# Network Motif Discovery: A GPU Approach

Wenqing Lin[†‡], Xiaokui Xiao[‡], Xing Xie[§], Xiao-Li Li[†]
[†]A*STAR, Singapore, {linw, xlli}@i2r.a-star.edu.sg
[‡]NTU, Singapore, xkxiao@ntu.edu.sg
[§]Microsoft Research Asia, xing.xie@microsoft.com

***Abstract*— The identification of network motifs has important applications in numerous domains, such as pattern detection in biological networks and graph analysis in digital circuits. However, mining network motifs is computationally challenging, as it requires enumerating subgraphs from a real-life graph, and computing the frequency of each subgraph in a large number of random graphs. In particular, existing solutions often require days to derive network motifs from biological networks with only a few thousand vertices. To address this problem, this paper presents a novel study on network motif discovery using Graphical Processing Units (GPUs). The basic idea is to employ GPUs to parallelize a large number of subgraph matching tasks in computing subgraph frequencies from random graphs, so as to reduce the overall computation time of network motif discovery. We explore the design space of GPU-based subgraph matching algorithms, with careful analysis of several crucial factors that affect the performance of GPU programs. Based on our analysis, we develop a GPU-based solution that (i) considerably differs from existing CPU-based methods, and (ii) exploits the strengths of GPUs in terms of parallelism while mitigating their limitations in terms of the computation power per GPU core. With extensive experiments on a variety of biological networks, we show that our solution is up to two orders of magnitude faster than the best CPU-based approach, and is around $20$ times more cost-effective than the latter, when taking into account the monetary costs of the CPU and GPUs used.**

## I. Introduction

Given a graph $G$, a *network motif* in $G$ is a subgraph $g$ of $G$, such that $g$ appears much more frequently in $G$ than in *random graphs* whose degree distributions are similar to that of $G$ [1]. The identification of network motifs finds important applications in numerous domains. For example, network motifs are used (i) in system biology to predict protein interactions in biological networks and discover functional sub-units [2], (ii) in electronic engineering to understand the characteristics of circuits [3], and (iii) in brain science to study the functionalities of brain networks [4].

Numerous techniques [5]–[13] have been proposed to identify network motifs from sizable graphs. Roughly speaking, all existing techniques adopt a common two-phase framework as follows:

• *Subgraph Enumeration*: Given a graph $G$ and a parameter $k$, enumerate the subgraphs $g$ of $G$ with $k$ vertices each;
• *Frequency Estimation*: For each subgraph $g$ identified in the subgraph enumeration phase, estimate its expected frequency (i.e., expected number of occurrences) in a random graph with identical degree distribution to $G$; if $g$'s frequency in $G$ is significantly higher than the expected frequency in a random graph, then return $g$ as a motif.

The above framework, albeit conceptually simple, is difficult to implement efficiently due to the significant computation overhead incurred by the frequency estimation phase. Specifically, to estimate the expected frequency of a subgraph $g$ in a random graph, the standard approach [10] is to generate a sizable number $r$ of random graphs (e.g., $r = 1000$), and then take the average frequency of $g$ in those graphs as an estimation. To compute the frequency of $g$ in a random graph $G'$, however, we need to derive the number of subgraphs of $G$ that are isomorphic to $g$ – this requires a large number of *subgraph isomorphism tests* [14], which are known to be computationally expensive. The high costs of subgraph isomorphism tests, coupled with the large number $r$ of random graphs, render the frequency estimation phase a computational challenge. Existing techniques attempt to resolve this issue by improving the efficiency of subgraph isomorphism tests, but only achieve limited success. As shown in Section VII, even the state-of-the-art solutions require days to derive network motifs from graphs with only a few thousand vertices.

Motivated by the deficiency of existing work, we present an in-depth study on efficient solutions for network motif discovery. Instead of focusing on the efficiency of individual subgraph isomorphism tests, we propose to utilize Graphics Processing Units (GPUs) to parallelize a large number of isomorphism tests, in order to reduce the computation time of the frequency estimation phase. This idea is intuitive, and yet, it presents a research challenge since there is no existing algorithm for testing subgraph isomorphisms on GPUs. Furthermore, as shown in Section III, existing CPU-based algorithms for subgraph isomorphism tests cannot be translated into efficient solutions on GPUs, as the characteristics of GPUs make them inherently unsuitable for several key procedures used in CPU-based algorithms.

To address above challenges, we propose a novel subgraph matching technique tailored for GPUs. Our technique adopts the filter-refinement paradigm, and is developed with careful considerations of three crucial factors that affect the performance of GPU programs, namely, load balancing on GPU cores, branch divergences in GPU codes, and memory access patterns on the GPU (see Section II-B). Based on those considerations, we make design choices that (i) drastically differ from existing CPU-based methods, but (ii) lead to superior efficiency on GPUs. In addition, our technique incorporates several optimization methods that considerably improve scalability and efficiency. We experimentally evaluate our solution against the state-of-the-art CPU-based methods on a variety of biological networks using two machines, each of which has a 500-dollar CPU, a low-end 300-dollar GPU, and a high-end 2700-dollar GPU. We show that, when running with the high-end (resp. low-end) GPU, our solution outperforms the best CPU-based approach by two orders of magnitude (resp. one order of magnitude) in terms of computation efficiency. Furthermore, the *per-dollar* performance of our solution is roughly 20 times higher than that of the best CPU-based method. This note only establishes the superiority of our solution, but also demonstrates that, for network motif discovery, GPU-based methods are much more cost-effective than CPU-based ones.

In summary, we present the *first* study on GPU-based algorithms for network motif discovery, and make the following contributions. First, we analyze the deficiency of the existing CPU-based methods, and pinpoint the reasons that they cannot be translated into efficient algorithms on GPUs. Based on our analysis, we propose a novel solution that exploits the strengths of GPUs in terms of parallelism, and mitigates their limitations in terms of the computation power per GPU core. (Sections III, IV, and V)

Second, we develop three optimization techniques that improve the scalability of our solution, avoid under-utilization of GPU, and eliminate redundant computation. Together, those optimizations reduce the computation cost of our solution by $75\%$, and enable our solution to handle graphs that are ten times larger than those studied in previous work. (Section VI)

Finally, we empirically compare our solution against the state-of-the-art CPU-based methods, using the largest datasets ever tested in the literature of network motif discovery. We show that, even with a low-end GPU, our solution runs 10 times faster than the best CPU-based method, and this performance gap is further widen by 10-fold when a high-end GPU is used. Furthermore, our solution is around 20 times more cost-effective than the best CPU-based method, when taking into account the monetary costs of the CPU and GPUs used. (Section VII)

## II. Preliminaries

This section first defines several basic concepts and formalizes the network motif discovery problem, and then introduces the architecture of Graphics Processing Units (GPUs).

### A. Problem Definition

Let $G = (V, E)$ be a directed, unlabelled graph[1] with a set $V$ of vertices and a set $E$ of edges. For any two vertices $u, v$ in $V$, we say that $v$ is an *out-neighbor* of $u$ if there is a directed edge from $u$ to $v$, i.e., $(u, v) \in E$. Conversely, we refer to $u$ as an *in-neighbor* of $v$. We define the *in-degree* (resp. *out-degree*) of $u$ as the number of in-neighbors (resp. out-neighbors) of $u$. In addition, we define the *bi-degree* of $u$ as the number of vertices that are both in-neighbors and out-neighbors of $u$, and we refer to those vertices as the *bi-neighbors* of $u$.

Let $g = (V_g, E_g)$ be a connected graph. We say that $g$ is a *subgraph* of $G$ (denoted by $g \subseteq G$), if and only if there exists at least one injective function $\zeta : V_g \to V$, such that (i) for any vertex $v \in V_g$, we have $\zeta(v) \in V$, (ii) for any edge $(u, v) \in E_g$, there is an edge $\big(\zeta(u), \zeta(v)\big) \in E$. For each subgraph of $G$ that is isomorphic to $g$, we refer to it as an *occurrence* of $g$ in $G$. The *frequency* of $g$ in $G$, denoted by $f(g, G)$, is the total number of occurrences of $g$ in $G$. $g$ is a *size-$k$* subgraph, if it contains exactly $k$ vertices.

[1]We consider that $G$ is directed and unlabelled, as it is a standard assumption in the literature. Nevertheless, our solution can be easily extended to handle undirected or labelled graphs.

We say that a graph $G' = (V', E')$ is *degree-equivalent* to $G$, if and only if (i) $|V'| = |V|$ and $|E'| = |E|$, and (ii) there exists a bijection $\psi : V \to V'$, such that for any node $v \in V$, $v$ and $\psi(v)$ have the same in-degree, out-degree, and bi-degree.

Let $\mathcal{G}$ denote the set of all graphs that are degree-equivalent to $G$. For any subgraph $g$ of $G$, its *expected frequency* $\overline{f}(g)$ is defined as its average frequency in all graphs in $\mathcal{G}$, i.e.,

$$\overline{f}(g) = \frac{1}{|\mathcal{G}|} \sum_{G' \in \mathcal{G}} f(g, G'). \tag{1}$$

Note that the exact value of $\overline{f}(g)$ is difficult to compute due to the enormous size of $\mathcal{G}$. Following the standard practice in the literature [1], we estimate $\overline{f}(g)$ using a sample set of $\mathcal{G}$ with $r$ graphs, denoted as $\mathcal{G}_r$. The estimation thus obtained is

$$\widetilde{f}(g) = \frac{1}{r} \sum_{G' \in \mathcal{G}_r} f(g, G'), \tag{2}$$

and the corresponding sample standard deviation is

$$\widetilde{\sigma}(g) = \sqrt{\frac{1}{r-1} \sum_{G' \in \mathcal{G}_r} \Big(f(g, G') - \widetilde{f}(g)\Big)^2}. \tag{3}$$

*Definition 1 (Motif):* Let $\theta > 0$ be a user-defined threshold. Given $G$, $\mathcal{G}_r$, and $\theta$, a subgraph $g$ of $G$ is a **motif** of $G$, if and only if $\widetilde{\sigma}(g) > 0$ and

$$f(g, G) - \widetilde{f}(g) \;\geq\; \theta \cdot \widetilde{\sigma}(g). \tag{4}$$

In other words, a motif of $G$ is a subgraph of $G$ that appears more frequently in $G$ than in random graphs that are degree-equivalent to $G$.

**Problem Statement.** Given $G$, $r > 0$, $k > 2$, and $\theta > 0$, the problem of network motif discovery asks for all size-$k$ motifs of $G$ with respect to $\mathcal{G}_r$ (i.e., a sample set of $\mathcal{G}$ with $r$ random graphs).

### B. Graphics Processing Units

GPUs were initially designed for graphical processing, but are now widely used for general-purpose parallel computing, e.g., sorting [15] and data mining [16]. Figure 1 shows the general architecture of a GPU. Compared with a CPU (which usually contains only a few cores), a GPU can easily have thousands of computation units. Specifically, a GPU contains several multiprocessors (MPs), each of which has a large number of stream processors (SPs). The SPs in each multiprocessor work in single-instruction multiple-data (SIMD) manner, i.e., they execute the same instructions at the same time on different input data. Each MP has a small but fast memory that is shared by all of its SPs. In addition, all SPs in the GPU share accesses to a larger but slower *global memory* of the GPU. Data can be exchanged between GPU's global memory and the main memory via a high speed I/O bus (e.g., PCI-Express), albeit at a relatively slow rate.

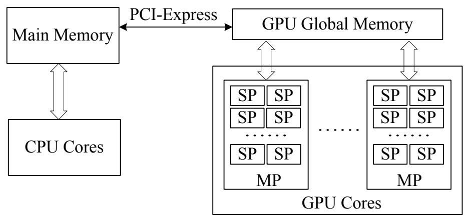


Fig. 1. The general architecture of a GPU.

For parallel computing on GPUs, we adopt the Nvidia CUDA programming framework. In the following, we introduce several key concepts in CUDA, so as to facilitate our discussions in the subsequent sections.

*Kernels.* A CUDA program alternates between codes running on the CPU and those on the GPU. The latter are referred to as *kernels*, and they are invoked only by the CPU. Each kernel starts by transferring input data from the main memory to the GPU's global memory, and then processes the data on the GPU; after that, it copies the results from the GPU's global memory back to the main memory, and then terminates.

*Thread Hierarchy.* The GPU executes each kernel with a user-specified number of threads. The threads are divided into a number of *blocks*, each of which is assigned to one MP (and cannot be re-assigned at runtime). In turn, each MP divides an assigned block of threads into smaller *warps*, and executes each warp of threads concurrently. Note that threads in the GPU cannot communicate with each other directly, but are allowed to retrieve data from, or write data to, arbitrary locations of the GPU's global memory.

*Branch Divergences.* Due to the SIMD nature of the GPU's SPs, all threads on the GPU cannot execute different programs at a given time. As a consequence, if two threads in a warp have different execution paths, then the GPU would execute those paths sequentially. For example, suppose that a piece of GPU code contains a statement "if $A$ then $B$, else $C$". For this statement, the GPU first asks each thread in a warp to evaluate condition $A$. Then, if $A$ equals *true* in some threads, the GPU executes $B$ on those threads; Meanwhile, the remaining threads in the warp remain idle, and they execute $C$ only after all other threads finish performing $B$. Such *branch divergence* is detrimental to the efficiency of GPU programs and should be avoided whenever possible [17].

*Memory Coalescing.* Suppose that the threads in a warp request to access data in the GPU's global memory, and the set of data requested is stored in consecutive memory addresses. In that case, the MP responsible for the warp would retrieve all data with one memory access and then distribute them to each thread, instead of issuing one access for each thread individually. This is referred to as *memory coalescing*, and it helps reduce memory access overheads. In contrast, if the data to be accessed is stored in $k$ disjoint memory spaces, then $k$ random accesses are required. Thus, it is important that we carefully arrange data in the GPU's global memory, so that the data required by each warp resides in consecutive locations.

## III. CPU-Based Methods Revisited

This section revisits existing CPU-based methods for network motif discovery. Section III-A summarizes the existing methods, while Section III-B elaborates why they cannot be translated into efficient solutions on GPUs.

### A. Summary of CPU-Based Methods

As mentioned in Section I, existing CPU-based methods [5], [8]–[13] for motif discovery typically run in two steps:

1) *Subgraph enumeration*: Compute the set $S_k$ of all subgraphs in the input graph $G$, as well as the frequency $f(g, G)$ of each subgraph $g$ in $G$.
2) *Frequency estimation*: Generate a set $\mathcal{G}_r$ of $r$ random graphs that are degree-equivalent to $G$. For each subgraph $g \in S_k$ and each random graph $G' \in \mathcal{G}_r$, compute the frequency $f(g, G')$ of $g$ in $G'$, and then determine whether $g$ is a motif of $G$ according to Equation 4.

Among the above two phases, *frequency estimation* incurs by far the highest overhead due to the large number of random graphs in $\mathcal{G}_r$ that need to be examined. To alleviate this overhead, existing methods construct indices on the subgraphs in $S_k$ for efficient subgraph search. In what follows, we first clarify the indices utilized by existing methods, and then explain the details of each phase.

**CL-Index and AM-Index.** Consider a random graph $G'$ in $\mathcal{G}_r$. To derive subgraph frequencies in $G'$, we need to identify, for any given subgraph $g'$ in $G'$, whether $g'$ appears in $S_k$. For this purpose, existing methods index the subgraphs in $S_k$ as follows. First, for each subgraph $g$ in $S_k$, they compute the *canonical labeling* [14] of $g$, which is a sequence of numbers that uniquely identifies a graph, i.e., two graphs have the same canonical labeling if and only if they are isomorphic. A hash index, referred to as the *canonical labeling index (CL-index)*, is built to map each canonical labeling to each subgraph $g \in S_k$.

Given the CL-index, we can determine whether a subgraph $g'$ appears in $S_k$, by first computing the canonical labeling of $g'$ and then checking whether the labeling appears in the CL-index. However, deriving the canonical labeling of $g'$ is often computationally expensive. To mitigate this issue, existing methods construct an additional hash index, referred to as the *adjacency matrix index (AM-index)*, that maps adjacency matrices to subgraphs in $S_k$. Specifically, for each subgraph $g$ in $S_k$, the AM-index records the adjacency matrix of at least one graph that is isomorphic to $g$. (Note that two isomorphic graphs may have different adjacency matrices.) As such, when we are to check whether a subgraph $g'$ is in $S_k$, we can first examine whether the adjacency matrix of $g'$ is indexed by AM-index. If it is indexed, then we have $g' \in S_k$; otherwise, we proceed to compute the canonical labeling of $g'$, and check if it appears in the CL-index. This filter-refinement approach leads to higher efficiency, as the adjacency matrix of $g'$ is much easier to compute than its canonical labeling.

**Phase 1: Subgraph Enumeration.** To enumerate all size-$k$ subgraphs in a given graph $G$, a naive approach is to examine all possible combinations of $k$ vertices in $G$. This approach, however, incurs prohibitive overheads due to the enormous number of vertex combinations that need to be inspected. In fact, most of the vertex combinations do not induce connected graphs, and hence, could have been ignored. (Recall that we require any subgraph to be connected.) Motivated by this, existing methods adopt the following heuristic to avoid inspecting all vertex combinations in subgraph enumeration. For each vertex $v$ in $G$, they first identify a set $nbr(v, k)$ of vertices that are at most $k$ hops to $v$ in $G$ (regardless of the directions of the edges). Observe that if a size-$k$ subgraph contains $v$, then all vertices in the subgraph must appear in $nbr(v, k) \cup \{v\}$. Accordingly, existing methods enumerate all size-$k$ subgraphs containing $v$, by inspecting the combinations of $k$ vertices in $nbr(v, k) \cup \{v\}$; if a vertex combination induces a connected graph $g$ containing $v$, then $g$'s adjacency matrix and canonical labeling are computed and inserted into the AM-index and CL-index, respectively. Once the $k$-hop neighborhood $nbr(v, k)$ of each vertex $v$ is processed, the subgraph enumeration step terminates.

**Phase 2.1: Random Graph Generation.** Existing methods adopt Monte Carlo techniques to generate random graphs that are degree-equivalent to $G$. The most well adopted technique is the *switching algorithm* [1], which runs in an iterative manner. In each iteration, the algorithm randomly selects two directed edges $\langle v_1, v_2\rangle$ and $\langle v_3, v_4\rangle$ from $G$, and then replaces them with two new edges $\langle v_1, v_4\rangle$ and $\langle v_3, v_2\rangle$, if the replacement does not result in a self-loop or two identical edges in $G$. The algorithm terminates after $\alpha|E|$ iterations, where $\alpha$ is a large pre-defined constant.

**Phase 2.2: Frequency Computation.** To compute the frequency of each subgraph $g \in S_k$ in each random graph $G' \in \mathcal{G}_r$, existing methods enumerate each size-$k$ subgraph $g'$ of $G'$, and then utilize the AM-index and CL-index to check if it appears in $S_k$. In addition, if $g'$ appears in the CL-index but not the AM-index, then its adjacency matrix $m$ is inserted into the AM-index, so that any other subgraph with the same adjacency matrix can be efficiently processed without inspecting the CL-index. Once all size-$k$ subgraphs from all random graphs are examined, the estimated frequency of each subgraph in $S_k$ (with respect to $\mathcal{G}_r$) is computed, based on which the motifs of $G$ are identified.

### *B. Difficulties in GPU Translations*

The CPU-based methods reviewed in Section III-A are difficult to be adopted on GPUs, for three reasons. First, both phases of the existing methods require computing the canonical labelings of numerous subgraphs. The algorithms [14], [18] for computing canonical labelings, however, contain complicated execution paths with a large number of branches. As a consequence, if we are to directly adopt those algorithms on GPUs, the execution of the algorithms would be highly inefficient due to the effects of branch divergences (see Section II-B).

Second, CPU-based methods rely on the CL-index and AM-index to check whether a subgraph appears in $S_k$. If we adopt the same approach on a GPU, then we need to store the CL-index and AM-index in the GPU's global memory, and ask each GPU thread to probe the indices for subgraph matching. In that case, the GPU threads in each warp are likely to access drastically different memory, which prevents the GPU from applying memory coalescing to reduce memory access costs. Furthermore, when $G$ is large, the CL-index and AM-index can become so large that they do not even fit in the global memory of the GPU.

Finally, if we are to ask each GPU thread to examine whether a subgraph appears in $S_k$, then it is likely that some threads will incur considerably higher overheads than the others, since the subgraphs in random graphs may have much different structures. Therefore, there can be significant imbalance in the GPU threads' workload. In that case, all GPU threads in the same warp would need to wait for the slowest thread to finish, before they can be terminated to allow new GPU threads to be created. This leads to severe under-utilization of the GPU's parallel processing power.

**Algorithm 1:** FreqComp

```
input : $g \in S_k$ and $G' \in \mathcal{G}_r$
output: $f(g, G')$
1 [CPU]: Choose a matching order of the vertices in $g$, denoted as
  $\langle u_1, u_2, \ldots, u_k\rangle$ (see Section V-B);
2 Let $g(2)$ be the subgraph of $g$ induced by $\{u_1, u_2\}$;
3 [CPU]: Identify the set $C_2$ of all subgraphs in $G'$ that are isomorphic to $g_2$;
4 for $i = 3, \cdots, k$ do
5     Let $g(i)$ be the subgraph of $g$ induced by $\{u_1, u_2, \ldots, u_i\}$;
6     [GPU]: Based on $C_{i-1}$, compute the set $C_i$ of subgraphs in $G$ that are
      isomorphic to $g(i)$ (see Algorithm 2);
7 return $|C_k|$;
```

## IV. Solution Overview

As with existing CPU-based methods, our solution also consists a subgraph enumeration phase and a frequency estimation phase. In particular, the subgraph enumeration phase of our solution adopts the CPU-based method in [12], and the frequency estimation phase also utilizes the CPU-based switching algorithm [1] to generate the set $\mathcal{G}_r$ of random graphs that are degree-equivalent to $G$. To compute the average frequency of each subgraph $g \in S_k$ with respect to $\mathcal{G}_r$, however, we employ a GPU-based algorithm that provides much higher efficiency than existing CPU-based methods. The reason that we focus on optimizing the computation of average subgraph frequency is that it incurs significantly higher overheads than the other components of our solution (due to the large number of random graphs to be processed). In the following, we present an overview of our GPU-based algorithm, assuming that $G$, $S_k$, and $\mathcal{G}_r$ are given.

In a nutshell, our algorithm examines each pair of $g$ and $G'$ where $g \in S_k$ and $G' \in \mathcal{G}_r$, and it computes the frequency of $g$ in $G'$ with the *FreqComp* method in Algorithm 1. The algorithm first arranges the vertices in $g$ in a certain sequence $\langle u_1, u_2, \ldots, u_k\rangle$ (Line 1), such that for any $i \in [2, k]$, $u_1, u_2, \ldots, u_i$ induces a connected subgraph of $g$ (denoted as $g(i)$). We refer to $\langle u_1, u_2, \ldots, u_k\rangle$ as a *matching order*, and we clarify how it is derived in Section V-B. After that, it identifies all subgraphs of $G'$ that are isomorphic to $g(2)$, and stores them in a set $C_2$ (Lines 2-3).

The subsequent part of the algorithm runs in $k-2$ iterations (Lines 4-6), such that each iteration utilizes the GPU to transform $C_{i-1}$ ($i \in [3, k]$) into $C_i$, i.e., the set of all subgraphs of $G'$ that are isomorphic to $g(i)$. Once $C_k$ is computed, the algorithm terminates and returns $|C_k|$, which equals the frequency of $g$ in $G'$ (Line 7). We clarify the generation of $C_i$ in Section V-B.

Compared with the existing CPU-based methods [5], [8]–[13], *FreqComp* does not compute any canonical labeling or construct any index on $S_k$, which helps avoid the branch divergence and memory coalescing issues that render existing methods inefficient on GPUs. Instead, *FreqComp* adopts an incremental approach that first identifies the subgraphs of $G'$ that can be matched to parts of $g$ (i.e., $g(2), g(3), \ldots, g(k-1)$), and then utilizes such "partial occurrences" of $g$ to pinpoint the size-$k$ subgraphs of $G'$ that are isomorphic to $g$. This incremental approach is not as efficient as the CPU-based indexing methods in deciding whether a *single* subgraph of $G'$ is isomorphic to $g$, but it is much more amendable to GPU paralellization.

It is noteworthy that *FreqComp* is similar in spirit to existing CPU-based algorithms [18]–[21] for *subgraph isomorphism tests*, which incrementally match the vertices in a small graph $g$ to those in a larger graph $G'$ to decide whether $g$ appears in $G'$. However, *FreqComp* aims to decide the exact number of occurrences of $g$ in $G'$, whereas the algorithms in [18], [20], [21] only determine whether $g$ has at least one occurrence in $G'$. Furthermore, as the algorithms in [18]–[21] are CPU-based, they involve complex execution paths, which render them unsuitable for GPU adoptions, due to the effect of branch divergences. In contrast, *FreqComp* is devised with careful considerations of GPUs' characteristics, which lead to design choices that drastically differ from those in [18]–[21], as we demonstrate in Section V.

## V. GPU-Based Subgraph Matching

This section presents the details of the *FreqComp* algorithm. Section V-A clarifies how we represent each random graph $G'$ in a GPU's global memory. Sections V-B and V-C elaborates each step of *FreqComp*. Section V-D proves *FreqComp*'s correctness.

### A. Representation of Graphs

We represent each random graph $G' \in \mathcal{G}_r$ using six arrays in the GPU's global memory, namely, $E_{out}$, $E_{in}$, $E_{bi}$, $O_{out}$, $O_{in}$, and $O_{bi}$. Each element in $E_{out}$ (resp. $E_{in}$) is an ordered pair of vertices $\langle v_a, v_b \rangle$, such that $v_b$ is an out-neighbor (resp. in-neighbor) of $v_a$. Meanwhile, each element in $E_{bi}$ is an ordered pair of vertices that are bi-neighbors of each other. All pairs in $E_{out}$, $E_{in}$, and $E_{bi}$ are sorted by their first vertices, with ties broken based on the second vertices. As a consequence, the pairs with the same first vertices are stored as a *block* of consecutive elements in $E_{out}$, $E_{in}$, and $E_{bi}$. We refer to $E_{out}$, $E_{in}$, and $E_{bi}$ as the *edge arrays*.

On the other hand, $O_{out}$ is an array that maps each vertex in $G'$ to its corresponding block in $E_{out}$. Specifically, for the $i$-th vertex $v_i$ in $G'$, if it has at least one out-neighbor, then the $i$-th element in $O_{out}$ records the position of the first element in $E_{out}$ where $v_i$ is the first vertex. If $v_i$ has no out-neighbor, however, then the $i$-th element in $O_{out}$ is identical to the $(i+1)$-th element. For convenience, we append an extra element to the end of $O_{out}$, and set its value to the total number of elements in $E_{out}$ plus one. We refer to $O_{out}$ as the *offset array* for $E_{out}$. Accordingly, $O_{in}$ and $O_{bi}$ are the offset arrays for $E_{in}$ and $E_{bi}$, respectively, and are defined in the same manner.

By the way $O_{out}$ is constructed, if we are to identify the out-degree of the $i$-th vertex in $G'$, then we can simply subtract the $i$-th element in $O_{out}$ from the $(i+1)$-th element. The in-degree (resp. bi-degree) of any vertex can be computed from $O_{in}$ (resp. $O_{bi}$) in the same manner.

### B. Construction of $C_i$

As mentioned in Section IV, the *FreqComp* algorithm first determines a matching order $\langle u_1, u_2, \ldots, u_k \rangle$ for the vertices in $g$, and then iteratively identifies the set $C_i$ ($i \in [2, k]$) of subgraphs in $G'$ that are isomorphic to $g(i)$, i.e., the subgraph of $g$ induced by $\{u_1, u_2, \ldots, u_i\}$. For ease of exposition, we defer the discussion of the matching order to end of Section V-B. In what follows, we first clarify the construction of $C_i$, assuming that $\langle u_1, u_2, \ldots, u_k \rangle$ are given.

**Construction of $\boldsymbol{C_2}$.** First, consider the case when $i = 2$. If $u_1$ and $u_2$ are bi-neighbors, then $C_2$ consists of all 2-cycles in $G'$; otherwise, $C_2$ contains all forward (resp. backward) edges in $G'$ if $u_2$ is an out-neighbor (resp. in-neighbor) of $u_1$. In any of those three cases, $C_2$ can be constructed with a linear scan of $E_{in}$, $E_{out}$, or $E_{bi}$.

For each graph $c \in C_2$, we record the two vertices of the graph as an ordered pair, where the first and second vertices are mapped to $u_1$ and $u_2$, respectively, in the isomorphism of $c$ and $g(2)$. In case that there exist multiple isomorphisms (i.e., when $u_1$ and $u_2$ are bi-neighbors), we store in $C_2$ one ordered pair of each mapping. In general, for graph $c \in C_i$ and each isomorphism of $c$ and $g(i)$, we store a sequence of $i$ vertices in $C_i$, such that the $j$-th ($j \in [1, i]$) vertex in the sequence is the vertex in $c$ mapped to $u_j$ in the isomorphism. (We discuss in Section V-C how we may reduce the number of sequences in $C_i$ without affecting the correctness of our solution.) For convenience, we refer to each sequence $s$ in $C_i$ as a *size-i candidate*, and we abuse notation by using $s$ to refer to the graph that it represents.

**Construction of $\boldsymbol{C_3, C_4, \ldots, C_k}$.** Next, suppose that we have constructed $C_{i-1}$ ($i \in [3, k]$), based on which we are to compute $C_i$ by launching a kernel on the GPU. Our basic idea is to invoke a large number of parallel GPU threads, such that each thread (i) examines a size-$(i-1)$ candidate $c \in C_{i-1}$ and (ii) tries to transform $c$ into a size-$i$ candidate $c^*$ by adding one vertex in $G'$ into $c$.

To explain, recall that $c$ is a size-$(i-1)$ subgraph of $G'$ that is isomorphic to $g(i-1)$. Let $v_j$ ($j \in [1, i-1]$) be the

**Algorithm 2:** BuildCi

```
input : $g$, $G'$, and $C_{i-1}$
output: $C_i$
1 $\{A_o, A_v\}$ = InitArray ($g$, $C_{i-1}$);    // see Algorithm 3
2 $\{I, A'_o\}$ = GenCand($A_o$, $A_v$, $g$, $G'$, $C_{i-1}$);    // see Algorithm 4
3 $C_i$ = CleanCand($I$, $A'_o$);    // see Algorithm 5
4 return $C_i$;
```

vertex in $c$ that is mapped to $u_j$ in $g(i-1)$. To convert $c$ into a graph isomorphic to $g(i)$, a natural approach is to inspect each neighbor $v$ of each $v_j$ to see if $v$ can be mapped to $u_i$. That is, we check whether the following condition holds:

- *Vertex Validity Condition:* For all $j \in [1, i-1]$, if $u_j$ is an in-neighbor of $u_i$ in $g$, then $v_j$ is an in-neighbor of $v_i$; furthermore, if $u_j$ is an out-neighbor of $u_i$ in $g$, then $v_j$ is an out-neighbor of $v_i$.

If the above condition holds for $v$, the subgraph of $G'$ induced by $\{v_1, \ldots, v_{i-1}, v\}$ must be isomorphic to $g(i)$; accordingly, we can record the sequence $\langle v_1, \ldots, v_{i-1}, v\rangle$ as a size-$i$ candidate in $C_i$.

To implement the above approach on a GPU, a straightforward method is to create one GPU thread for each neighbor $v$ of a vertex in $c \in C_{i-1}$, to check whether the vertex validity condition holds for $v$. This method, however, requires different threads in the same warp to access the neighbors of different vertices $v$, which diminishes the chance of memory coalescing since the edges of different vertices are unlikely to reside in consecutive memory addresses. Furthermore, the method leads to workload unbalance among the threads, as different vertices $v$ may have drastically different numbers of neighbors.

We address the above deficiencies with a more advanced method as follows. Without loss of generality, assume that $u_i$ is an out-neighbor of a vertex $u_\alpha$ ($\alpha \in [1, i-1]$) in $g$. Then, for each vertex $v$ in $G'$ that is an out-neighbor of $v_\alpha$, we create $|E(v)|$ GPU threads, where $E(v)$ is the set of all edges incident to $v$ in $G'$. (We refer to $v_\alpha$ as the *anchor vertex*.) In particular, the $\ell$-th thread examines the $\ell$-th edge $e' \in E(v)$, and checks whether the following *edge validity conditions* hold simultaneously:

1) $e'$ connects $v$ to some $v_j$ ($j \in [1, i-1]$) in $c$.
2) If $e'$ is an outgoing edge from $v$, then $u_j$ is an out-neighbor of $u$.
3) If $e'$ is an incoming edge to $v$, then $u_j$ is an in-neighbor of $u$.

The thread returns *true* if all of the above conditions hold, and *false* otherwise. After all $|E(v)|$ threads terminate, we count the number of threads that return *true*. If this number equals the number of edges incident to $u_i$, then we confirm that $v$ satisfies the validity condition. In that case, we insert $\langle v_1, \ldots, v_{i-1}, v\rangle$ into $C_i$ as a size-$i$ candidate.

Compared with the straightforward method, the advanced method increases the total workload on GPU, since the latter examines all $|E(v)|$ edges of each $v$, whereas the former only needs to perform $|E(u_i)|$ binary searches on $v$'s edge lists. As a trade-off, however, the advanced approach has a much smaller running time for two reasons. First, it ensures a balanced workload for each GPU thread. Second, it facilitates memory coalescing, because (i) the threads in the same warp are likely to handle the neighbors of the same $v$, and (ii) the edges of $v$ are stored in consecutive addresses.

**Algorithm 3:** InitArray

```
input : $g$ and $C_{i-1}$
output: $A_o$ and $A_v$
1 create arrays $A_{deg}$ and $A_v$, both of size $|C_{i-1}|$;
2 for each $x = 1, 2, \ldots, |C_{i-1}|$ in parallel do
3     let $c_x = \langle v_1, \ldots, v_{i-1}\rangle$ be the $x$-th graph in $C_{i-1}$;
4     identify the vertex $v_\alpha$ in $c_x$ with the smallest non-zero relevant degree;
5     $A_{deg}[x] \leftarrow$ the relevant degree of $v_\alpha$;
6     $A_v[x] \leftarrow v_\alpha$;
7 run a parallel prefix sum on $A_{deg}$; let $A_o$ be the resulting array;
8 return $A_v$ and $A_o$;
```

It remains to discuss how we select the anchor vertex $v_\alpha$ ($\alpha \in [1, i-1]$) to start the exploration of candidate vertices $v$. For any $j \in [1, i-1]$, we define $v_j$'s *relevant neighbor set* as:

$$R(v_j) = \begin{cases} v_j\text{'s out-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s out-neighbor;} \\ v_j\text{'s in-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s in-neighbor;} \\ v_j\text{'s bi-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s bi-neighbor;} \\ \emptyset, & \text{otherwise.} \end{cases}$$

We also define $|R(v_j)|$ as the *relevant degree* of $v_j$. Observe that (i) we can set $v_\alpha = v_j$ only if $R(v_j) \neq \emptyset$, and (ii) if $v_\alpha = v_j$, then the advanced method needs to explore $|R(v_j)|$ of $v_j$'s relevant neighbors. To minimize the number of vertices that need to be explored, we set $v_\alpha$ to the vertex in $c$ with the smallest *non-zero* relevant degree.

**Implementation.** Algorithm 2 shows the pseudo-code of our method (dubbed *BuildCi*) for constructing $C_i$ from $C_{i-1}$ ($i \in [3, k]$). The algorithm first invokes the *InitArray* function (Algorithm 3) to create two arrays $A_v$ and $A_o$. In particular, $A_v$ stores $|C_{i-1}|$ vertices, such that the $x$-th vertex is the anchor vertex in the $x$-th graph in $C_{i-1}$. Meanwhile, $A_o$ records $|C_{i-1}|$ *offset* values, such that the $x$-th offset equals the sum of the relevant degrees of first $x$ vertices in $A_v$. These offsets are used to indicate the memory locations where the GPU threads in the subsequent step should write their outputs to.

Next, *BuildCi* feeds $A_v$ and $A_o$ as inputs to the *GenCand* function (Algorithm 4). Let $c_x$ denote the $x$-th graph in $C_{i-1}$. *GenCand* examines each $c_x = \langle v_1, \ldots, v_{i-1}\rangle$, and retrieves from $A_v$ the anchor vertex $v_\alpha$ in $c_x$. For each vertex $v$ in the relevant neighbor set of $v_\alpha$, *GenCand* regards $\langle v_1, \ldots, v_{i-1}, v\rangle$ as a potential size-$i$ candidate, and uses a GPU thread to write it into an array $I$. In addition, *GenCand* creates an array $A'_{deg}$, where the $j$-th element equals the number of edges incident to the vertex $v$ associated with the $j$-th element in $I$. It then generates an array $A'_o$ of offset values, by computing the prefix sum of $A'_{deg}$. Finally, it returns $I$ and $A'_o$.

Finally, *BuildCi* applies the *RefineCand* function (Algorithm 5) to refine the potential size-$i$ candidates in $I$. In particular, for each $\langle v_1, \ldots, v_{i-1}, v\rangle$ recorded in $I$, *RefineCand* creates $|E(v)|$ GPU threads, each of which (i) checks whether an edge of $v$ satisfies all of the edge validity conditions, and (ii) writes the result of the check into an array $B$. Then,

**Algorithm 4:** GenCand

```
input : $A_o$, $A_v$, $g$, $G'$, and $C_{i-1}$
output: $I$ and $A'_o$
1  let $\theta$ be the last element of $A_o$;
2  create arrays $A'_{deg}$ and $I$, both of the size $\theta$;
3  for each $y = 1, 2, \ldots, \theta$ in parallel do
4      identify the integer $x$ such that $A_o[x] \le y < A_o[x+1]$;
5      let $c_x$ be the $x$-th graph in $C_{i-1}$;
6      let $v_\alpha$ be the vertex recorded in $A_v[x]$;
7      let $z = y - A_o[x]$;
8      $v \leftarrow$ the $z$-th vertex in $v_\alpha$'s relevant neighbor set;
9      $I[y] \leftarrow \langle v_1, \ldots, v_{i-1}, v \rangle$;
10     $A'_{deg}[y] \leftarrow$ the number of edges incident to $v$ in $G'$;
11 run a parallel prefix sum on $A'_{deg}$; let $A'_o$ be the resulting array;
12 return $I$ and $A'_o$;
```

*RefineCand* examines $B$ to identify those vertices $v$ that has $E(u_i)$ edges passing the validity check; for each such $v$, it inserts $\langle v_1, \ldots, v_{i-1}, v \rangle$ into $C_i$ as a size-$i$ candidate. After that, the algorithm returns $C_i$ and terminates.

**Matching Order.** We now discuss how we decide the matching order for the vertices in $g$. In general, we aim to select a matching order that minimizes the sizes of $C_i$ ($i \in [2, k-1]$), so as to reduce computation overheads. That is, we aim to arrange the vertices in $g$ into a sequence $u_1, u_2, \ldots, u_k$, such that each subgraph induced by $u_1, u_2, \ldots, u_j$ ($j \in [2, k]$) has as fewer occurrences in $G'$ as possible. This problem has been studied in the context of subgraph isomorphism tests, and there exist several CPU-based heuristic solutions [18]–[21]. In our solution, we adopt the CPU-based technique in [21] for choosing a matching order for $g$. We do not consider GPU-based techniques, since the costs of generating matching orders are insignificant when compared with the overheads of the other parts of our solution.

## C. Avoiding Duplicates

Let $g'_i$ be an occurrence of $g_i$ in $G'$. As mentioned in Section V-B, if there are multiple isomorphisms of $g'_i$ and $g_i$, then we record each isomorphism as a vertex sequence in $C_i$. This could lead to an excessive number of vertex sequences in $C_i$. For example, if $g_i$ is a clique, then there exist $i!$ isomorphisms of $g_i$ and $g'_i$, which result in $i!$ vertex sequences in $C_i$. Previous work [6] addresses this problem with technique that exploits graph automorphism, and we adopt the same technique in our GPU-based solution. To explain, we first introduce the concept of *automorphism groups*.

*Definition 2 (Automorphism Groups): An* **automorphism group** *of $g$ is a set $A$ of ordered pairs that satisfy the following conditions:*

1) *The two elements in each order pair are vertices in $g$.*
2) *In each ordered pair $(u, u')$, the vertex ID of $u$ is smaller than that of $u'$.*
3) *The set of edges in $g$ remains unchanged even if, for each ordered pair $(u, u') \in A$, we exchange $u$ and $u'$ in all edges incident to $u$ or $u'$.*

For example, consider the graph $g_1$ in Figure 2, assuming that the ID of each node $u_i$ ($i \in [1, 6]$) equals $i$. $A_1 = \{(u_1, u_2)\}$ is an automorphism group of $g_3$ because (i) there

**Algorithm 5:** RefineCand

```
input : $I$ and $A'_o$
output: $C_i$
1  let $\gamma$ be the number of edges incident to $u_i$ in $g$ ;
2  create an array $B$ of size $\gamma \cdot |I|$, with all elements set to 0 ;
3  let $\theta'$ be the value of the last element of $A'_o$;
4  for each $z = 1, 2, \ldots, \theta'$ in parallel do
5      identify the integer $y$ such that $A'_o[y] \le z < A'_o[y+1]$;
6      let $\ell = z - A'_o[y]$;
7      let $\langle v_1, \ldots, v_{i-1}, v \rangle$ be the $y$-th element of $I$;
8      let $e'$ be the $\ell$-th edge incident to $v$ in $G'$;
9      let $v'$ be the node that is connected to $v$ by $e$ ;
10     if there exists $v_j = v'$ ($j \in [1, i-1]$) then
11         scan the edges of $u$;
12         if the $\beta$-th edge $e$ connects $u$ to $u_j$ then
13             if ($e$ starts from $v$ and $e'$ starts from $u$) or ($e$ starts from $v_j$ and
               $e'$ starts from $u_j$) then
14                 $B[y \cdot \gamma + \beta] = 1$;
15 create an array $B^*$ of size $|I|$ with all elements set to 0;
16 for each $y = 1, 2, \ldots, |I|$ in parallel do
17     if $B[y \times \gamma + \ell]$ equals 1 for each $\ell \in [1, \gamma]$ then
18         $B^*[y] = 1$;
19 run a parallel prefix sum on $B^*$; let $A^*_o$ be the result;
20 let $\theta^*$ be the last element of $A^*_o$;
21 create an array $C_i$ of size $\theta^*$;
22 for each $y = 1, 2, \ldots, |I|$ in parallel do
23     if $B^*[y]$ equals 1 then
24         $C_i[A^*_o[y]] \leftarrow I[y]$;
25 return $C_i$;
```

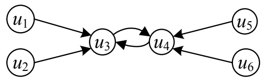


Fig. 2. A graph $g_1$ with several automorphism groups.

are only two edges in $g_3$ that are incident to $u_1$ or $u_2$, namely, $\langle u_1, u_3 \rangle$ and $\langle u_2, u_3 \rangle$, and (ii) even if we exchange $u_1$ and $u_2$ into those two edges, we still have $\langle u_2, u_3 \rangle$ and $\langle u_1, u_3 \rangle$, i.e., the set of edges in $g_3$ remain unchanged. It can be verified that $g_3$ has another two automorphism groups $A_2 = \{(u_1, u_5), (u_2, u_6), (u_3, u_4)\}$ and $A_3 = \{(u_5, u_6)\}$.

Based on $g$'s automorphism groups, we construct a *symmetry constraint set (SCS) $Q$* for $g$, which contains exactly one ordered pair from each automorphism group. For example, for the graph $g_3$ in Figure 2, $\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$ is an SCS, since, as mentioned, (i) $g_3$ has three automorphism groups $A_1$, $A_2$, and $A_3$, and (ii) $(u_1, u_2) \in A_1$, $(u_3, u_4) \in A_2$, and $(u_5, u_6) \in A_3$.

Given an SCS $Q$ for $g$, we impose the following *symmetry constraint* on each size-$i$ candidate $c = \langle v_1, v_2, \ldots, v_i \rangle$ in $C_i$:

- *Symmetry Constraint:* For any ordered pair $(u_x, u_y) \in Q$ with $1 \le x < y \le i$, the vertex $v_x$ in $c$ has a smaller ID than the vertex $v_y$ does.

It is proved in [6] that even if there are multiple isomorphisms of $g$ and a subgraph of $G'$, only one of them satisfies the symmetry constraint given an SCS. Hence, imposing the symmetry constraint eliminates duplicates[2] in $C_k$, and ensures that the frequency of $g$ can be correctly computed from $C_k$.

In our solution, we compute an SCS $Q$ for $g$ using the CPU-based algorithm in [6], and we impose the symmetry constraint

[2]It also reduces the number of duplicates in $C_i$ ($i \in [2, k-1]$) but does not necessarily eliminate them, since an SCS is computed based on the automorphism groups of $g$ instead of $g(i)$.

on $C_i$ during its generation in our GPU-based Algorithm 4. In particular, after Line 9 of Algorithm 4 constructs a potential size-$i$ candidate $c = \langle v_1, v_2, \ldots, v_i \rangle$, we test whether $c$ satisfies the symmetry constraint by inspecting all ordered pairs in $Q$ that contain $u_i$. If $c$ fails the test, then we set $A'_{deg}[y] = 0$ in Line 10; this ensures that $c$ will be subsequently eliminated by the *RefineCand* function.

### D. Correctness

The following lemma shows the correctness of our solution.

*Lemma 1: For any $g$ and $G' \in \mathcal{G}_r$, Algorithm 1 correctly computes the frequency of $g$ in $G'$.*

*Proof Sketch:* To prove the lemma, we show that the set $C_k$ constructed by Algorithm 2 contains exactly one vertex sequence for each occurrence of $g$ in $G'$. First, due to the symmetry constraint approach [6] in Section V-C, $C_k$ contains at most one vertex sequence for each occurrence of $g$ in $G'$. Second, by an induction on $i$, we can prove that there is at least one vertex sequence in $C_i$ for each occurrence of $g_i$ in $G'$, since (i) each occurrence of $g_i$ in $G'$ can be obtained by extending an occurrence of $g_{i-1}$ in $G'$ by one vertex, and (ii) Algorithm 2 considers all such extensions when constructing $C_i$ from $C_{i-1}$. ■

## VI. Optimizations

This section presents several crucial techniques for optimizing the performance of our GPU-based method.

### A. Handling Large Candidate Sets

Our GPU-based solution requires generating a few intermediate results, e.g., the size-$i$ candidate sets $C_i$ and the temporary arrays (e.g., $A_{deg}$, $A_o$, $I$, $B$) utilized in Algorithms 3, 4, and 5. When $G_r$ is sizable, those intermediate results could be too large to fit in the global memory of the GPU. To address this issue, one straightforward approach is to use the machine's main memory (and harddisk, if necessary) as a secondary storage for the GPU. In particular, if the intermediate results in the conversion from $C_i$ to $C_{i+1}$ ($i \in [2, k-1]$) exceed the size of the GPU memory, then we may store $C_i$ in the main memory, and divide it into several subsets $C_i^{(1)}, C_i^{(2)}, \ldots, C_i^{(\beta)}$, such that each subset is small enough to be processed by the GPU. After that, we transfer the subsets to the GPU one by one, and ask the GPU to (i) convert each subset $C_i^{(j)}$ into a *partial* set $C_{i+1}^{(j)}$ of size-$(i+1)$ candidates and (ii) send each $C_{i+1}^{(j)}$ back to the main memory. Once all $C_{i+1}^{(j)}$ are produced, we take their union to obtain the size-$(i+1)$ candidate set $C_{i+1}$.

The above approach, however, is inefficient as it requires numerous rounds of data transfers between the main memory and the GPU memory, which are only connected via a (relatively slow) I/O bus. To address this problem, we propose a divide-and-conquer approach that processes all data in the GPU's global memory, without utilizing the main memory as a secondary storage. To explain, assume that the GPU memory is sufficient to construct $C_2, C_3, \ldots, C_i$, but not $C_{i+1}$. That is, the GPU would first run out of memory when transforming $C_i$ to $C_{i+1}$. We first clarify how our approach works when $i = k-2$, and then extend our discussion to the general case.

Given $C_{k-2}$, we first invoke Algorithm 3 to obtain two arrays $A_v$ and $A_o$. Recall that the $j$-th element of $A_o$ equals the number of potential size-$(k-1)$ candidates that we need to generate from the $z$-th graphs in $C_{k-2}$ where $z \leq j$. Therefore, based on $A_o$, we can calculate the amount of memory required in processing each graph in $C_{k-2}$. Given this information, we divide $C_{k-2}$ into subsets, such that each subset $C_{k-2}^{(j)}$ can be converted into a set $C_{k-1}^{(j)}$ of size-$(k-1)$ candidates using a fraction $\lambda$ of the vacant memory on the GPU. (We will discuss the setting of $\lambda$ shortly.) Then, we process each $C_{k-2}^{(j)}$ in turn. Whenever a size-$(k-1)$ candidate subset $C_{k-1}^{(j)}$ is generated, however, we do not transfer it to the main memory of the machine; instead, we use the remaining $1-\lambda$ fraction of the vacant GPU memory to convert $C_{k-1}^{(j)}$ into a size-$k$ candidate subset $C_k^{(j)}$. In case that this conversion requires more memory than available, we further divide $C_{k-1}^{(j)}$ into subsets and process each subset in turn, in the same manner as the processing of $C_{k-2}$. Once a subset of size-$k$ candidates are produced, we record the size of the subset, and then delete the subset from the GPU memory to make room for the processing of other subsets of $C_{k-1}^{(j)}$. In summary, we partition the vacant memory of the GPU into two parts, and use them to pipeline the generation of size-$(k-1)$ and size-$k$ candidates.

In general, if we have sufficient GPU memory to construct $C_2, \ldots, C_i$ but not $C_{i+1}$, we start pipelining right after $C_i$ is generated. Specifically, we divide the vacant GPU memory into $k-i$ parts, and assign the $j$-th part for the conversion from $C_{i+j-1}$ to $C_{i+j}$. We heuristically set the size of the $j$-th part to be $\lambda$ fraction of the GPU memory that is vacant after the first $j-1$ parts are assigned, except that the last part utilizes all remaining GPU memory. To choose an appropriate value for $\lambda$, we model the total number of candidate subsets produced in the pipelining process (i.e., the total number of "splits" required on $C_i, \ldots, C_{k-1}$) as a function of $\lambda$, and we derive the $\lambda$ that minimizes the function. The rationale is as follows: each candidate subset needs to be processed with a few GPU kernels, each of which takes a certain amount of time to start up; therefore, if the total number of candidate subsets is large, then the total start-up overhead of the GPU kernels would be significant, which leads to inferior efficiency.

Let $M$ be the amount of vacant GPU memory right after $C_i$ is constructed. Observe that, in the conversion from $C_i$ to $C_{i+1}$, the total size of the intermediate results is $O(|C_i|)$. Given that we assign $\lambda M$ GPU memory for the conversion from $C_i$ to $C_{i+1}$, the number of subsets of $C_i$ generated is roughly proportional to $\frac{|C_i|}{\lambda M}$. By the same reasoning, the number of subsets of $C_j$ ($j > i$) produced is approximately proportional to

$$\begin{cases} \dfrac{|C_j|}{\lambda \cdot (1-\lambda)^{j-i} \cdot M}, & \text{if } j \in [i+1, k-2]; \\ \dfrac{|C_{k-1}|}{(1-\lambda)^{k-i-1} \cdot M}, & \text{if } j = k-1. \end{cases}$$

Observe that $|C_{j+1}| \leq d \cdot |C_j|$, where $d$ is the maximum vertex degree in $G$. We consider that $|C_{j+1}| = d \cdot |C_j|$, in which case the total number of subsets produced in the pipelining process is roughly proportional to:

$$\left( \sum_{j=i}^{k-2} \frac{d^{j-i} \cdot |C_i|}{\lambda \cdot (1-\lambda)^{j-i} \cdot M} \right) + \frac{d^{k-i-1} \cdot |C_i|}{(1-\lambda)^{k-i-1} \cdot M}. \qquad (5)$$

It can be verified that Equation 5 is minimized when $\lambda = \frac{1}{k-i}$. Therefore, we set $\lambda = \frac{1}{k-i}$ in our solution.

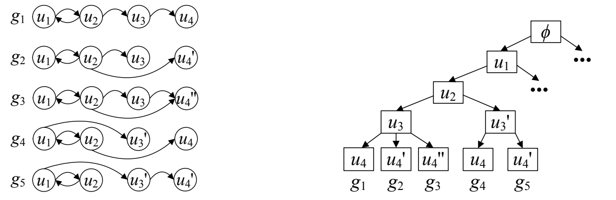
### B. Handling Multiple Graphs

Our previous discussions have focused on computing subgraph frequencies in one random graph $G' \in \mathcal{G}_r$. When $G'$ contains relatively small numbers of vertices and edges, the frequency computation processes on $G'$ may not engage all GPU cores, which leads to under-utilization of the GPU. We address this issue as follows. First, we divide the random graphs in $\mathcal{G}_r$ into several groups, each of which contains $\mu$ graphs, where $\mu$ is a tunable parameter. After that, for each group $R$ of random graphs, we regard it as a graph $G^*$ consisting of $|R|$ disjoint components, each of which is a graph in $R$. Then, we invoke our GPU-based frequency estimation method on $G^*$, with additional bookkeeping to (i) record the subgraph frequencies in each random graph separately, and (ii) ignore any subgraph of $G^*$ that contains vertices from different graphs in $R$. In other words, we process the random graphs in each group in a batch manner, and thus, we avoid under-utilizing the GPU.

One crucial question remains: How do we decide the number $\mu$ of random graphs in each group? A naive approach is to set $\mu = |\mathcal{G}_r|$, i.e., we process all random graphs in $\mathcal{G}_r$ in one batch. This, however, severely exacerbates the GPU memory issue discussed in Section VI-A, and leads to inferior efficiency. To tackle the problem, we choose $\mu$ using a heuristic method, as explained in the following. First, observe that for any subgraph $g$ in $G$ and any random graph $G' \in \mathcal{G}_r$, the size-2 candidate set of $G'$ (i.e., $C_2$) has a size at most $\mu \cdot m$, where $m$ is the number of edges in $G'$. In addition, each size-2 candidate in $C_2$ leads to at most $d$ possible size-3 candidates in Algorithm 4, where $d$ is the maximum vertex degree in $G$. Given $\mu \cdot m$ and $d$, we can derive an upperbound $\tau$ of the amount of GPU memory required in the conversion from $C_2$ to $C_3$, and we set $\mu$ to the maximum integer such that the upperbound $\tau$ no more than $\frac{1}{k-2}$ fraction of the vacant GPU memory. In other words, we ensure that the conversion from $C_2$ to $C_3$ can be performed without invoking the divide-and-conquer method in Section VI-A, which helps avoid overloading the GPU. Although such a $\mu$ thus obtained does not guarantee that the GPU won't be overloaded in the computation of $C_4, C_5, \ldots, C_k$, we find that it leads to satisfactory performance in our experiments.

### C. Matching Tree

Let $g$ and $g'$ be two size-$k$ subgraphs of $G$ (i.e., $g, g' \in S_k$) that differ in only one node. Further assume that the matching orders of $g$ and $g'$ share a common prefix of length $k-1$, e.g.,



$g_1$ $u_1$ $u_2$ $u_3$ $u_4$
$g_2$ $u_1$ $u_2$ $u_3$ $u_4'$
$g_3$ $u_1$ $u_2$ $u_3$ $u_4''$
$g_4$ $u_1$ $u_2$ $u_3'$ $u_4$
$g_5$ $u_1$ $u_2$ $u_3'$ $u_4'$
$\phi$
$u_1$
$u_2$
$u_3$
$u_3'$
$u_4$ $u_4'$ $u_4''$ $u_4$ $u_4'$
$g_1$ $g_2$ $g_3$ $g_4$ $g_5$

(a) 5 of the graphs in $S_k$. (b) The matching tree for $S_k$.

Fig. 3. Illustration of a matching tree.

$g_1 = \langle u_1, \ldots, u_{k-1}, u_k \rangle$ and $g_2 = \langle u_1, \ldots, u_{k-1}, u'_k \rangle$. Then, when we compute the frequency of $g$ in a random graph $G' \in \mathcal{G}_r$ (i.e., $f(g, G')$), the size-$(k-1)$ candidate set would be the same as in the computation of the frequency of $f(g', G')$. In other words, the computation of $f(g, G')$ overlaps significantly with that of $f(g', G')$.

Generally, if two graphs in $S_k$ share a common prefix in their matching order, then the frequency estimation processes for the two graphs share a common component. A natural question is: How can we avoid redundant computation in the processing of such "similar" graphs? We answer this question with a method that carefully arranges the order in which we process the graphs in $S_k$.

Specifically, we first compute the matching order for every graph in $S_k$. After that, we organize all matching orders into a prefix tree, referred as the *matching tree*. Then, we perform a depth-first search (DFS) on the matching tree, and we process the graphs in $S_k$ in the order in which they are encountered during the DFS. We refer to such a sequence of graphs induced by the DFS as the *DFS order*.

For example, Figure 3 illustrates 5 graphs in an $S_k$ with $k = 4$, as well as a matching tree for $S_k$. The DFS order corresponding to the matching tree is $g_1, g_2, g_3, g_4, g_5, \ldots$. Given this DFS order, we avoid redundant computation in processing $g_i$ ($i \in [1, 5]$) as follows. First, given any random graph $G' \in \mathcal{G}_r$, we compute $g_1$'s size-2 candidate set $C_2$ and size-3 candidate set $C_3$. Based on $C_3$, we derive the occurrences of $g_1$ in $G'$, as well as those of $g_2$ and $g_3$, i.e., we avoid recomputing $C_3$ for $g_2$ and $g_3$. This is feasible since the matching orders of $g_1$, $g_2$, and $g_3$ share a common prefix $\langle u_1, u_2, u_3 \rangle$. After that, we reuse $C_2$ to the derive the size-3 candidate set $C'_3$ for $g_4$ (as $g_1$ and $g_4$ have a common prefix of length 2), and then utilize $C'_3$ to compute $f(g_4, G')$ and $f(g_5, G')$ in one batch.

In general, for any size-$i$ ($i \geq 2$) candidate set $C_i$ that corresponds to a node $p_i$ in the matching tree, we compute $C_i$ only once and reuse it for deriving the occurrences of any graph in $S_k$ that corresponds to a leaf node in the subtree under $p_i$. As such, we avoid all redundant computation of candidate sets. The downside of this approach is that it requires retaining $C_i$ until all graphs in the subtree of $p_i$ are processed, but this issue can be easily addressed with the divide-and-conquer method in Section VI-A. That is, we divide $C_i$ into several subsets and process each subset in turn. For each subset $C_i^{(j)}$, we derive the occurrences of all graphs depending on $C_i^{(j)}$, and transfer the relevant results to the main memory of the

TABLE I
SPECIFICATIONS OF E5645, Q2000, AND K20C.

| Name | # of cores | Core freq. | GPU memory | Price (USD)[3] |
|---|---|---|---|---|
| E5645 | 6 | 2400MHz | N/A | 513.39 |
| Q2000 | 192 | 625MHz | 1GB | 277.77 |
| K20 | 2496 | 706MHz | 5GB | 2695.00 |

machine. After that, we remove $C_i^{(j)}$ from the GPU's memory and use the freed space to process the next subset $C_i^{(j+1)}$.

## VII. EXPERIMENTS

### A. Experimental Settings

We implement our GPU-based algorithm for network motif discovery (dubbed *NemoGPU*) in C++ under Nvidia CUDA 5.5, and compare it against four state-of-the-art CPU-based algorithms: *Kavosh* [8], *QuateXelero* [10], *NetMode* [11], and *DistributedNM* [12]. We adopt the C++ implementations of *Kavosh*, *QuateXelero*, and *NetMode* made available by their respective inventors, and we implement *DistributedNM* in C++ with multi-core optimizations. All of our experiments are conducted on two machines with identical hardware and software configurations. In particular, each machine runs CentOS 5.0, and has 32GB main memory, an Intel Xeon E5645 CPU, a low-end Nvidia Quadro 2000 (Q2000) GPU, as well as a high-end Nvidia Telsa K20 GPU. Table I shows the specifications of the CPU and GPUs. We run *NetMode*, *DistributedNM*, and the CPU part of *NemoGPU* with 6 threads (i.e., one thread per CPU core), but *Kavosh* and *QuateXelero* with only one thread as their implementations do not support parallelism. The GPU part of *NemoGPU* is ran on Q2000 and K20 separately.

We use seven biological networks in our experiments, as shown in Table II. In particular, *Yeast* (YE) is the transcription network of yeasts; *H.sapiens* (HS) captures the protein-protein interaction (PPI) in the MINT dataset; *YeastPPI* (YP), *M.musculus* (MM), and *D.melanogaster* (DM) are the PPI networks of the budding yeast, fly genes, and mouse genes, respectively; *A.thaliana* (AT) describes the shared domains in Arabidopsis proteins; *C.elegans* (CE) represents the co-expression of worm genes. YE and YP are obtained from [22], while the other datasets are available from [23]. Our datasets are relatively small compared with those (with millions of nodes and edges) used in the literature of graph databases [19], [24], but we note that (i) biological networks are typically small, and (ii) identifying network motifs from our data is highly challenging due to the large number of random graphs that we need to process, and the huge number of subgraph isomorphism tests required. Furthermore, to our knowledge, YP is the largest dataset used in the previous work on network motif discovery [5]–[11]. In other words, the datasets that we use are up to one order of magnitude larger than those used in previous work, in terms of the numbers of nodes and edges.

Following previous work [6], [9], [11], [12], we vary $k$ (i.e., the size of the network motifs to be discovered) from 4 to 8, and set the default value of $r$ (i.e., the number of random graphs) to 1000. However, if an algorithm's running time exceeds 24 hours in an experiment, then we reduce $r$ to 100 for that particular algorithm in the experiment, and measure

[3]These prices were obtained from Amazon.com in August 2014.

TABLE II
DATASETS.

| Name | $\|V\|$ | $\|E\|$ | AVG deg. | MAX out-deg. | MAX in-deg. |
|---|---|---|---|---|---|
| YE | 688 | 1,079 | 3.14 | 71 | 13 |
| HS | 1,509 | 5,598 | 7.42 | 71 | 45 |
| YP | 2,361 | 6,646 | 5.63 | 64 | 47 |
| MM | 4,293 | 7,987 | 3.72 | 91 | 111 |
| DM | 6,303 | 18,224 | 5.78 | 88 | 122 |
| AT | 9,216 | 50,669 | 11.00 | 58 | 89 |
| CE | 17,179 | 124,599 | 14.51 | 67 | 107 |

its computation time $t_1$ (resp. $t_2$) for the first (resp. second) phase of network motif discovery; after that, we estimate the running time of the algorithm for $r = 1000$ as $t_1 + 10 \cdot t_2$. That said, if the algorithm does not terminate within 24 hours even when $r = 100$, then we omit it from the experiment. We repeat each experiment 3 times and report the average computation cost of each method.

### B. Comparisons with CPU-Based Techniques

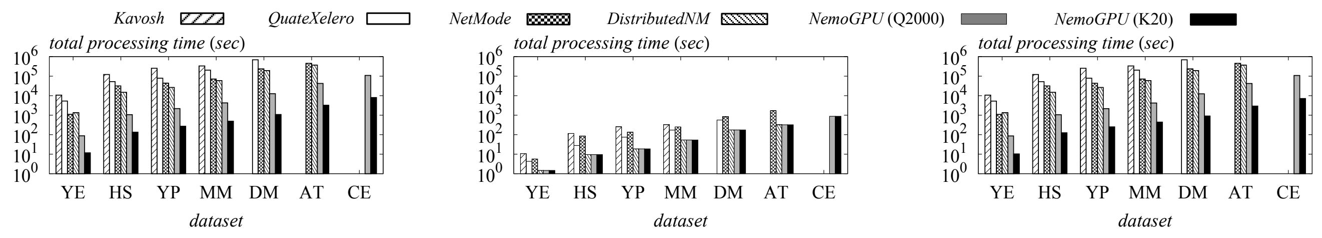
In the first set of experiments, we evaluate the computation time of all algorithms on all datasets, setting $k = 6$. Figure 4 illustrates the results. As shown in Figure 4a, our *NemoGPU* algorithm, when running with the high-end K20 GPU, outperforms all CPU-based methods by two orders of magnitude in terms of computation efficiency, regardless of the dataset used. When running with the low-end Q2000 GPU, *NemoGPU* is approximately ten times slower than with K20, as Q2000 has a much smaller GPU memory and considerably few GPU cores. However, even with Q2000, *NemoGPU* is still significantly more efficient than all CPU-based solutions. Among the CPU-based methods, *NetMode* and *DistributedNM* yield similar performance, with the latter slightly outperforming the former in most cases. Meanwhile, *Kavosh* and *QuateXelero* incur noticeably larger overheads than *NetMode* and *DistributedNM*.

Figure 4b and 4c illustrate the running time of each algorithm's first and second phases, respectively. Observe that, for all algorithms, the computation overhead of the first phase is negligible compared with that of the second phase, which justifies our choice of optimizing only the second phase in our GPU-based solution. The first phase of *NemoGPU* incurs exactly the same overhead as *DistributedNM* does, since they adopt the same CPU-based algorithm for the first phase, as mentioned in Section IV.

To more clearly illustrate the superiority of our GPU-based solution, we compute the *improvement ratio* of *NemoGPU* over *DistributedNM* (i.e., the most efficient CPU-based method), defined as the running time of the latter divided by that of the former. Figure 6a shows the improvement ratio of *NemoGPU* when $k = 6$, on all datasets except CE (as *DistributedNM* fails to terminate on CE). On the other hand, we also take into account the price differences among the CPU and GPUs, and compute the *performance-price ratio* of *NemoGPU*, defined as

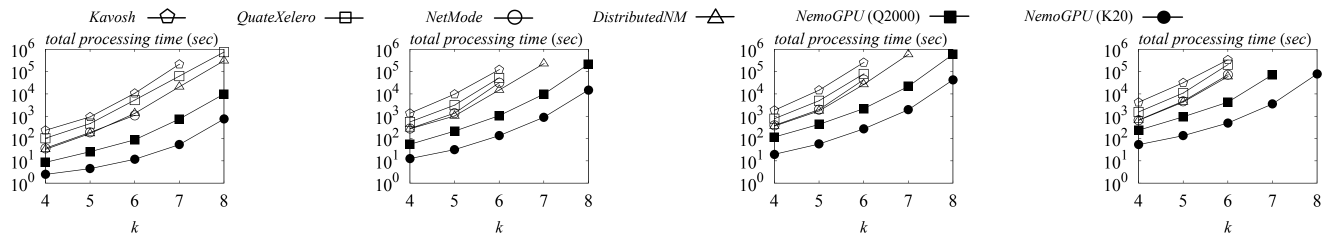
$$\text{Improvement ratio of } \textit{NemoGPU} \times \frac{\text{Price of the E5625 CPU}}{\text{Price of the GPU}}.$$

Figure 6b plots the performance-price ratio of *NemoGPU* with Q2000 and K20. The ratio for K20 (resp. Q2000) is up to 33 (resp. 27), and is above 18 (resp. 15) in all cases. This indicates that both K20 and Q2000 yield much higher "performance per



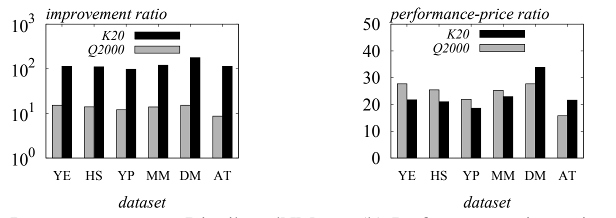
(a) Total running time. (b) Running time for the first phase. (c) Running time for the second phase.

Fig. 4. Computation time on all datasets.



(a) YE. (b) HS. (c) YP. (d) MM.

Fig. 5. Computation time vs. $k$.



(a) Improvements over DistributedNM. (b) Performance-price ratio.

Fig. 6. *NemoGPU* (with Q2000 and K20) vs. *DistributedNM*.

dollar" than the E5625 CPU does. In other words, if one is to improve the efficiency of network motif discovery, it is much more economical to invest in GPUs instead of CPUs.

Next, we evaluate the effects of $k$ on the efficiency of network motif discovery. Figure 5 shows the computation time of each algorithm as a function of $k$, using datasets YE, HS, YP, and MM. We omit DM, AT, and CE from this experiment, as all CPU-based methods incur prohibitive overheads on those datasets. In addition, we omit *NetMode* when $k > 6$, since it is exclusively designed for the cases when $k \leq 6$. As shown in Figure 5, our GPU-based solutions still outperform all CPU-based methods by large margins, regardless of the value of $k$. In particular, when running with K20, *NemoGPU* is more than 250 times faster than *DistributedNM* for $k \geq 7$.

## C. Effects of Optimizations

Finally, we evaluate the effects of the three optimization techniques proposed in Section VI: the divide-and-conquer (DC) method for handling large candidate sets, the graph merging (GM) technique for processing multiple random graphs simultaneously, and the matching tree (MT) approach for avoiding redundant computation. We consider three "crippled" versions of *NemoGPU* on K20: one with all three optimization disabled (denoted as *NA*), one with only DC enabled (denoted as *DC*), and one with only DC and GM enabled (denoted as *DC-GM*). For each crippled version of *NemoGPU*, we define its *relative overhead* on a dataset $D$ as its running time on $D$ divided by the running time of *NemoGPU* with all three optimizations enabled.

Figure 7 shows the relative overheads of *DC-GM*, *DC*, and *NA* on each dataset. *DC-GM*'s relative overhead is around 2 in all cases, which indicates that the MT optimization reduces the running time of *NemoGPU* by half. Meanwhile, the relative overhead of *DC* is roughly two times that of *DC-GM*, implying that the GM optimization improves the efficiency of *NemoGPU* by a factor of 2. On the other hand, *NA*'s relative overhead is slightly lower than that of *DC* on the two smallest datasets, YE and HS. The reason is that the pipelining approach employed by *DC* incurred additional costs in terms of the running time of *NemoGPU*. However, *NA* fails to handle any of the four larger graphs due to its excessive demand on the GPU's global memory. This shows that, although *DC* entails additional overheads, it is crucial for the scalability of *NemoGPU*. In summary, the three optimizations in Section VI improve the efficiency of *NemoGPU* by four-fold, and help scale *NemoGPU* to large graphs whose candidate sets do not fit in the GPU memory.

## VIII. Related Work

A plethora of CPU-based techniques [5]–[12], [25] have been proposed for network motif discovery. As discussed in Section III, those techniques typically utilize indices and complex algorithms to improve the efficiency of individual subgraph isomorphism tests in the frequency estimation phase, which make them unsuitable for GPU adoption. Furthermore, as we show in Section VII, our GPU-based solution significantly outperforms the state-of-the-art CPU-based methods in terms of both computation efficiency and cost-effectiveness.

Besides the aforementioned algorithms, there exist a number of CPU-based algorithms [26]–[28] for *approximate* network motif discovery. The basic idea is to heuristically sample subgraphs from the input graph $G$ and the random graphs in $\mathcal{G}_r$, and then identify motifs from the those samples. Those algorithms are generally more efficient than conventional methods for network motif discovery, but due to their heuristic nature, they fail to provide any quality guarantees on the results produced.

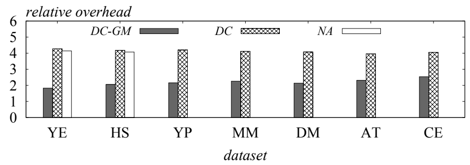


Fig. 7. Effects of optimization vs. dataset.

In addition, there are several recent studies [29], [30] on *subgraph listing*, i.e., identify the occurrences of a query graph $g$ in a large graph $G$. Although this problem is closely related to network motif discovery, the techniques in [29], [30] focus on the scenario where $G$ is a sizable graph (with billions of nodes and edges) that does not fit in the main memory of a single machine, and they employ distributed systems (e.g., MapReduce) to address the scalability issues that arise from this particular scenario. In contrast, in network motif discovery, we focus on the case where (i) $G$ is relatively small, but (ii) there exist a large number of random graphs whose subgraphs need to be compare with those in $G$. As a consequence, the techniques in [29], [30] are not suitable for network motif discovery.

Finally, there exist numerous techniques for *frequent subgraph mining* (see [24] for a survey) and *significant subgraph mining* [31]–[33], but those two problems are considerably different from network motif discovery. In particular, in frequent subgraph mining, we are given a set of graph $\mathcal{G}$, and we aim to identify the subgraphs that appear in a large portion of the graphs in $\mathcal{G}$, disregarding the number of occurrences of each subgraph $g$ in each individual graph. Similarly, *significant subgraph mining* also focuses on the *portion* of graphs in $\mathcal{G}$ where each subgraph $g$ appears, and it quantifies the significance of $g$ based on this portion instead of the frequency of $g$ in each graph in $\mathcal{G}$. Therefore, algorithms for frequent subgraph mining and significant subgraph mining are inapplicable for identifying network motifs.

## IX. Conclusions

This paper studies the problem of network motif discovery, and proposes the first GPU-based solution to the problem. Our solution is considerably different from the existing CPU-based method, due to the design choices that we make to exploit the strengths of GPUs in terms of parallelism and mitigate their limitations in terms of the computation power per GPU core. With extensive experiments on a variety of biological networks, we show that our solution is up to two orders of magnitude faster than the best CPU-based approach, and is around 20 times more cost-effective than the latter, when taking into account the monetary costs of the CPU and GPUs used. For future work, we plan to investigate how our solution can be extended to other network analysis tasks.

## Acknowledgement

Xiaokui Xiao was supported by a grant (ARC19/14) from MOE, Singapore and a gift from Microsoft Research Asia.

## References

[1] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298, no. 5594, pp. 824–827, October 2002.
[2] R. Sole and S. Valverde, "Are network motifs the sprandrels of cellular complexity?" *Trends Ecol. Evol.*, vol. 21, no. 8, pp. 419–422, 2006.
[3] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon, "Coarse-graining and self-dissimilarity of complex networks," *Phys. Rev. E*, vol. 71, p. 016127, 2005.
[4] O. Sporns and R. Ktter, "Motifs in brain networks," *PLoS Biol*, vol. 2, no. 11, p. e369, 10 2004.
[5] S. Wernicke and F. Rasche, "Fanmod: a tool for fast network motif detection," *Bioinformatics*, vol. 22, no. 9, pp. 1152–1153, 2006.
[6] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *RECOMB*, 2007, pp. 92–106.
[7] S. Omidi, F. Schreiber, and A. Masoudi-Nejad, "MODA: an efficient algorithm for network motif discovery in biological networks." *Genes & genetic systems*, vol. 84, no. 5, pp. 385–395, 2009.
[8] Z. R. M. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad, "Kavosh: a new algorithm for finding network motifs," *BMC Bioinformatics*, vol. 10, p. 318, 2009.
[9] P. M. P. Ribeiro and F. M. A. Silva, "g-tries: an efficient data structure for discovering network motifs," in *SAC*, 2010, pp. 1559–1566.
[10] S. Khakabimamaghani, I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad, "Quatexelero: An accelerated exact network motif detection algorithm," *PLoS ONE*, vol. 8, no. 7, p. e68073, 07 2013.
[11] X. Li, D. S. Stones, H. Wang, H. Deng, X. Liu, and G. Wang, "Netmode: Network motif detection without nauty," *PLoS ONE*, vol. 7, no. 12, p. e50093, 12 2012.
[12] P. M. P. Ribeiro, F. M. A. Silva, and L. M. B. Lopes, "Parallel discovery of network motifs," *JPDC*, vol. 72, no. 2, pp. 144–154, 2012.
[13] T. Wang, J. W. Touchman, W. Zhang, E. B. Suh, and G. Xue, "A parallel algorithm for extracting transcription regulatory network motifs," in *BIBE*, 2005, pp. 193–200.
[14] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *J. Symb. Comput.*, vol. 60, pp. 94–112, 2014.
[15] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *JPDC*, vol. 68, no. 10, pp. 1381–1388, 2008.
[16] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *DaMoN*, 2009, pp. 34–42.
[17] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *GPGPU*, 2011, p. 3.
[18] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *ICDM*, 2002, pp. 721–724.
[19] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD Conference*, 2013, pp. 337–348.
[20] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.
[21] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, vol. 1, no. 1, pp. 364–375, 2008.
[22] http://lbb.ut.ac.ir/Download/LBBsoft/QuateXelero/networks/.
[23] http://genemania.org.
[24] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *Knowledge Eng. Review*, vol. 28, no. 1, pp. 75–105, 2013.
[25] J. Chen, W. Hsu, M.-L. Lee, and S.-K. Ng, "Nemofinder: dissecting genome-wide protein-protein interactions with meso-scale network motifs," in *KDD*, 2006, pp. 106–115.
[26] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon, "Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs," *Bioinformatics*, vol. 20, no. 11, pp. 1746–1758, 2004.
[27] I. Bordino, D. Donato, A. Gionis, and S. Leonardi, "Mining large networks with subgraph counting," in *ICDM*, 2008, pp. 737–742.
[28] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," in *ISMB*, 2008, pp. 241–249.
[29] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *ICDE*, 2013, pp. 62–73.
[30] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *SIGMOD Conference*, 2014, pp. 625–636.
[31] S. Ranu and A. K. Singh, "Graphsig: A scalable approach to mining significant subgraphs in large graph databases," in *ICDE*, 2009, pp. 844–855.
[32] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *SIGMOD Conference*, 2008, pp. 433–444.
[33] H. He and A. K. Singh, "Graphrank: Statistical modeling and mining of significant subgraphs in the feature space," in *ICDM*, 2006, pp. 885–890.