cuAlign: Scalable Network Alignment on GPU Accelerators

Lizhi Xiang
The University of Utah
Salt Lake City, UT, USA
xianglizhi456@gmail.com

Arif Khan
Meta
Menlo Park, CA, USA
arifkhan@meta.com

S. M. Ferdous
Pacific Northwest National Lab
Richland, WA, USA
sm.ferdous@pnnl.gov

SR Aravind
Meta
Menlo Park, CA, USA
aravind_sr@outlook.com

Mahantesh Halappanavar
Pacific Northwest National Lab
Richland, WA, USA
hala@pnnl.gov

ABSTRACT

Given two graphs, the objective of network alignment is to find a one-to-one mapping of vertices in one graph (A) to vertices in the other (B), such that the number of overlaps is maximized. We say that edges $(i, j) \in A$ and $(i', j') \in B$ are overlapped if i is mapped to i' and j is mapped to j'. Network alignment is an important optimization problem with several applications in bioinformatics, computer vision and ontology matching. Since it is an NP-hard problem, efficient heuristics and scalable implementations are necessary. However, a combination of combinatorial and algebraic kernels within the network alignment algorithm poses significant hurdles for parallelization. Further, load imbalance and irregular DRAM traffic limit achievable performance on GPUs. In this work, we introduce a novel framework (cuAlign) that combines intra-network proximity using node (vertex) embedding, sparsification for computational efficiency, and belief propagation (BP) and approximate weighted matching for alignment. We demonstrate qualitative improvements up to 22% over state-of-the-art approaches. We provide a scalable implementation targeting modern GPU accelerators. Our novel approach identifies and exploits unique structural properties of the BP-based algorithm and employs code fusion to reduce data movement between different steps of the algorithm. Using a diverse set of inputs, we demonstrate up to 19× speedup for belief propagation, 3× speedup for approximate weighted matching, and 15× total, relative to a state-of-the-art multi-threaded implementation.

ACM Reference Format:
Lizhi Xiang, Arif Khan, S. M. Ferdous, SR Aravind, and Mahantesh Halappanavar. 2023. cuAlign: Scalable Network Alignment on GPU Accelerators. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3625129

1 INTRODUCTION

Network alignment is an optimization problem to find a one-to-one mapping between the vertices of a pair of graphs such that the number of overlaps is maximized. As a generalization of the graph isomorphism problem, it has several applications in domains such as bioinformatics [1, 2, 21], computer vision [8], and ontology mapping [11]. An illustration of network alignment is shown in Figure 1, and a detailed formulation is provided in §2.

The setup begins with two input graphs, $A = (V_A, E_A)$ and $B = (V_B, E_B)$. A weighted bipartite graph $L = (V_A \cup V_B, E_L)$ is computed over the vertices of A and B. Let M be a *matching* on L that maps the vertices of A to B, where matching M is a subset of edges such that no two edges in M are incident on the same vertex. An edge (i, j) in A is considered as *overlapped* with an edge (i', j') in B if the two edges (i, i') and (j, j') in L are matched in M. The *overlapped-edge* relationship between a pair of graphs is a generalization of triangular relationships in a single graph. The objective of network alignment, also referred as the *matched neighborhood consistency* [7], is to compute a matching M in L that maximizes the sum of the overlapped edges in L. In this paper, we focus on the global network alignment problem that takes a holistic view of the problem of aligning two graphs. The global network alignment finds one-to-one mapping between vertices in two netowrks whereas local network alignment finds many-to-many mappings. Although the global network alignment problem is shown to be NP-hard [12], several heuristics exist. A special case, that we term as the full network alignment problem, arises when L is a complete and weighted bipartite graph. This case arises in the absence or limited amount of prior knowledge about the input networks. Given the large search space ($|V_A| \times |V_B|$), this case is both computationally expensive and memory intensive, and has therefore attracted extensive research on methodologies that avoid full consideration [4, 11, 15, 18, 22].

We provide a brief overview of different approaches for network alignment in §3. A large body of work in the current literature focuses on the utilization of auxiliary information to prune the feasible set of matches to a computationally tractable size. Further, pruning of the feasible set has been demonstrated to improve the quality of alignment by minimizing the impact of noise in domains such as bioinformatics. However, there is a disconnect between the research on generating feasible sets with prior knowledge and the actual alignment algorithms that use the feasible set as an initial solution. A key goal of our work is to design efficient algorithms that leverage node embedding techniques to improve computational efficiency. We detail this new approach in §4. We demonstrate the efficiency of this approach, with up to 22% improvement in quality over state-of-the-art techniques, as detailed in §6.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3625129

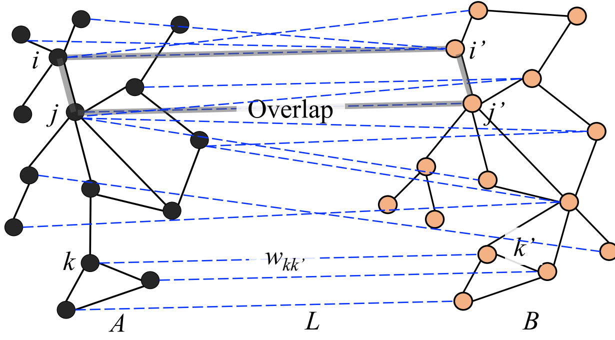

Figure 1: An illustration of the alignment problem: Find a subset of edges from L that form a weighted matching between A and B with as many overlaps as possible.

High computational costs of network alignment necessitates efficient parallel implementations. Building on the multi-threaded work of Khan *et al.,* [12] we develop efficient GPU implementation of our framework, `cuAlign`. We detail the challenges in parallelization and our approach in §5. We demonstrate excellent speedups of our implementation in §6.

The key contributions of our work are:

(1) We present a novel framework for network alignment by combining the ideas of subspace alignment, node embedding, sparsification, belief propagation and weighted matching (§4).
(2) We demonstrate the efficacy of our approach using five inputs and show up to 22% improvement in quality over the state-of-the-art methods (§6).
(3) We develop an efficient GPU implementation, `cuAlign`, by carefully addressing different challenges (§5), and demonstrate up to 15× total speed up (19× for belief propagation and 3× for approximate matching) over a CPU-only implementation (§6).

To the best of our knowledge, it is the first GPU implementation of network alignment algorithm, and a framework to combine different algorithms to achieve quality improvements that were previously not possible. Given the importance of network alignment in several domains of science, we believe that our work will benefit both researchers and practitioners in computer science and application domains.

2 PROBLEM FORMULATION

In this section, we will formulate the Network Alignment problem and define the preliminaries. Let $A = (V_A, E_A)$ and $B = (V_B, E_B)$ be two input graphs, where $V_A(V_B)$ and $E_A(E_B)$ are the set of vertices and edges of the graph $A(B)$. We assume $|V_A| = |V_B| = n$. This assumption is to keep the description of the algorithms simple and does not sacrifice the generality. We also define L as a weighted bipartite graph between the vertex sets of A and B, i.e., $L = (V_A \cup V_B, E_L, w)$. Here, w is the weight function defined on the edges of L. The graph L is constructed from A and B (Figure 1).

A matching M, of the graph L, is a subset of edges ($M \subseteq E_L$) such that no two edges in M are incident on the same vertex. The weight of the matching M is given by the summation of the weight of the matched edges. We say that an edge (i, j) in E_A is overlapped to an edge (i', j') in E_B if both (i, i') and (j, j') are in M.

Given $\alpha, \beta \geq 0$, the network alignment problem is to find a *matching* M in L that maximizes: $\alpha \times$ weight of matching + $\beta \times$ number of overlapped edges. To encode this, we adopt an integer quadratic programming framework [4, 5, 13]. Recall that w is the weight function on the edges in L. We vectorize w and use $w_{i,i'}$ to indicate an element of the vector, which is a weight on the edge $i \in V_A$ to $i' \in V_B$. Let $\mathbf{x}$ be an indicator vector over the edges of L in that same ordering of $\mathbf{w}$, such that $x_{i,i'} = 1$ if the edge is in the matching and 0 otherwise. The weight of the matching subset is then given by the inner-product, $\mathbf{x}^T\mathbf{w} = \sum_{(i,i')\in E_L} x_{i,i'} w_{i,i'}$.

Let $\mathbf{C}$ be the $(|V_A| + |V_B|) \times |E_L|$ node-edge incidence matrix of the graph L. Then the constraint $\mathbf{Cx} \leq \mathbf{e}$, where $\mathbf{e}$ is a vector of all ones, enforces that $\mathbf{x}$ corresponds to a matching. In order to compute the number of overlapped edges in A and B, we introduce the $|E_L|\times|E_L|$ matrix S, where the rows and columns of S correspond to edges in E_L. We set $\mathrm{S}_{(i,i'),(j,j')}$ to 1 if $(i, j) \in E_A$ and $(i', j') \in E_B$. Otherwise, we set $\mathrm{S}_{(i,i'),(j,j')}$ to 0. So, the number of overlapped edges in A and B is given by, $\frac{\mathbf{x}^T\mathbf{S}\mathbf{x}}{2}$. Putting these altogether, the Network Alignment problem can be formulated as the following quadratic integer program:

$$\begin{aligned} \max \quad & \alpha\mathbf{x}^T\mathbf{w} + \beta\frac{\mathbf{x}^T\mathbf{S}\mathbf{x}}{2} \\ \text{subject to} \quad & \mathbf{Cx} \leq \mathbf{e} \\ & x_{i,i'} \in \{0, 1\}, \forall (i, i') \in L. \end{aligned} \tag{1}$$

The complexity of the Formulation 1 depends on the size of L and S. Thus the construction of the bipartite graph L is one of the most important steps for our Network Alignment algorithm. One may define L as a complete bipartite graph known as *full network alignment*. But, assuming $|V_A| = |V_B| = n$, this could result in $O(n^2)$ edges in L and consequently $O(n^4)$ nonzeros in S. This would make the problem computationally infeasible, and unimplementable to GPUs due to the large memory requirement. Also, this dense graph will likely be noisy and deter the alignment quality. One of the contributions of this paper is to sparsify the complete graph such that the number of edges remains $O(n)$, thus improving both computational and quality aspects of the alignment. The sparsification is detailed in §4 and the impacts of sparsifications on runtime and quality is shown in §6.

3 RELATED WORK

Motivated by numerous applications, network alignment has been studied extensively not only from an algorithmic perspective but also application-specific perspective. The literature on network alignment can be categorized in multiple ways. Algorithms can be divided into local and global algorithms. We do not consider local algorithms in this paper and refer to a recent survey for details [2]. Global algorithms can be further classified into two broad categories: those with prior information on potential alignment [4, 13, 27], and those without prior information [15, 16, 18, 22]. Many practical techniques exploit prior information in the computational pipeline. However, they can also include scoring components that are computed without using prior information [18, 22].

The goal of solving the network alignment without prior information is to estimate some information on how well graphs A and B align in a structural sense. The techniques used in [18, 22] are

based on computing a signature score for each vertex in the two networks. They measure the distances between vertices by comparing signature scores. Other approaches use information from orthogonal projections of the adjacency matrices [9] or the graph Laplacian [14] for slightly different types of signatures. Once they compute the similarity scores between all vertices, the scores are sparsified and rounded to get a matching. It is clear that these techniques heavily rely on some form of vertex embedding in order to compute vertex signatures.

The drawback of computing vertex signatures is that the signatures are graph specific. In other words, the signatures are only meaningful within the context of a specific graph, for example, in comparing two vertices in the same graph. Comparing signatures of two vertices, such as $i \in A$ and $i' \in B$, often leads to incorrect conclusions even when those signatures are computed by using the same vertex embedding algorithm [25]. Chen *et al.* [7] investigated this vertex embedding issue and designed an *optimal transport theory* based technique, which allows vertex signatures from different graphs to be compared, and proposed a k-nearest neighbor (kNN) based network alignment algorithm, called `cone-align`. However, the kNN-based network alignment algorithm does not perform well in real-world full network cases where the objective is to maximize the number of overlapping edges.

Our work employs the vertex embedding technique of Chen *et al.* [7] as the initial input to Problem 1, and then iteratively solves it using belief propagation and graph matching algorithms. Since network alignment is NP-hard, the best practical algorithms are heuristic in nature. Therefore, including vertex embedding information helps in guiding the algorithm towards better solutions both in terms of quality and in terms computational efficiency. Thus, the novelty of our approach comes from combining two independent classes of algorithms for achieving quality that cannot be achieved by individual approaches.

A class of approximation algorithms for network alignment are based on linear programming relaxations of the quadratic assignment objective [13] combined with rounding techniques [19]. We observed that the results from belief propagation are nearly as good as these techniques and can be parallelized efficiently. There are also efficient heuristics that use a small set of aligned nodes to complete the alignment to the rest of the graph [17]. The literature on GPU implementations for network alignment is limited. Zhang *et al.* [31]] proposed a Graph Convolutional Neural Network (GCN) based network alignment algorithm, where they used GPUs for GCN kernels implemented in Tensorflow.

A closely related problem is the the problem of *subgraph isomorphism*, which determines whether a given query graph Q is isomorphic to a subgraph of a data graph G. We note that, the subgraph isomorphism problem is a special case of the Network Alignment problem developed in Formulation 1 for $\alpha = 0$ and $\beta = 1$. Several works such as [30] and [29] developed GPU implementations to solve the subgraph isomorphism problem. These efforts highlight the challenges associated with solving graph algorithms on GPUs, especially the problem of load-balancing. However, network alignment is more challenging that subgraph isomorphism problem. Moreover, our work advances the algorithms used for network alignment that achieve significantly higher quality when compared to the state-of-the-art approaches. The proposed framework is designed to subsume and benefit from advances in vertex embedding and network alignment algorithms.

4 THE FRAMEWORK

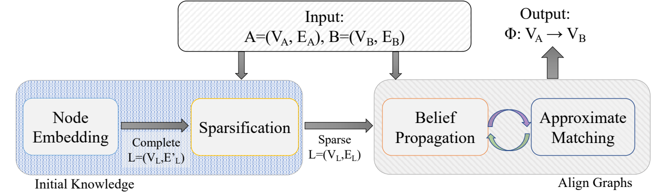

Figure 2: Computational framework of `cuAlign` that combines node embeddings, sparsification, belief propagation and approximate matching to compute high quality alignments.

We propose a novel GPU-accelerated network alignment framework, `cuAlign`, as illustrated in Figure 2, which has the following four building blocks.

(1) *Node embedding* provides a mechanism to include structural and prior information necessary to sparsify the complete graph.
(2) *Sparsification* constructs the bipartite graph L. It reduces the problem size from $O(n^4)$ to $O(n^2)$, thus enhancing the efficiency and scalability of the framework.
(3) *Belief propagation* solves a relaxed version of the Network Alignment Quadratic Program (NAQP) on sparse bipartite graph, L.
(4) *Weighted matching* provides a "good" feasible alignment using the solution obtained by the belief propagation.

As shown in Figure 2, the node embedding and sparsification are part of initialization. These two steps together create the bipartite graph L, and need to be executed only once. The two other blocks run iteratively until a stopping criterion is met. We observe that these two blocks also constitute the compute-heavy portion of the framework. We therefore employ GPU algorithms to scale these components (detailed in §5).

4.1 Embeddings and Sparsification

The goal of *embedding* the vertices of A and B, is to represent each vertex as a vector such that these vectors capture the relationship between vertices within the context of the two graphs. A proper embedding enables us to query the distance or similarity measure of a vertex of A to a vertex of B and vice versa, which is necessary to sparsify the complete bipartite graph.

Since we will employ the embedding to define a similarity or distance measure of the inter-edges of A and B (i.e., the edges of L), the embedding needs to incorporate the joint structural and a priori information of both graphs A and B. To facilitate this, we propose to develop the node embedding in two steps. First, we use standard proximity-based embedding methods to encode the proximity and structural relation of the individual graphs. Embedding based on node proximity preserves the neighborhood relationships, i.e., embeddings of the neighboring nodes tend to be *close* to each other for some distance metric [6, 25]. Next, we align the subspaces of the individual embedding vectors to generate the final embedding. Let

$Y_1, Y_2 \in \mathbb{R}^{n\times d}$ be the node embedding generated from a proximity-based embedding method for graphs A and B respectively, where d is the size of the embedding. Also assume that P is a permutation on Y_1 and Q is a linear transform on Y_2. We solve the following optimization problem of finding a Q that aligns the sub spaces defined by Y_1 and Y_2 to get our final embedding vectors Y_A and Y_B, resp.:

$$\min_{Q\in\mathbb{O}^d} \min_{P\in\mathbb{P}^n} ||Y_1Q - PY_2||_2^2. \quad (2)$$

Equation 2, can be solved iteratively by using singular value decomposition (SVD) and Sinkhorn optimization using super-linear approximations. A detailed explanation of can be found in [7].

With Y_A and Y_B, we can now construct the weighted bipartite graph $L = (V_A \cup V_B, E_L)$. One option could be to create the complete graph where the weight of an edge is some distance measure (such as Euclidean or Cosine distance) w.r.t the embeddings of the two endpoints. But the complete graph is often noisy and presents computational, memory and scalability challenges to the overall algorithm. We construct a sparse bipartite graph L by employing the popular k-Nearest Neighbor (kNN) technique. For each vertex of L, we choose k least-weighted incident edges, where $k > 0$. Since k is a constant, the number of edges of L is reduced to $O(n)$. Further, kNN is widely used to construct graphs in graph based machine learning [28] and has theoretical guarantees for quality of representations [23, 26]. The embedding and sparsification steps initialize the framework by creating the bipartite graph as shown in Algorithm 1. Here, lines from 2 through 6 generate the embedding vectors, lines 7 to 11 create the complete bipartite graph, and the rest of the algorithm sparsifies the complete graph.

Algorithm 1: Create Bipartite Graph, L

```
   Input: A(V_A, E_A), B(V_B, E_B)
   Output: L(V_A, V_B, E_L)
1  L ⟵ Bipartite(V_A, V_B, E_L = ∅)
2  Y_1 ⟵ proximity_embed(A)
3  Y_2 ⟵ proximity_embed(B)
4  Q ⟵ solution for Equation 2
5  Y_A ⟵ Y_1 Q
6  Y_B ⟵ Y_2
7  for u ∈ V_A do
8  |  for v ∈ V_B do
9  |  |  s ⟵ cosine_similarity(Y_A(u), Y_B(v))
10 |  |  E_L ⟵ E_L ∪ (u, v, s)
11 V_L ⟵ V_A ∪ V_B
12 for u ∈ V_L do
13 |  N_u ⟵ kNN(u)
14 |  E_L ⟵ E_L \ N_u
```

4.2 Belief Propagation

Quadratic Programs (QP), such as the problem represented by Equation 1, are NP-hard to solve, and therefore, need heuristics or approximate algorithms for solution. A popular heuristic to solve QPs is to use message passing procedure known as Belief Propagation (BP). We treat the quadratic program in Equation 1 as finding the maximum posterior estimation of the following distribution:

$$P(\mathbf{x}) = \frac{1}{Z} \exp(\alpha \mathbf{x}^T \mathbf{w} + \beta/2\mathbf{x}^T \mathbf{S}\mathbf{x}) \ \mathrm{Ind}[\mathbf{C}\mathbf{x} \le \mathbf{e}], \quad (3)$$

Algorithm 2: Belief Propagation

```
   Input: L(V_L, E_L), S ∈ R^{|E_L|×|E_L|}
   Output: bestM
1  p ⟵ 0
2  S^c ⟵ 0 where S^c ∈ R^{|E_L|×|E_L|}
3  S^p ⟵ 0 where S^p ∈ R^{|E_L|×|E_L|}
4  for i ∈ {1 ... |E_L|} in parallel do
5  |  {y^c(i), y^p(i), z^c(i), z^p(i), d^c(i), d^p(i)} ⟵ 0
6  bestScore = 0
7  bestM ⟵ ∅
8  for p ∈ {1 ... n_iter} do
9  |  F = bound_{0,β}[βS + S^{pT}]
10 |  d^c = αw + Fe
11 |  y^c = d^c − othermaxcol(z^p)
12 |  z^c = d^c − othermaxrow(y^p)
13 |  S^c = diag(y^c + z^c − d^c)S − F
14 |  y^p = γ^k y^c + (1 − γ^k) y^p
15 |  z^p = γ^k z^c + (1 − γ^k) z^p
16 |  S^p = γ^k S^c + (1 − γ^k) S^p
17 |  E_L = y^c
18 |  M_y = Approximate_Weighted_Matching(L)
19 |  E_L = z^c
20 |  M_z = Approximate_Weighted_Matching(L)
21 |  score, M = max(evaluate(M_y, S), evaluate(M_z, S))
22 |  if score > bestScore then
23 |  |  bestScore = score
24 |  |  bestM = M
```

where Z is an unknown normalizing constant and Ind() is the indicator function. We observe that our intended QP is in fact a discrete (binary to be exact) quadratic problem, where the goal is to generate a matching on graph L. $P(\mathbf{x})$ relaxes the problem to real valued one where we aim to construct a probability distribution. We employ belief propagation to construct $P(\mathbf{x})$, which is later transformed to a binary solution. This step is known as rounding, which is achieved by computing a half-approximate weighted matching. We apply the relaxation through BP and rounding through matching iteratively until a stopping criterion is met. The final matching is our desired solution for network alignment.

Here we provide a brief overview of the belief propagation (BP) method. For a detailed description of the algorithm, we refer to the work of Bayati *et al.* [3, 4] and Khan *et al.* [12]. Recall that our objective for the NAQP formulated in Problem 1 constitutes maximizing a linear combination of weights of the alignment and number of overlaps. Formally, We say two edges $e(i, i'), e'(j, j') \in E_L$ overlaps if $(i, j) \in E_A$ and $(i', j') \in E_B$ shown in Figure 1. Intuitively, an overlap captures the conserved relationship between vertices in two graphs, and considered to be a generalization of triangular (friends of friends) relationship among vertices in single graph. This overlapping information is encoded in a matrix $S \in \{0,1\}^{|E_L|\times|E_L|}$, where the value represents if the two edges overlap or not. Computing S is embarrassingly parallel (Algorithm 3).

Next, we apply BP technique described in Algorithm 2 by iteratively updating the two weight vectors $y^{(p)}, z^{(p)}$ and a weight matrix $S^{(p)}$. The first vector represents the log-likelihood of each edge in L occurring in the final matching given that we want each

Algorithm 3: Create Overlap Matrix S

```
   Input: A(V_A, E_A), B(V_B, E_B), L(V_L, E_L, W)
   Output: S ∈ {0,1}^{|E_L|×|E_L|}
1  for e(v,u) ∈ E_L in parallel do
2      processed(e) ⟵ true
3      for each neighbor u' of u ∈ V_A do
4          for each neighbor v' of v ∈ V_B do
5              if e'(u',v') ∈ E_L then
6                  S[e][e'] = 1
7              else
8                  S[e][e'] = 0
```

vertex in graph A matched to at most one vertex in graph B. The second vector represents the same aspect, but given that we want each vertex in graph B to match to at most one vertex in graph A. The weight matrix represents the log-likelihood of overlapped edges appearing in the solution.

At each iteration, belief propagation rules encode the logic of a local, greedy agent that attempts to determine its own likelihood, given the likelihoods of its neighbors computed in previous iteration (Lines 9-12). Because of this interpretation, the weight vectors are usually called messages as they communicate the "beliefs" of each "agent" (Line 13). In this particular problem, the neighborhood of an agent associated with an edge, $e \in L$ represents the $e' \in L$ that overlaps with e. The weight vectors or the messages do not generally converge [4], and thus, the iteration is artificially damped to enforce convergence (Lines 14-16). After each update of the messages, we round the messages by using a half-approximate bipartite maximum weighted matching (Lines 17-20). For matching we use the parallel algorithm described in [12], detailed in §4.3. Finally, we evaluate the objective function (Lines 21-24).

All the matrix and vector operations (Lines 9-16) in Algorithm 2 are element-wise operations, and are therefore embarrassingly parallel. An important observation to make is that the overlap matrices S, S^c, S^P are structurally symmetric. Since all the belief propagation computations are based on the overlap matrices, sparse data structures for vectors and matrices remain fixed. Hence, we precompute the sparsity structures of the matrices and vectors once. Only the values change during the course of the execution. This feature of the algorithm enabled us to efficiently allocate and assign "agents" to GPU warps to improve performance discussed in §5. The computation in function *othermaxrow* is as follows. Given a row, replace all non-zeros in that row with the maximum value for the row; except, for the element that is the maximum value, replace it with the second largest value. The *othermaxcol* function works similarly columns instead of rows. Since belief propagation has no natural stopping criteria, we run it for a fixed number of iterations, and take the best solution we find in any step of the computation.

4.3 Half-approximate Weighted Matching

While efficient serial algorithms for computing optimal matching exist, they have limited concurrency, and therefore we consider approximation algorithms that are not only amenable to parallel implementations but also compute high quality solutions. We consider approximation algorithms that guarantee half-approximate solutions with respect to optimal matchings. However, in practice, they provide solutions that are nearly optimal. Khan *et al.,* [12] used the *locally dominant algorithm*, which was first presented by Preis [24]. An edge that is at least as heavy as all other edges incident on its two end-points is a *locally dominant* edge. For details we refer you to Halappanavar *et al.* [10].

The locally dominant algorithm has two phases. In Phase 1, for each vertex u, the algorithm chooses the heaviest neighbor, v, in terms of the corresponding edge weight, $w(u, v)$ as the candidate mate. If the candidate mate, v also chooses the u as it's candidate mate then we call the edge, $w(u, v)$, a locally dominant edge and add it to the matching M. We match the endpoints of all the locally dominant edges, and the matched vertices are added to a queue (Q_C). In Phase 2 of the algorithm, we iterate over the queue until no new edges get matched. During each iteration, we process vertices matched in previous iterations and add new vertices to the queue that become eligible by checking whether any of their unmatched neighbors point to them. If so, those neighbors will have to find new candidates for matching. The algorithm ends when no new candidates are found. We use compressed sparse row (CSR) data structure to store graphs in memory and use CUDA programming model for parallelization. We use two queues (Q_C and Q_N) so that while we efficiently process (read) the vertices matched in the previous iteration in Q_C, we can enqueue (write) vertices matched in the current iteration in Q_N, and thus reducing contention.

The belief propagation algorithm (Algorithm 2) has regular memory access patterns since the matrix/vector sparsity structure does not change over during execution. In contrast, matching has irregular memory accesses. Since both of these algorithms are executed in tandem, it is difficult to design and implement a performant code. Thus, a major contribution of our work is the efficient design to exploit the underlying properties of the respective algorithms targeting GPU implementations as discussed next.

5 PARALLEL (GPU) IMPLEMENTATIONS

While the Network Alignment framework described in §4 is inherently parallel in CPU, there are significant challenges to implementing and scaling it in GPUs. These challenges arise primarily because of the irregular computations. Factors such as (*i*) Load-imbalance and thread idling, (*ii*) high data movement and low data reuse, and (*iii*) memory access efficiency can significantly affect the performance of the GPU implementation.

Load imbalance is a critical factor that affects the performance of the network alignment algorithm. Directly porting the CPU algorithm to GPUs will likely result in a significant load imbalance. Both belief propagation and half-approximate matching require accessing the neighborhood information. The number of neighbors per node can significantly vary, especially for power-law graphs. Assigning a single thread to process a single vertex can result in substantial warp divergence and intra- and inter-thread-block load imbalance. Another strategy is assigning one warp to process each vertex's neighbors collectively. This approach helps to achieve good memory-access efficiency (explained in the next paragraph). In addition to the previous challenges, this strategy can cause thread-idling (i.e., some of the threads do not have any work) if the number of neighbors of a given vertex is less than the warp size. This approach also suffers from intra- and inter-thread-block load imbalances.

We observe that the sparsity structure of the matrices used in Algorithm 2 remain fixed throughout the iteration. It means the

graph structures associated with these matrices also remain constant. This allows us to employ *binning* to solve the load imbalance problem. We simply group the vertices based on their degrees once and reuse this grouping throughout the algorithm. This approach allows us to achieve good load balance within each bin as all the threads within a bin have similar amount of work.

Listing 1: Fused kernel corresponding to line 9 in Algorithm 2

```
1  //Iterate over the rows of the matrix
2  for (int i=0; i<size; ++i) {
3   int h=ic[i]-1;
4   int t=ic[i+1]-1;
5   double sum = 0;
6   //Iterate over the columns
7   for(int j=h;j<t;j++) {
8     //Compute the value in F
9     double val = beta+s^p[perm[j]];
10    if(val<0)
11       val=0;
12     else {
13       if(val>beta)
14          val=beta;   }
15     F[j] = val;
16     sum =+ val; //reduction }
17   dc[i]=sum; }
```

Binning also allows us to determine the number of threads we should assign to process each vertex. For example, if a bin contains nodes that have approximately 64 neighbors, then it is better to assign each thread block with 64 threads to process each vertex. However, when the thread block size is less than 64, say 16, even if we only request 16 threads per thread block, the hardware will assign 32 threads per thread block as the hardware can only schedule work in terms of warps (a group of 32 threads). Hence, in the latter case, 16 threads will be idle. In this case, it is better to assign one thread block of size 32 and make it process two vertices. We use the term *virtual warp* to represent the number of threads assigned to process each node. We could always assign a virtual warp of size 16 to process all bins. However, this strategy will result in unnecessary uncoalesced memory access for high-degree nodes. Hence we choose the virtual warp size based on bin size. Since the virtual warp size should be a divisor (or multiple) of warp size to avoid thread idling, we only use the virtual warp sizes from the set $\{8, 16, 32, 64, 128, 256, 512\}$. For example, if the nodes in a given bin have close to 16 neighbors, assigning 16 threads to process each node will reduce tread idling. Hence for bins with vertices which has less than or equal to 16 neighbors, we split a warp into virtual warps (Figure 3) and assign each virtual warp to process each node. For nodes in the remaining bins, we assign one warp per vertex. However, launching one kernel per bin might lead to under utilization at the streaming multi-processor level. For example, a bin consists only of two nodes. Launching this kernel on an A100 GPU will use at most two out of 84 available streaming multi-processors. Therefore, we process bins in parallel by using CUDA streams to launch multiple bins in parallel.

Data movement and data reuse is arguably the most important parameter that affects performance of GPU algorithms. Our approach performs high-level *code fusion* and GPU architecture-aware low-level optimizations to improve data reuse. Several steps in Algorithm 2 share common data-structures. For example, consider line 9, which produces the F matrix, and line 10, which consumes it. GPU caches are relatively small, and due to cache eviction, it is unlikely to get reuse of the F matrix between these two lines. Code fusion optimization shown in Listing 1 reduces the total number of reads from line 9 and 10 to (2× #non-zeros in S) to (#non-zeros in S) resulting 50% reduction in data movement. Our approach also utilized low-level memory optimizations. For example, consider the loop in line 7 in Listing 1. Since we need to distribute this loop among multiple threads in a warp to achieve a good load balance, we must reduce the *sum* variable across multiple threads. We use warp reductions instead of global reductions to improve performance. Warp reductions perform reductions by only reading and writing at the register level. Furthermore, we also utilize shared-memory to achieve high data reuses. For example, in Algorithm 3 each neighbor of a given vertex is accessed multiple times. Hence we keep them in shared memory.

Memory access efficiency is another factor that affects the achievable performance in GPUs. Global memory coalescing is a crucial factor that determines memory access efficiency. In GPUs, the memory request from a group of 32 threads called a warp is aggregated by the memory controller into multiple transactions. If threads in a warp access contiguous memory, only one memory transaction is required; otherwise, multiple transactions are required which degrade the performance. There are several regions in Algorithm 2, such as lines 14-16, where we access the neighbors of a given node. If we assign one thread per vertex, continuous threads will access neighbors of different vertices resulting in uncoalesced access. We could solve this by assigning one warp per vertex, in which case contiguous threads will access the adjacent neighbors of a given vertex, resulting in coalesced access. However, this will result in resource under-utilization, as explained earlier.

Our approach for binning and virtual warping helps improve memory access efficiency without causing resource under-utilization. Note that our binning strategy is based on the number of neighbors of each node. If the number of neighbors is greater than the warp size, we assign one full warp to process it and thereby achieve full coalescing. However, if the number of neighbors is less than warp size, the number of threads assigned to process that vertex is less. Let us consider a bin where the number of neighbors is at most eight, and that we assign a virtual warp of size eight to process each vertex in this bin. The corresponding GPU code is shown in Listing 2. The contiguous eight threads can access the continuous eight neighbors, which is the maximum number of neighbors possible for nodes in the given bin and thus achieves maximum possible coalescing. The remaining threads in the warp process other vertices. While this causes partial uncoalescing, it avoids thread idling.

6 EXPERIMENTAL RESULTS

We now briefly describe the experimental setup, followed by the qualitative assessment of `cuAlign` compared to state-of-the-art method, and the relative GPU speedups.

Listing 2: GPU kernel design for kernels corresponding to line 13 in Algorithm 2

```
1
2 unsigned int warp_id = threadIdx.x / 8;
3 unsigned int lane_id = threadIdx.x % 8;
4 unsigned int global_id = blockIdx.x * 4 +
      warp_id;
5 for(unsigned int id=global_id;id<size;id+=(
      gridDim.x * 4)){
6     int i = rows[id];
7     vvc[i]=yc[i]+zc[i]-alpha*wi[i]-dc[i];
8     int s=ic[i]-1;
9     int t=ic[i+1]-1;
10    if(lane_id+s<t){
11        sp[lane_id+s] = vvc[i] - F[lane_id+s
              ];
12    }
13 }
```

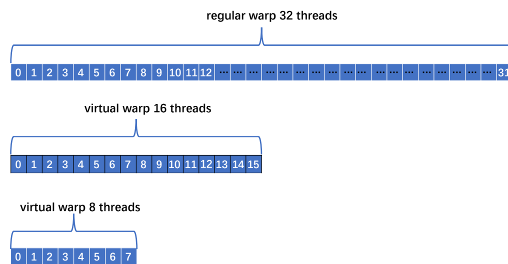

Figure 3: Conceptual view of virtual warp

6.1 Experimental Setup

Our experimental setup consists of an AMD EPYC 7702P 64-Core Processor (512 GB RAM) CPUs running Ubuntu 20.04 LTS. For GPU evaluation, we used an Nvidia Ampere A100 machine(108SMs, 40 GB global memory) with CUDA v11.1.

Network	Vertices	Edges
fly_Y2H1	7,094	23,356
fly_PHY1	7,885	36,271
human_Y2H1	9,996	39,984
Synthetic_4000	4,000	11,996
Synthetic_8000	8,000	63,977

Table 1: Input graphs used for empirical evaluation.

We evaluated cuAlign using three real-world graphs and two synthetic graphs, listed in Table 1. The real-world graphs are common biological inputs in network alignment studies [7], and the synthetic graphs are generated by methods described in [7]. For each graph A, we created a matching pair $B = P(A)$, by applying a random permutation P on the vertices of A. The permutation P is the true node alignment (ground truth). We input A and B to cuAlignand compare the computed alignment M against the true alignment P to compute the alignment score. We use the state-of-the-art scoring metric called $NCV - GS^3$, which measures how well the edges are conserved between two networks (i.e., overlaps) normalized by the sizes of the networks. A detailed explanation of various alignment scores are provided in [18, 20].

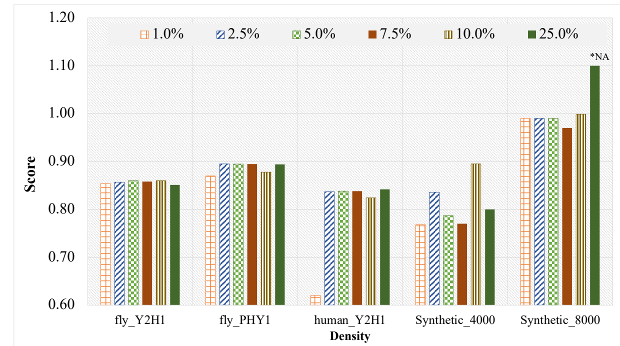

Figure 4: Quality of solution for different levels of density on the CPU platform. Computation did not finish for Synthetic_8000 with 25% density, which we represent with a large value (> 1)**.**

6.2 Qualitative Assessment

For qualitative assessment of cuAlign, we compare the solutions with cone-align, a state-of-the-art algorithm [7]. Since cuAlign uses the node embeddings from cone-align, the improvements in quality come from the optimization components through iterative execution of belief propagation and weighted matching techniques described in §4. First, we investigate the impact of sparsification on quality of cuAlign. In Figure 4, we consider the real world protein-protein interaction networks and observe that the quality goes down with the increase in density. We note that $density = (100 - sparsity)$; a complete graph has 100% density. The reason for sparsification improving the quality of solution is that the real world data is often noisy, therefore, removing low weight edges helps our heuristic framework to find better solution. In literature, alignment scores greater than 80% are considered as good solutions [20] and we observe that cuAlign achieves high quality solutions with ≤ 10% of total edges in L.

Next, we show the impact of sparsification on the compute time. Since, feasible combinations for alignment becomes prohibitively large in the complete network case (i.e., 100% density) and in Figure 5, we show how the run time increases with the increase in density. Note, that the Y-axis in Figure 5 is logarithmic. As expected, the run time increases significantly as we increase the density of the graph. We conclude that sparsification not only improves the quality of solution but also reduces the compute time significantly.

Finally, we demonstrate a significant contribution of our work in improving the quality of solutions over the state-of-the-art algorithm, cone-align. We pick 1% and 2.5% level of sparsification and observe that our framework finds better alignments for both sparsification levels. The improvement is up to 22% in terms of alignment score relative to the test set (Figure 6).

Problems	BP-CPU	BP-GPU	Speedup (BP)	Match-CPU	Match-GPU	Speedup (Match)	Total Speedup
fly_Y2H1	400	21	**19.05**	19	8.3	**2.29**	**14.03**
fly_PHY1	682	48	**14.21**	37	12.8	**2.89**	**11.82**
human_Y2H1	1384	79	**17.52**	51	19.4	**2.63**	**14.59**
Synthetic-4000	29	5.75	**5.04**	6	2.25	**2.67**	**4.37**
Synthetic-8000	627	57	**11.00**	52	17.9	**2.91**	**9.06**

Table 2: Run time improvement of Belief Propagation (BP) and Matching steps using GPU accelerator.

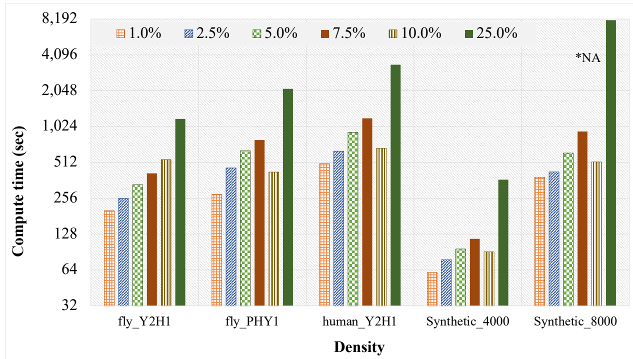

Figure 5: Compute time in seconds ($\log_2$ scale) for different levels of density on the CPU platform. Computation did not finish for Synthetic_8000 with 25% density, which we represent with a large value.

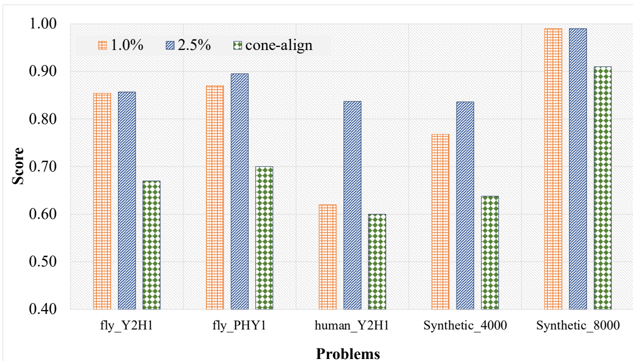

Figure 6: Quality comparison: `cuAlign` vs. `coneAlign`

6.3 Relative Performance

Recall that `cuAlign` augments the `cone-align` by combining the BP and matching techniques, presented by the authors in [12]. Both BP and matching are computationally heavy. `cuAlign` computes higher quality solutions over the state-of-the art `cone-align` by using these additional computation. Therefore, efficient implementation of that additional computation involving belief propagation and matching becomes critical. We provide the performance of GPU implementation of the belief propagation and matching w.r.t., multithreaded implementation of the same in [12]. The breakdowns are summarized in Table 2. Our GPU implementation is up to 19× faster for belief propagation and 2.91× faster for matching, resulting in up to 15× speedup for the entire optimization phase.

Finally, with the GPU implementation, we show in Figure 7 that the `cuAlign` achieves high quality solution without noticeable degrade w.r.t. `cone-align`. We emphasize to the fact that `cuAlign` is rather a general framework since one can easily switch the node embedding as well as sparsification algorithms based on the domain knowledge and obtain domain specific solution.

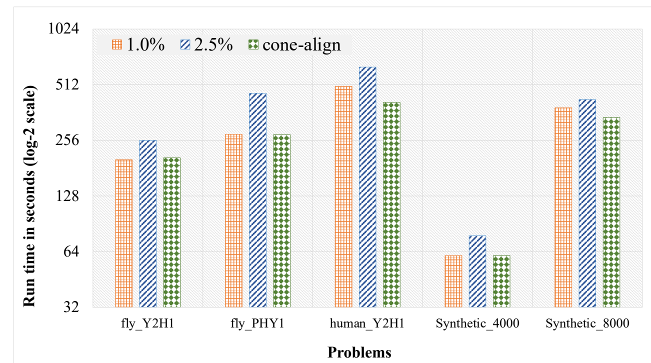

Figure 7: Run time comparison: `cuAlign-GPU` vs. `coneAlign` on the GPU platform

7 CONCLUSIONS AND FUTURE WORK

Network alignment is an important graph problem with numerous applications in diverse domains of science. We have made two key contributions in this paper. The first contribution comes from the design of a general framework that brings together methods for node embedding and sparsification with optimization approaches. The second contribution comes from scaling the optimization methods on modern GPU accelerators that provide significant computing resources, but are hard to program. Our framework provided improvements in quality of solution by up to 22%. We implement computationally heavy steps of belief propagation (BP) and graph matching on GPUs. We show that GPU implementation improves the BP step up to 19× and the matching step up to 3×, with up to 15× total speedups for the optimization phase.

We plan to explore the effectiveness of deep learning based node embeddings for structural learning in the first phase of *cuAlign*, along with new approaches for sparsification. We will also explore distributed multi-GPU implementations of belief propagation and weighted matching algorithms.

ACKNOWLEDGMENTS

The research is supported in part by the U.S. DOE Exascale Computing Project's (ECP) (17-SC-20-SC) ExaGraph codesign center and

Laboratory Directed Research and Development Program at Pacific Northwest National Laboratory (PNNL). S M Ferdous is grateful for the support of the Linus Pauling Distinguished Postdoctoral Fellowship program.

REFERENCES

[1] Ahmet E Aladağ and Cesim Erten. 2013. SPINAL: scalable protein interaction network alignment. *Bioinformatics* 29, 7 (2013), 917–924.

[2] Nir Atias and Roded Sharan. 2012. Comparative analysis of protein networks: hard problems, practical solutions. *Commun. ACM* 55, 5 (2012), 88–97.

[3] Mohsen Bayati, Margot Gerritsen, David F Gleich, Amin Saberi, and Ying Wang. 2009. Algorithms for large, sparse network alignment problems. In *2009 Ninth IEEE International Conference on Data Mining*. IEEE, 705–710.

[4] Mohsen Bayati, David F. Gleich, Amin Saberi, and Ying Wang. 2013. Message-Passing Algorithms for Sparse Network Alignment. *ACM Trans. Knowl. Discov. Data* 7, 1, Article 3 (mar 2013), 31 pages.

[5] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. 2012. *Assignment problems: revised reprint.* SIAM.

[6] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[7] Xiyuan Chen, Mark Heimann, Fatemeh Vahedian, and Danai Koutra. 2020. Cone-align: Consistent network alignment with proximity-preserving node embedding. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management.* 1985–1988.

[8] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence* 18, 03 (2004), 265–298.

[9] Catherine Fraikin, Yurii Nesterov, and Paul Van Dooren. 2008. A gradient-type algorithm optimizing the coupling between matrices. *Linear Algebra Appl.* 429, 5-6 (2008), 1229–1242.

[10] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. 2012. Approximate weighted matching on emerging manycore and multithreaded architectures. *The International Journal of High Performance Computing Applications* 26, 4 (2012), 413–430.

[11] Wei Hu, Yuzhong Qu, and Gong Cheng. 2008. Matching large ontologies: A divide-and-conquer approach. *Data & Knowledge Engineering* 67, 1 (2008), 140–160.

[12] Arif M Khan, David F Gleich, Alex Pothen, and Mahantesh Halappanavar. 2012. A multithreaded algorithm for network alignment via approximate matching. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[13] Gunnar W Klau. 2009. A new graph-based method for pairwise global network alignment. *BMC bioinformatics* 10, 1 (2009), 1–9.

[14] David Knossow, Avinash Sharma, Diana Mateus, and Radu Horaud. 2009. Inexact matching of large and sparse graphs using laplacian eigenvectors. In *International workshop on graph-based representations in pattern recognition*. Springer, 144–153.

[15] Giorgos Kollias, Shahin Mohammadi, and Ananth Grama. 2011. Network similarity decomposition (nsd): A fast and scalable approach to network alignment. *IEEE Transactions on Knowledge and Data Engineering* 24, 12 (2011), 2232–2243.

[16] Giorgos Kollias, Madan Sathe, Olaf Schenk, and Ananth Grama. 2014. Fast parallel algorithms for graph similarity and matching. *J. Parallel and Distrib. Comput.* 74, 5 (2014), 2400–2410.

[17] Nitish Korula and Silvio Lattanzi. 2013. An efficient reconciliation algorithm for social networks. *arXiv preprint arXiv:1307.1690* (2013).

[18] Oleksii Kuchaiev, Tijana Milenković, Vesna Memišević, Wayne Hayes, and Nataša Pržulj. 2010. Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface* 7, 50 (2010), 1341–1354.

[19] Konstantin Makarychev, Rajsekar Manokaran, and Maxim Sviridenko. 2010. Maximum quadratic assignment problem: Reduction from maximum label cover and lp-based approximation algorithm. In *International Colloquium on Automata, Languages, and Programming*. Springer, 594–604.

[20] Lei Meng, Aaron Striegel, and Tijana Milenković. 2016. Local versus global biological network alignment. *Bioinformatics* 32, 20 (2016), 3155–3164.

[21] Misael Mongiovì and Roded Sharan. 2013. Global alignment of protein–protein interaction networks. In *Data Mining for Systems Biology*. Springer, 21–34.

[22] Rob Patro and Carl Kingsford. 2012. Global network alignment using multiscale spectral signatures. *Bioinformatics* 28, 23 (2012), 3105–3114.

[23] Alex Pothen, S. M. Ferdous, and Fredrik Manne. 2019. Approximation algorithms in combinatorial scientific computing. *Acta Numerica* 28 (2019), 541–633.

[24] Robert Preis. 1999. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 259–269.

[25] Ryan A Rossi, Di Jin, Sungchul Kim, Nesreen K Ahmed, Danai Koutra, and John Boaz Lee. 2020. On proximity and structural role-based embeddings in networks: Misconceptions, techniques, and applications. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14, 5 (2020), 1–37.

[26] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. 2011. Local Graph Sparsification for Scalable Clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. ACM, New York, NY, USA, 721–732. https://doi.org/10.1145/1989323.1989399

[27] Rohit Singh, Jinbo Xu, and Bonnie Berger. 2007. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Annual international conference on research in computational molecular biology*. Springer, 16–31.

[28] Amarnag Subramanya and Partha Pratim Talukdar. 2014. Graph-Based Semi-Supervised Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8, 4 (2014), 1–125. https://doi.org/10.2200/S00590ED1V01Y201408AIM029 arXiv:https://doi.org/10.2200/S00590ED1V01Y201408AIM029

[29] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[30] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang. 2020. GSI: GPU-friendly Subgraph Isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1249–1260. https://doi.org/10.1109/ICDE48307.2020.00112

[31] Si Zhang, Hanghang Tong, Jiejun Xu, Yifan Hu, and Ross Maciejewski. 2019. Origin: Non-rigid network alignment. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 998–1007.