, the speedups on RD-5K (∼508.52 nodes per graph on average) reach 14.4× and 12.9× over HyGCN and AWB-GCN. Figure 18 also shows that more redundant matching is removed in large graphs (e.g., 97% in RD-5K) compared to small graphs (e.g., 67% in AIDS). This is because large graphs usually comprise more duplicated subgraphs than small graphs.

We next study the effectiveness of CEGMA in reducing DRAM accesses. Figure 17 shows DRAM accesses of HyGCN, AWB-GCN, and CEGMA. All the results are normalized to the DRAM access of HyGCN. From the figure, we make the following observations. First, CEGMA reduces 59% and 61% DRAM accesses compared to HyGCN and AWB-GCN, respectively. HyGCN and AWB-GCN show similar

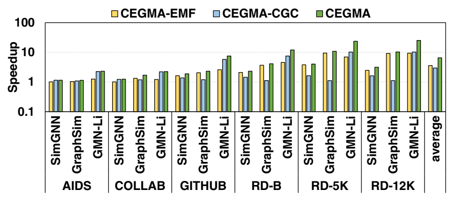


Fig. 21. End-to-End Speedup of CEGMA-EMF, CEGMA-CGC and CEGMA over AWB-GCN.

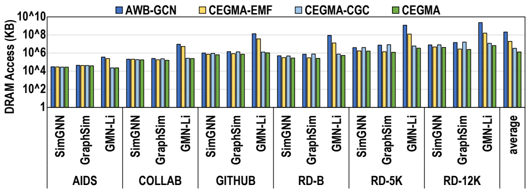


Fig. 22. DRAM access of CEGMA-EMF, CEGMA-CGC and CEGMA compare to AWB-GCN.

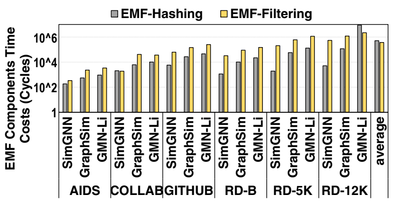


Fig. 23. Number of cycles of EMF-Hashing and EMF-Filtering.

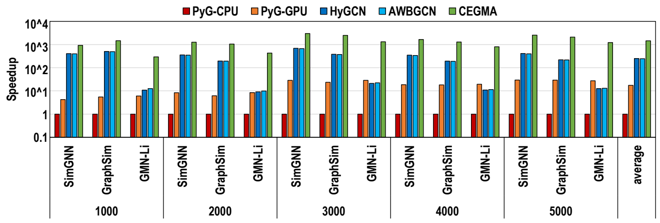


Fig. 24. Throughput on benchmarks in prior approaches and CEGMA.

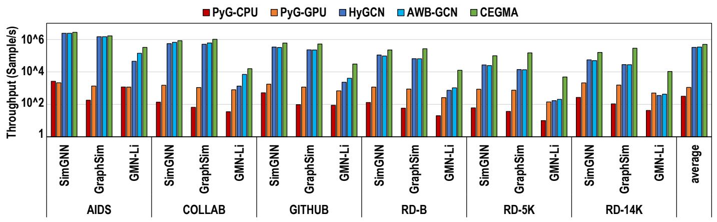


Fig. 25. Speedup of CEGMA and prior approaches on large graphs.

memory footprints since they both suffer redundant revisits of node features. In contrast, CEGMA significantly reduces the DRAM access because (i) CGC in CEGMA effectively reduces the redundant node revisits, and (ii) EMF in CEGMA reduces node visits for redundant matching. Also, the CGC significantly reduces the data reuse distances where the on-chip buffer can capture more reuses. To quantify this, we show the node reuse distances in Figure 20. It uses the same GMN model (GraphSim [5]) as Figure 4. One can observe that the node reuse distances are significantly reduced (e.g., in RD-B, 90.3% reuses are within the distance of 2^8, compared to only 0.02% reuses in Figure 4). Third, the amount of reduced DRAM accesses varies across different GMN models. For instance, CEGMA reduces 98.18% and 98.16% DRAM accesses in GMN-Li compared to HyGCN and AWB-GCN. In contrast, it reduces 31.6% and 38.14% DRAM accesses in SimGNN. This is because GMN-Li performs matching in each layer, thus benefiting more from reuse distance reduction and redundancy matching removal. Finally, DRAM access reduction also varies across different datasets. CEGMA reduces 69%, 81%, and 76% DRAM accesses on RD-B, RD-5K, and RD-12K, respectively. In contrast, it reduces DRAM accesses to 38%, 50%, and 61% on GITHUB, COLLAB, and AIDS. This is because CEGMA avoids more redundant matching in RD-B, RD-5K, and RD-12K, as shown in Figure 18.

We investigate the energy efficiency of CEGMA in Figure 19. The results are normalized to the energy consumption of HyGCN. On average, CEGMA consumes 63% and 62% less energy than HyGCN and AWB-GCN, respectively. The reason is two-fold: First, EMF significantly reduces redundant matching, leading to fewer amounts of computation and memory accesses, which reduces energy consumption. Second, CGC effectively improves inter-stage data locality, further reducing memory access, which brings energy benefits.

C. Breakdown Analysis

We further analyze the performance gain from each component in CEGMA. We evaluate (1) CEGMA-EMF where EMF is enabled and CGC is disabled; and (2) CEGMA-CGC where CGC is enabled and EMF is disabled. We compare the speedup and DRAM accesses with AWB-GCN as it performs best among GMN implementations mentioned before.

The results are shown in Figure 21. On average, CEGMA-EMF performs 3.6× speedup over AWB-GCN. The performance gain of EMF increases when the graph size grows. While CEGMA-EMF brings only 1.1× speedup over AWB-GCN on AIDS, it achieves 7.1× speedup over AWB-GCN on RD-5K. On the other hand, CEGMA-CGC gains an average 2.9× speedup over the baseline since CGC reduces node revisits. With growing graph sizes, CGC brings more performance gain as well. Specifically, it achieves 1.5× speedup on AIDS and 4.3× speedup on RD-5K. Compared to CEGMA-EMF, CEGMA-CGC achieves less speedup in large graphs. The reason is that less latency is caused by DRAM access when GMNs perform on large graphs. In smaller graphs, nodes from one graph are compared with fewer nodes from the other, incurring fewer reuses, so the PEs frequently wait for data to be loaded to the buffer. However, in larger graphs, nodes from one graph are compared with more nodes from the other, incurring more reuses, so the memory bandwidth is underutilized and waiting for computing to be finished.

We also show DRAM accesses of CEGMA-EMF and CEGMA-CGC in Figure 22. Both EMF and CGC reduce memory costs effectively. Compared to the AWB-GCN, EMF reduces DRAM accesses by 49% on average. The reduction comes from the omitted duplicate matching. The improvements are more significant in larger graphs. CEGMA-EMF gains 72% DRAM access reduction in RD-5K compared to 14% DRAM access reduction compared to AWB-GCN in AIDS. CGC also reduces DRAM access since it removes redundant revisits caused by node matching. On average, it reduces DRAM accesses by 34% compared to the AWB-GCN. The Algorithm 2 can achieve 90% precision compared to the optimal decisions. EMP brings more significant improvements than CGC since node matching dominates DRAM accesses in many cases. Hiding its DRAM accesses into node embedding has less benefit than cutting off the total amount of matching pairs.

To investigate the overhead brought by EMF, we show the absolute cycle counts of EMF components in Figure 23. The EMF-Hashing stands for time spent on computing hashtags of node features, and the EMF-Filtering stands for time spent on searching the same tags and generating masks for nodes. On average, EMF-Hashing takes 284 cycles and EMF-Hashing takes 429 cycles per graph—the overheads are less than one microsecond under 1GHz. Since CEGMA calculates hashing and filters for nodes in parallel, the overheads are ignorable and have little impact to the deadline requirements. Even in RD-12K, EMF-Hashing takes 1488 cycles (0.001ms) per graph and EMF-Filtering takes 655 cycles (0.0006ms), which are far less than deadline requirements in milliseconds (e.g., 20ms).

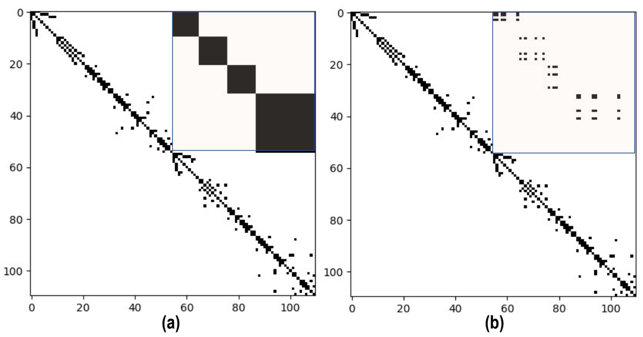


Fig. 26. An example of a four pairs batch workload (global adjacency matrix) (a) before and (b) after EMF.

Next, we show the inference throughput (i.e., graph pairs per second) in Figure 24. Overall, CEGMA achieves an average of $353\times$, $8.4\times$, and $6.5\times$ throughputs over the PyG-GPU, HyGCN, and AWB-GCN. It demonstrates that CEGMA provides excellent potential to satisfy the graph matching deadlines. For instance, in RD-K5, CEGMA can take 5000 query graphs per second on GMN-Li. In comparison, PyG-GPU takes 312 query graphs per second, and AWB-GCN takes 588 query graphs per second. Figure 26 shows the effectiveness of EMF via an example from AIDS. Given a batch of 4 graph pairs, EMF effectively reduces redundant matching (black pixels in the top-right area) in the global adjacency matrix.

D. Large Graphs

To investigate how CEGMA performs on large graphs, we generate graphs following the graph generation algorithm presented in [24]. Specifically, we generate 8 original graphs for each size and produce graph pairs as described in Section V-A. Figure 25 shows the log-scaled speedups over the baseline. We observe that CEGMA consistently achieves high performance over other platforms. The speedups on large graphs are more significant. For instance, with the graphs having 1000 nodes, CEGMA achieves $10.8\times$ and $9.6\times$ average speedup over HyGCN and AWB-GCN, respectively; the speedup increase to $37.5\times$ and $36.6\times$ with graphs having 5000 nodes. The major reason behind this is that there are more duplicate subgraphs in large graphs, leading to more duplicate computation and redundant memory access in matching. Therefore, our approach is more effective on large graphs.

VI. Related Work

Studies have revealed that GNNs computing exhibits not only a poor data locality but also a significant imbalance between computing and memory access [1]. Many hardware accelerators have been proposed to address these challenges [8], [13], [14], [20], [22], [25], [42]. *HyGCN* utilizes a heterogeneous architecture for aggregation and combination with a dedicated cooperative working mode to solve the hybrid execution in GNN [42]. However, *HyGCN* targets workload partitioning within single graphs and adopts the separate combination engine and aggregate engine, where the dense matrix multiplication is handled by part of the PEs. This leads to potential imbalanced throughput in GMNs since the dense comparison could potentially congest the combination engine while the aggregation engine could be stalled. *AWB-GCN* employs execution phase reordering with an auto-tuning workload balancing approach to solve the imbalance problem between nodes with different degrees [13]. Nevertheless, the architecture focuses on general GCN only, which assumes only sparse-dense matrix multiplication and no cross-graph computations, leaving potential inefficiencies in GMN computing. *I-GCN* adopts a node-reordering approach named *islandization* that reconstructs the graph to enhance runtime data locality and remove redundant computations [14]. However, the intra-graph islandization method does not consider cross-graph matching workloads. In summary, prior GNN accelerators focused on addressing poor data locality and hybrid execution challenges in the node embedding stage depicted as Equation 1. However, the proposed optimizations are insufficient to efficiently handle graph matching computing, where pairs of graphs and corresponding cross-graph computing are involved.

VII. Conclusion

In this work, we propose CEGMA, a software-hardware co-design accelerator, to address computing challenges in GNN-based Graph Similarity networks. We first propose an elastic matching scheme to detect and remove redundant matching in GMNs. Furthermore, we design a cross-graph coordination scheme to enhance data locality between GMN computing stages. Our approach effectively reduces both computing complexity and memory consumption in GMN execution. Experimental results show that CEGMA achieves $353\times$ and $6.5\times$ speedup over GPUs and state-of-the-art GNN accelerators.

Acknowledgement

The authors sincerely thank the anonymous HPCA reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, #2154973, #1910413, #1725657, and a startup funding from the University of Pittsburgh.

References

[1] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–38, 2021.

[2] S. Alam, I. Traore, and I. Sogukpinar, "Annotated control flow graph for metamorphic malware detection," *The Computer Journal*, vol. 58, no. 10, pp. 2608–2621, 2015.

[3] L. Bai, J. Pang, Y. Zhang, W. Fu, and J. Zhu, "Detecting malicious behavior using critical api-calling graph matching," in *2009 First International Conference on Information Science and Engineering*. IEEE, 2009, pp. 1716–1719.

[4] 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.

[5] 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.

[6] 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.

[7] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2006, pp. 129–143.

[8] X. Chen, Y. Wang, X. Xie, X. Hu, A. Basak, L. Liang, M. Yan, L. Deng, Y. Ding, Z. Du *et al.*, "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[9] Cyan4973, "Cyan4973/xxhash: Extremely fast non-cryptographic hash algorithm." [Online]. Available: https://github.com/Cyan4973/xxHash

[10] 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.

[11] K. Fu, S. Liu, X. Luo, and M. Wang, "Robust point cloud registration framework based on deep graph matching," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 8893–8902.

[12] Q. Gao, F. Wang, N. Xue, J.-G. Yu, and G.-S. Xia, "Deep graph matching under quadratic constraint," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 5069–5078.

[13] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[14] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1051–1063.

[15] I. Gog, S. Kalra, P. Schafhalter, M. A. Wright, J. E. Gonzalez, and I. Stoica, "Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 8806–8813.

[16] X. Guo, J. Hu, J. Chen, F. Deng, and T. L. Lam, "Semantic histogram based graph matching for real-time multi-robot global localization in large scale environment," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 8349–8356, 2021.

[17] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[18] 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.

[19] E. Kazemi, S. H. Hassani, and M. Grossglauser, "Growing a graph matching from a handful of seeds," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1010–1021, 2015.

[20] K. Kiningham, C. Re, and P. Levis, "Grip: a graph neural network accelerator architecture," *arXiv preprint arXiv:2007.13828*, 2020.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[22] J. Li, A. Louri, A. Karanth, and R. Bunescu, "Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 775–788.

[23] 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.

[24] 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.

[25] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, "Engn: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computers*, 2020.

[26] G. Ma, N. K. Ahmed, T. L. Willke, and S. Y. Philip, "Deep graph similarity learning: A survey," *Data Mining and Knowledge Discovery*, pp. 1–38, 2021.

[27] 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.

[28] N. Molton, A. J. Davison, and I. Reid, "Locally planar patch features for real-time structure from motion." in *Bmvc*, 2004, pp. 1–10.

[29] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[30] K. Ogaard, H. Roy, S. Kase, R. Nagi, K. Sambhoos, and M. Sudit, "Discovering patterns in social networks with graph matching algorithms," in *International Conference on Social Computing, Behavioral-Cultural Modeling, and Prediction*. Springer, 2013, pp. 341–349.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang, "Detecting malware variants via function-call graph similarity," in *2010 5th International Conference on Malicious and Unwanted Software*. IEEE, 2010, pp. 113–120.

[37] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[38] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[39] 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.

[40] 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.

[41] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[42] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in

2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020, pp. 15–29.

[43] 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.