Network Motif Discovery: A GPU Approach

Lin, Wenqing; Xiao, Xiaokui; Xie, Xing; Li, Xiao-Li

2016

Lin, W., Xiao, X., Xie, X., & Li, X.-L. (2017). Network Motif Discovery: A GPU Approach. IEEE Transactions on Knowledge and Data Engineering, 29(3), 513-528.

https://hdl.handle.net/10356/81386

https://doi.org/10.1109/TKDE.2016.2566618

Network Motif Discovery: A GPU Approach

Wenqing Lin, Xiaokui Xiao, Xing Xie, Xiao-Li Li

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 (such as branch divergences and memory coalescing) 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 in how it enumerates subgraphs, 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.

Index Terms—Network Motif, GPU, Graph Mining, Algorithms.

1 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]. Identifying 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], [6], [7], [8], [9], [10], [11], [12], [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. In addition, even the subgraph enumeration phase could incur substantial overheads, as it requires enumerating a large number of subgraphs from G and eliminating duplicates by checking subgraph isomorphisms. Existing techniques attempt to resolve these issues by improving the efficiency of subgraph isomorphism tests, but only achieve limited success. As shown in Section 7, 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. 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 2.3, existing CPU-based algorithms for subgraph isomorphism tests cannot be translated into efficient solutions on GPUs, because 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 diver-

- *Wenqing Lin and Xiao-Li Li are with Data Analytics Department of the Institute for Infocomm Research, A*STAR, Singapore (e-mail: {linw, xlli}@i2r.a-star.edu.sg).*
- *Xiaokui Xiao is with School of Computer Science and Engineering, Nanyang Technological University, Singapore (e-mail: xkxiao@ntu.edu.sg).*
- *Xing Xie is with Microsoft Research Asia (e-mail: xing.xie@microsoft.com).*

gences in GPU codes, and memory access patterns on the GPU (see Section 2.2). Based on those considerations, we make design choices that drastically differ from existing CPU-based methods but 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 not 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 network motif discovery, and make the following contributions:

- We analyze the deficiency of 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 3, 4, and 6)
- 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 5)
- We empirically compare our solution against the state-of-the-art CPU-based methods, using the largest datasets ever tested in the literature.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 widened 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 7)

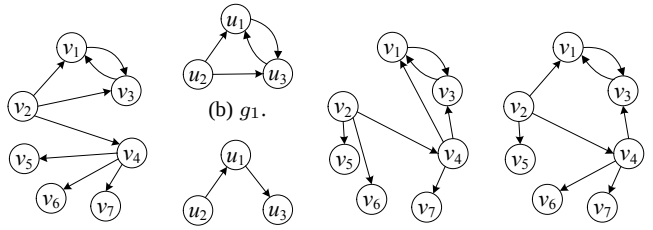
2 PRELIMINARIES

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

2.1 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. We say that a vertex u' is a neighbor of u if u' is an in-neighbor or out-neighbor of u, and denote $nbr(u)$ as the set of neighbors of u. We say that there is a *vertex order* o on V if for any $u, u' \in V$ we have $o(u) > o(u')$ or $o(u) < o(u')$.

1. It is a standard assumption in the literature to consider unlabelled graphs. But our solution can be extended to handle labelled graphs (see Section 6.3).



(a) G. (b) g_1. (c) g_2. (d) G_1. (e) G_2.

Fig. 1. An input graph G, two subgraphs g_1 and g_2 of G, and two random graphs G_1 and G_2 degree-equivalent to G.

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 bijective function ζ that maps V_g to a subset V_{sub} of V, such that for any edge $(u, v) \in E_g$, there is an edge $\big(\zeta(u), \zeta(v)\big) \in E$, and vice versa[2]. 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. 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 $\psi : V \rightarrow V'$, 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 θ, a subgraph g of G is a* **motif** *of G, if and only if $\widetilde{\sigma}(g) > 0$ and*

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

Problem Statement. Given G, $r > 0$, $k > 2$, and $\theta > 0$, the network motif discovery problem 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).

2.2 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 2 shows the general architecture

2. Note that this definition of subgraph slightly differs from the common definition, which does not require every edge between two nodes in V_g to be mapped to an edge in g. However, we adopt the former definition since it is standard in literature on network motif discovery.

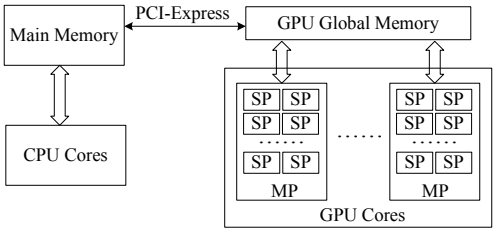


Fig. 2. The general architecture of a GPU.

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.

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 (unless the CPU and GPU are integrated and share the same 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.

2.3 Existing CPU-Based Methods

As mentioned in Section 1, existing CPU-based methods [5], [8], [9], [10], [11], [12], [13] typically run in two steps:

1) *Subgraph enumeration*: Compute the set $\mathcal{S}_k$ 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 $g \in \mathcal{S}_k$ 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.

The above two phases highly rely on the *canonical labeling* [14] of subgraphs, 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. However, it is NP-hard to compute the canonical labeling of a graph [14]. To alleviate this overhead, existing methods construct indices on subgraphs in $\mathcal{S}_k$ for efficient subgraph search and mapping the enumeration of each subgraph to its canonical labeling.

The aforementioned CPU-based methods 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 2.2).

Second, CPU-based methods rely on the index structures to check whether a subgraph appears in $\mathcal{S}_k$. If we adopt the same approach on a GPU, we need to store the indices in the GPU's global memory, and ask each GPU thread to probe the indices for subgraph matching. Then, the GPU threads in each warp are likely to access non-consecutive memory locations, which prevents the GPU from applying memory coalescing to reduce access costs. Furthermore, when G is large, the indices can become so large that they do not fit in the GPU's memory (typically a few GBs).

Finally, if we are to ask each GPU thread to examine whether a subgraph appears in $\mathcal{S}_k$, 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.

3 Solution Overview

As with the existing CPU-based methods, our solution also consists of a subgraph enumeration phase and a frequency estimation phase. In particular, the subgraph enumeration phase of our solution adopts a CPU-based method to compute the canonical

Algorithm 1: *FreqComp*

input : $g \in \mathcal{S}_k$ 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 4.2);
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** $i = 3, \ldots, k$ **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** $|C_k|$;

labeling for each adjacency matrix, 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. However, for all other parts of the subgraph enumeration and frequency estimation phases (i.e., the enumeration of vertex combinations and the computation of the average frequency of each subgraph $g \in \mathcal{S}_k$ with respect to $\mathcal{G}_r$), we employ GPU-based algorithms that provide much higher efficiency than existing CPU-based methods. The reason that we optimize those parts of the computation is that they incur significantly higher overheads (i.e., over 99% of processing time) than the other components, due to the large number of vertex combinations and random graphs to be processed. In what follows, we present an overview of our GPU-based algorithm for the frequency estimation phase, assuming that G, $\mathcal{S}_k$, and $\mathcal{G}_r$ are given. For ease of exposition, we defer the the details of our solution for the subgraph enumeration phase to Section 6, as it utilizes several techniques that are devised for the frequency estimation phase.

In a nutshell, our algorithm for the frequency estimation phase examines each pair of g and G' where $g \in \mathcal{S}_k$ 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*. 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 $k - 2$ iterations (Lines 4-6), each of which 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).

Compared with the existing CPU-based methods [5], [8], [9], [10], [11], [12], [13], *FreqComp* does not compute any canonical labeling or construct any index on $\mathcal{S}_k$, 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.

It is noteworthy that *FreqComp* is similar in spirit to existing CPU-based algorithms [18], [19], [20], [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], [19], [20], [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], [19], [20], [21], as we demonstrate in Section 4.

4 GPU-Based Subgraph Matching

This section presents the details of the *FreqComp* algorithm. Section 4.1 clarifies how we represent each random graph G' in a GPU's global memory. Sections 4.2 and 4.3 elaborates each step of *FreqComp*. Section 4.4 proves *FreqComp*'s correctness.

4.1 Representation of Graphs

We represent each random graph $G' \in \mathcal{G}_r$ in the Compressed Sparse Row (CSR) format [22]. Instead of using only two arrays, for fast retrieving, we use 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.

4.2 Construction of C_i

As mentioned in Section 3, 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 $\{u_1, \ldots, u_i\}$. For ease of exposition, we defer the discussion of the matching order to Section 4.2. 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 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

Algorithm 2: *BuildCi*

input : g, G', and C_{i-1}
output: C_i

1 $\{A_o, A_v\}$ = *InitArray* (g, C_{i-1}); // (Algorithm 3)
2 $\{I, A'_o\}$ = *GenCand*(A_o, A_v, g, G', C_{i-1}); // (Algorithm 4)
3 C_i = *RefineCand*(I, A'_o); // (Algorithm 5)
4 **return** C_i;

three cases, C_2 can be constructed by inspecting each element once in 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 the 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 4.3 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 $C_3, C_4, \ldots, C_k$. 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 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 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_α ($\alpha \in [1, i-1]$) in g. Then, for each vertex v in G' that is an out-neighbor of v_α, we create $|E(v)|$ GPU threads, where $E(v)$ is the set of all edges incident to v in G'. (We refer to v_α as the *anchor vertex*.) The ℓ-th thread examines the ℓ-th edge $e' \in E(v)$, and checks whether the following *edge validity conditions* hold simultaneously:

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_α in c_x with the smallest non-zero relevant degree;
5 $A_{deg}[x] \leftarrow$ the relevant degree of v_α;
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;

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

The thread returns *true* if all of the above conditions hold, and *false* otherwise. (See Lines 10-13 in Algorithm 5.) 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.

It remains to discuss how we select the anchor vertex v_α ($\alpha \in [1, i-1]$) 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_α to the vertex in c with the smallest *non-zero* relevant degree, which is computed by invoking a GPU thread for each size-i candidate to inspect the vertices in it.

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

Algorithm 4: *GenCand*

input : A_o, A_v, g, G', and C_{i-1}
output: I and A'_o

1 let θ be the last element of A_o;
2 create arrays A'_{deg} and I, both of the size θ;
3 **for** *each* $y = 1, 2, \ldots, \theta$ **in parallel do**
4 identify the integer x such that $A_o[x] \leq y < A_o[x+1]$;
5 let c_x be the x-th graph in C_{i-1};
6 let v_α be the vertex recorded in $A_v[x]$;
7 let $z = y - A_o[x]$;
8 $v \leftarrow$ the z-th vertex in v_α'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;

from A_v the anchor vertex v_α in c_x. For each vertex v in the relevant neighbor set of v_α, *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 $\langle v_1, \ldots, v_{i-1}, v \rangle$ in I, *RefineCand* creates $|E(v)|$ GPU threads, each of which (i) identifies an vertex v' in $E(v)$ as well as the edge e' connecting v and v' (Lines 5-9), (ii) checks whether the edge e' of v satisfies all edge validity conditions (Lines 10-13), and (iii) writes the result of the check into an array B (Line 14). Then, *RefineCand* examines B to identify those vertices v that have $|E(u_i)|$ edges passing the validity check, and writes the results into an array B^* of size $|I|$ (Lines 15-18). For each such v, it inserts $\langle v_1, \ldots, v_{i-1}, v \rangle$ into C_i as a size-i candidate (Lines 19-24). After that, the algorithm returns C_i and terminates.

Although Algorithm 5 has a few *if* statements, they do not lead to branch divergence since none of them comes with an *else* statement. Specifically, the threads on which the *if* statement is false would remain idle, without incurring any overhead. In addition, although Algorithm 3-5 have a few loops, every thread executes exactly the same number of iterations and performs the same operations per iteration, which avoids branch divergence and workload imbalance. Finally, we note that the threads run in a *bulk synchronous* manner, i.e., they process different data and write the outputs to separate memory locations, which simplifies synchronization (see Lines 15-20 in Algorithm 5 for example).

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], [19], [20], [21], [23]. 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.

Algorithm 5: *RefineCand*

input : I and A'_o
output: C_i

1 let γ 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 θ' 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 $A'_o[y] \leq z < A'_o[y+1]$;
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 ℓ-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* β*-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 θ^* be the last element of A^*_o;
21 create an array C_i of size θ^*;
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;

4.3 Avoiding Duplicates

Let g'_i be an occurrence of g_i in G'. As mentioned in Section 4.2, 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* [6]. An automorphism group of g is a set A of ordered pairs (u, u') such that (i) u and u' are vertices in g, (ii) the vertex ID of u is smaller than that of u', and (iii) 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'.

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. 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 $1 \leq x < y \leq i$, 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[3] in C_k, and ensures that the frequency of g

3. 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 $g(i)$.

can be correctly computed from C_k.

In our solution, we compute an SCS Q for g using the CPU-based algorithm in [6], which also includes the algorithm to compute the canonical labelings. Then, we impose the symmetry constraint 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.

4.4 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 4.3, 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}. □

5 Optimizations

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

5.1 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 hard disk, 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 λ of the available memory on the GPU. (We will discuss the setting of λ 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 available 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 is 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 available 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 available 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 λ fraction of the GPU memory available after the first $j-1$ parts are assigned, except that the last part utilizes all remaining GPU memory. To choose an appropriate value for λ, 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 λ, and we derive the λ 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 (e.g., more than 10% of the total processing cost), which leads to inferior efficiency.

Let M be the amount of available 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 λM GPU memory for the conversion from C_i to C_{i+1}, the number of subsets of C_i generated is $O\left(\frac{|C_i|}{\lambda M}\right)$. By the same reasoning, the number of subsets of C_j ($j > i$) produced is 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 proportional to:

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

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.

5.2 Handling Multiple Graphs

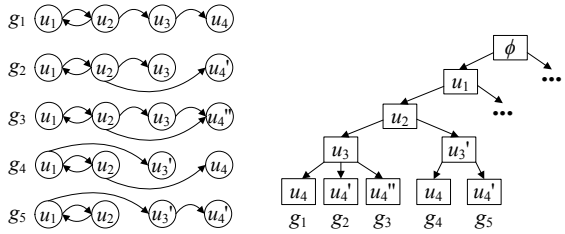
Our previous discussions have focused on computing subgraph frequencies in one random graph $G' \in \mathcal{G}_r$. When G' contains relatively small numbers (e.g., 100) of vertices and edges, the frequency computation processes on G' may not engage all GPU cores, which leads to under-utilization of 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 μ graphs, where μ 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 μ 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 5.1, and leads to inferior efficiency. To tackle the problem, we choose μ using a heuristic method as follows. 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 τ of the amount of GPU memory required in the conversion from C_2 to C_3, and we set μ to the maximum integer such that the upperbound τ is no more than $\frac{1}{k-2}$ fraction of the available 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 5.1, which helps avoid overloading the GPU. Our experiments show that this choice of μ leads to good computation efficiency. (See Figure 9 in Section 7.3.)

5.3 Matching Tree

Let g and g' be two size-k subgraphs of G (i.e., $g, g' \in \mathcal{S}_k$) 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., $g_1 = \langle u_1, \ldots, u_{k-1}, u_k \rangle$ 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 that in the computation 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 $\mathcal{S}_k$ 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 $\mathcal{S}_k$. Specifically, we first compute the matching order for every graph in $\mathcal{S}_k$. 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 $\mathcal{S}_k$ 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*.



(a) 5 of the graphs in $\mathcal{S}_k$. (b) The matching tree for $\mathcal{S}_k$.

Fig. 3. Illustration of a matching tree.

For example, Figure 3 illustrates 5 graphs in an $\mathcal{S}_k$ with $k = 4$, as well as a matching tree for $\mathcal{S}_k$. The DFS order corresponding to the matching tree is $g_1, g_2, g_3, \ldots$. 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', along with 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 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 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 corresponding 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 $\mathcal{S}_k$ 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 p_i's subtree are processed; but this can be easily addressed with the divide-and-conquer method in Section 5.1. 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 machine. After that, we remove $C_i^{(j)}$ from the GPU's memory and use the freed space to process the next subset $C_i^{(j+1)}$.

6 Subgraph Enumeration with GPUs

In the previous sections, we have focused on the frequency estimation phase of network motif discovery, which incurs the majority of the overall computation cost when the number r of random graphs is large (e.g., $r = 1000$). However, existing work [1], [8], [9], [24] suggests that one may also use a moderate value for r (e.g., $r = 100$) to trade accuracy for efficiency. In that case, the computation cost of the subgraph enumeration phase could become significant. Motivated by this, this section presents a GPU-based method for the subgraph enumeration phase that is an order of magnitude faster than the existing CPU-based algorithms.

6.1 Algorithmic Framework

Existing CPU-based algorithms for the subgraph enumeration phase typically run in three steps:

1) Enumerate every unique k-vertex combination in G that induces a connected subgraph. (We refer to such vertex combinations as *valid k-combinations*.)
2) Compute the adjacency matrix of each valid k-combination, and then derive the set $\mathcal{M}_k$ of unique adjacency matrices, as well as the frequency of each adjacency matrix in $\mathcal{M}_k$.
3) Compute the canonical labeling of each adjacency matrix in $\mathcal{M}_k$, and then derive the set $\mathcal{S}_k$ of unique canonical labelings, as well as the frequency of each canonical labeling in $\mathcal{S}_k$.

Through experiments, we observe that Steps 1 and 2 dominate the computation cost of the subgraph enumeration phase. (For example, in the state-of-the-art CPU-based solution [12], Steps 1 and 2 account for over 99% of the processing time in subgraph enumeration.) The reason for this is two fold. First, there exist a large number of valid k-combinations in G (especially when k is large), due to which Steps 1 and 2 have to process a significant amount of data. Second, although the valid k-combinations are large in number, their adjacency matrices often duplicate each other, because of which the set $\mathcal{M}_k$ produced by Step 2 is relatively small (e.g., $|\mathcal{M}_k| = 100$). In turn, a small $\mathcal{M}_k$ leads to a small processing cost in Step 3, since the number of canonical labelings computed in Step 3 equals $|\mathcal{M}_k|$.

Based on the above observation, we devise a GPU-based solution to accelerate Steps 1 and 2, so as to mitigate the major overhead of the subgraph enumeration phase. On the other hand, we adopt a CPU-based method [12] for Step 3 because (i) the computation time of Step 3 is insignificant in comparison to Steps 1 and 2, and (ii) as we explain in Section 2.3, it is difficult to compute canonical labelings on GPU efficiently due to branch divergence issues.

The basic idea of our solution is as follows. First, we transfer the input graph $G = (V, E)$ to the GPU global memory and store it using six arrays E_{out}, E_{in}, E_{bi}, O_{out}, O_{in}, and O_{bi}, as described in Section 4.1. We also create two additional arrays O_{nbr} and E_{nbr} in the GPU memory, such that for each vertex u in V, E_{nbr} stores the set $nbr(u)$ of u's neighbors and O_{nbr} records the position of the first neighbor of u in E_{nbr}. Furthermore, the vertices in $nbr(u)$ are stored consecutively in E_{nbr}, and are ordered by their vertex IDs.

Second, we enumerate the valid k-vertex combinations in G using an incremental approach. Specifically, we first generate the set of all valid 1-combinations (i.e., each combination contains a distinct vertex in G), and then iteratively compute the valid i-combinations ($i \in [2, k]$) by adding a vertex to each of the valid $(i-1)$-combinations. At the first glance, this approach may look similar to our subgraph matching method in Section 4, as they both inspect subgraphs of a given graph incrementally. However, the search space tackled by the two techniques are drastically different. Specifically, the subgraph matching method aims to identify the subgraphs that are isomorphic to a query graph g, and hence, it can omit a subgraph as soon as it finds that the subgraph contains a vertex that cannot be matched to any vertex in g. In contrast, when we enumerate k-combinations, we are not given a query graph that can be used for pruning; instead, we need to identify *every* possible k-combination that is valid. This leads us to devise a different algorithm for enumerating k-combinations, which we elaborate in Section 6.2.

Third, once we obtain the valid k-combinations in G, we compute the adjacency matrix of each of them, and then perform a *group-by* of the adjacency matrices to derive the set $\mathcal{M}_k$ on the GPU. After that, we transfer $\mathcal{M}_k$ back to CPU main memory, and compute the canonical labeling for each adjacency matrix in $\mathcal{M}_k$ in parallel, which results in the set $\mathcal{S}_k$. We present the details of this step in Section 6.3.

6.2 Generation of Vertex Combinations

Suppose that we have computed the set T_i of all valid i-combinations in G ($i \in [1, k]$), and we are to derive the set T_{i+1} of all valid $(i+1)$-combinations based on T_i. Towards this end, we aim to identify, for each vertex combination $t \in T_i$, a set $X(t)$ of vertices such that each $v \in X(t)$ can be added into t to produce a valid $(i+1)$-combination. We refer to $X(t)$ as the *extension set* for t. A naive approach to compute $X(t)$ is to insert into $X(t)$ the neighbors of each vertex in t. This approach, however, leads to duplicate k-combinations because there could exist two i-combinations t_1 and t_2 as well as two vertices v_1 and v_2, such that $t_1 \cup \{v_1\}$ and $t_2 \cup \{v_2\}$ result in the same valid $(i+1)$-combination. One may note that, in Section 4.3, we have addressed a similar issue about duplicates that arises in the frequency estimation phase, where we need to enumerate subgraphs that match a given graph g. However, our solution there relies on the automorphism groups of g, which makes it inapplicable in the current context as we no longer have a query graph g to exploit.

To tackle the above problem, we devise a new technique to construct the extension set $X(t)$ for any i-vertex combination t without incurring duplicates, as we explain in the following. First, we require that each vertex in $X(t)$ should not only be a neighbor of a vertex in t, but also have a larger vertex ID than the *first vertex of* t, which is defined as the vertex in the 1-combination that t originates from. For example, suppose that a valid 3-combination $t = \langle u_1, u_2, u_3 \rangle$ is obtained by first adding u_2 into a 1-vertex combination $\langle u_1 \rangle$, and then adding u_3. Then, u_1 is the first vertex of t. Similarly, we refer to u_2 and u_3 as the second and the third vertices of t, respectively. The above constraint on vertex IDs ensures that if two i-combinations t_1 and t_2 have different first vertices, then they are guaranteed to be distinct.

Second, we require that any node in $X(t)$ should have a lower *composite rank* than all nodes in t. In particular, for each $t \in T_i$ and for each vertex v that is neighbor of a node in t, we define the composite rank of v as an ordered pair $\langle r, id \rangle$, where id denotes the vertex ID of v, and r is the smallest integer in $[1, i]$ such that the r-th vertex of t is adjacent to v. Let v_1 and v_2 be two vertices with composite ranks $\langle r_1, id_1 \rangle$ and $\langle r_2, id_2 \rangle$, respectively. We say that v_1 has a higher composite rank than v_2 (denoted as $v_1 < v_2$), if and only if $\langle r_1, id_1 \rangle < \langle r_2, id_2 \rangle$, namely (i) $r_1 < r_2$, or (ii) $r_1 = r_2$ and $id_1 < id_2$. In other words, we rank v_1 and v_2 based on their neighbors in t, and we break ties based on vertex IDs. The following lemma shows the correctness of our approach for constructing extension sets.

Lemma 2. *Let $T_i(u)$ be the set of all valid i-combinations that share the first vertex u, and $X(t)$ be the extension set of any $t \in T_i(u)$ constructed based on composite ranks. Then, for any $t_1, t_2 \in T_i(u)$, any $v_1 \in X(t_1)$, and $v_2 \in X(t_2)$, we have $t_1 \cup \{v_1\} \neq t_2 \cup \{v_2\}$. In addition, for any $t^* \in T_{i+1}$, there exist $t \in T_i$ and $v \in X(t)$, such that $t \cup \{v\} = t^*$.*

Proof. By contradiction, assume that there exists a size-k subgraph g of G such that g has two different k-vertex combinations $t = \langle u_1, u_2, \ldots, u_k \rangle$ and $t' = \langle u'_1, u'_2, \ldots, u'_k \rangle$ enumerated by the algorithm. In other words, t and t' share the same set of vertices, but there exists a $j \in [1, k]$ such that $u_j \neq u'_j$.

Since (i) the first vertex of t (resp. t') is the one with the smallest vertex ID in t (resp. t'), and (ii) t and t' share the same set of vertices, we have $u_1 = u'_1$. Let S (resp. S') be the set of neighbors of u_1 (resp. u'_1) in t (resp. t'). Then, we have that (i) $S = S'$, and (ii) u_2 (resp. u'_2) is the one with the lowest composite rank in S (resp. S'). Therefore, $u_2 = u'_2$. By induction, we have that $u_j = u'_j$ for each $j \in [1, k]$, which contradicts the assumption and completes the proof. □

TABLE 1
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

6.3 Processing of Adjacency Matrices

After the set T_k of all valid k-combinations is generated, we compute the adjacency matrix for each vertex combination $t \in T_k$ and store the matrix in k^2 bits. Then, we invoke a GPU-based group-by algorithm [25] to compute the set $\mathcal{M}_k$ of distinct adjacency matrices, as well as their numbers of occurrences.

To generate the adjacency matrices, a straightforward approach is to create one GPU thread for each vertex combination $t \in T_k$, and then ask the thread to check the adjacency of every pair of vertices in t and write the result in k^2 bits. This method, however, is highly inefficient as it causes workload imbalance. In particular, when a GPU thread checks whether two vertices are adjacent, it would have to access the edge list of either vertex. Meanwhile, the edge lists of different vertices could be drastically different in size. Given that different GPU threads handle different vertex combinations, it is likely that some threads would need to process much larger edge lists than the others, which leads to a workload imbalance and degrades efficiency.

To avoid the above deficiency, we generate the adjacency matrices of the k-combinations in T_k with a different technique that runs in $k-1$ rounds, such that the i-th round ($i \in [1, k-1]$) checks, for each k-combination t, whether the i-th vertex u_i in t is adjacent to the j-th vertex u_j ($j \in [i+1, k]$). In particular, in the i-th round, we first create one GPU thread for each t to calculate the size of u_i's neighbor set $nbr(u_i)$. After that, we create $|nbr(u_i)|$ GPU threads for t, such that the x-th thread checks if the x-th neighbor of u_i appears in t by scanning the k vertices in t. As such, each GPU thread processes an equal amount of data, thus avoiding workload imbalances.

The above discussions assume that the set T_k of all valid k-combinations fits in the global memory of the GPU. In case that this assumption does not hold, we adopt the divide-and-conquer approach in Section 5.1 to pipeline the generation of $\mathcal{M}_k$. For example, suppose that we have sufficient GPU memory to accommodate $T_1, T_2, \ldots, T_i$, but not T_{i+1}. In that case, after we generate T_i, we divide the available GPU memory into $k-i+1$ parts. We use the j-th part ($j \in [1, k-i]$) as the buffer for the generation of T_{i+j}, and the last part as the buffer for computing adjacency matrices and performing group-by. Whenever a batch of group-by results is generated, we transfer them from the buffer back to the main memory, and then clear the buffer to make room for subsequent computation. Note that the partial group by results would need to be postprocessed by the CPU since one adjacency matrix may appear in different batches of group-by results.

Remark. Although we have focused on unlabeled graphs, our solution can be extended to handle labeled graphs, with a few changes as follows. Given a graph with labels on vertices and edges, we modify the graph representation in Section 4.1 to include two additional arrays that store the node labels and edge labels, respectively. Then, in the frequency estimation phase, we add one additional requirement in the vertex validity condition: for all $j \in [1, i-1]$, the label of v_j equals the label of u_j. Similarly, the edge validity conditions are extended to require that, if e' is an outgoing edge from v, then the label of e' should equal the label of the edge from u to u_j; meanwhile, if e' is an incoming edge to v, then the label of e' should equal the label of the edge from u_j to u. Furthermore, in the subgraph enumeration phase, given all valid k-combinations, we utilize an existing technique [26] to encode each combination into a tuple with three values that represent its structure, edge labels, and vertex labels, respectively. The remaining parts of the our technique remain the same.

TABLE 2
Specifications of E5645, Q2000, and K20C.

Name	# of cores	Core freq.	GPU memory	Price (USD)[4]
E5645	6	2400MHz	N/A	513.39
Q2000	192	625MHz	1GB	277.77
K20	2496	706MHz	5GB	2695.00

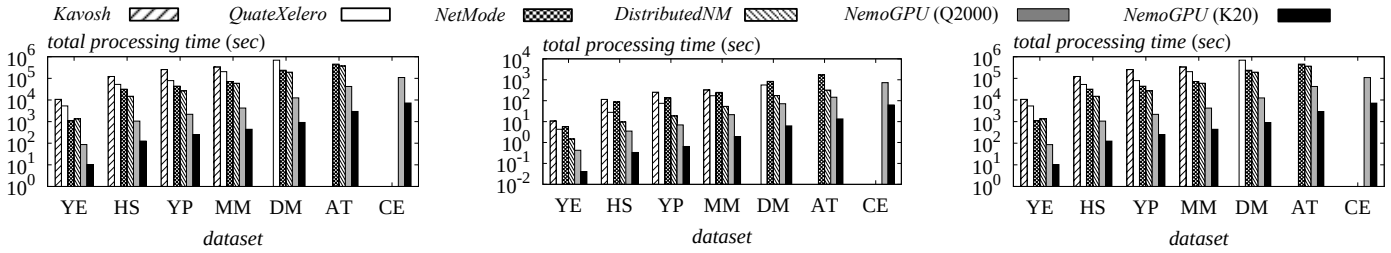
7 Experiments

7.1 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 Tesla K20 GPU. Table 2 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. We run the GPU part of *NemoGPU* on Q2000 and K20 separately. Note that *NemoGPU* requires preprocessing the input graph to convert it into the CSR format mentioned in Section 4.1; we include the conversion cost as part of *NemoGPU*'s computation overhead.

We use seven biological networks in our experiments, as shown in Table 1. In particular, *Yeast* (YE) is the transcription network of yeasts; *H.sapiens* (HS) captures the human 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 and mouse, 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 [27], while the other datasets are available from [28]. Our datasets are relatively small compared to those (with millions of nodes and edges) used in the literature of graph databases [19], [29], 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 already the largest dataset used in the previous work on network

4. These prices were obtained from Amazon.com in August 2014.



(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.

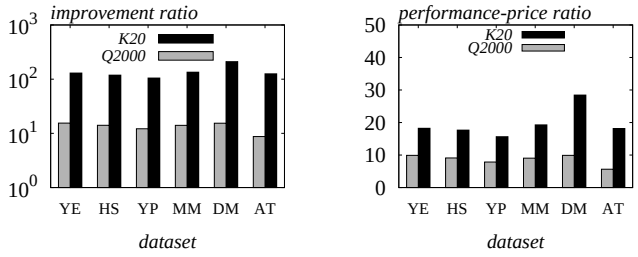
motif discovery [5], [6], [7], [8], [9], [10], [11], and yet, our largest dataset CE is around an order of magnitude larger than YP in terms of the numbers of nodes and edges.

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 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. In each experiment, we measure the running time of a technique by taking into account all costs incurred, e.g., the CPU-GPU data transfer cost. We repeat each experiment 3 times and report the average computation cost of each method.

7.2 Comparisons with CPU-Based Techniques

In the first set of experiments, we demonstrate the superiority of the GPU-based solution by evaluating the computation time of all the algorithms on all the datasets, setting $k = 6$. Figure 4 shows 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. This is mainly because *NemoGPU* is able to leverage the thousands of computation cores in the K20 GPU, whereas the CPU-based methods can only utilize up to 6 CPU threads. 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, as the number of cores in Q2000 is still significantly larger than that in an E5645 CPU. 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*, since the former exploits only one CPU thread while the latter exploits multi-threading.

Figure 4b and 4c illustrate the running time of each algorithm's first and second phases, respectively. Observe that *NemoGPU* significantly outperforms the CPU-based approaches in both phases, regardless of the GPU used. In particular, *NemoGPU* on K20 is up to 35 times faster than *DistributedNM* (i.e., the most efficient CPU-based method) in the first phase, since *NemoGPU* parallelizes the generation of vertex combination (i.e., the most intensive tasks in the first phase) with thousands of threads. Besides, *NemoGPU* on K20 is up to 210 times faster than *DistributedNM* in the second phase, which again demonstrates the



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

Fig. 5. *NemoGPU* (with Q2000 and K20) vs. *DistributedNM*.

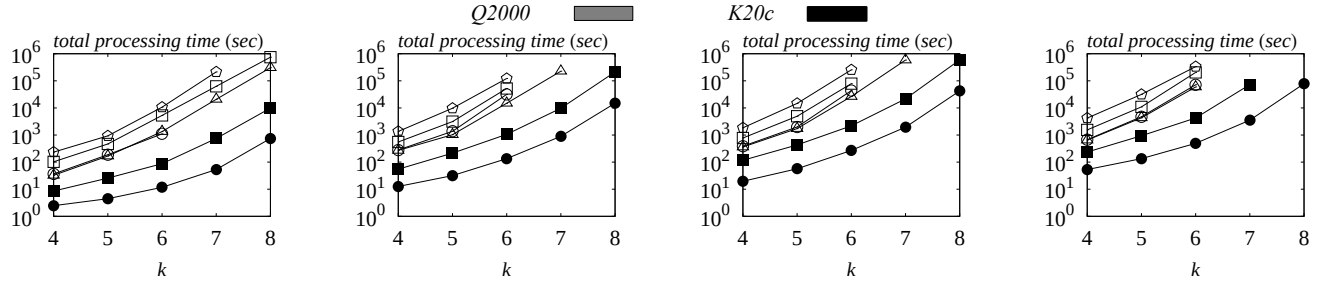
superiority of GPU parallelism. Note that, *NemoGPU* achieves more performance gain in the second phase than in the first phase, since *NemoGPU* does not need to compute canonical labelings and can process multiple graphs simultaneously in the second phase. Meanwhile, *NemoGPU* on Q2000 improves over *DistributedNM* by more than 2 times in the first phase, and up to 15 times in the second phase. For all algorithms, the computation overhead of the first phase is relatively small compared to that of the second phase. This indicates that, when $r = 1000$, the major overhead of network motif discovery arises from the second phase.

To more clearly illustrate the superiority of our GPU-based solution, we compute the *improvement ratio* of *NemoGPU* over *DistributedNM*, defined as the running time of the latter divided by that of the former. Figure 5a shows the improvement ratio of *NemoGPU* when $r = 1000$ and $k = 6$, on all datasets except CE (as *DistributedNM* fails to terminate on CE). In addition, we take into account the price differences among the CPU and GPUs, and compute the *performance-price ratio* of *NemoGPU*, defined as

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

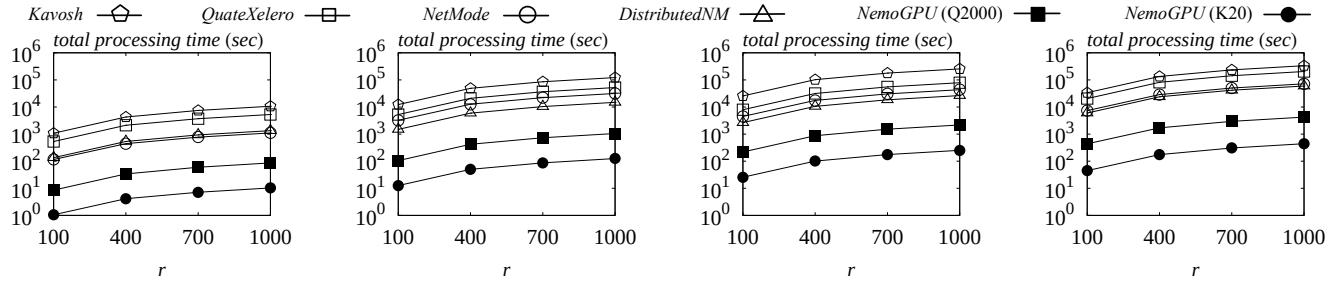
Figure 5b plots the performance-price ratio of *NemoGPU* with Q2000 and K20 when $r = 1000$. The ratio for K20 (resp. Q2000) is up to 29 (resp. 10), and is above 15 (resp. 7) in all cases. This indicates that both K20 and Q2000 yield much higher "performance per dollar" than the E5625 CPU does. Hence, 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 6 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 6, 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



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

Fig. 6. Computation time vs. k.



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

Fig. 7. Computation time vs. r.

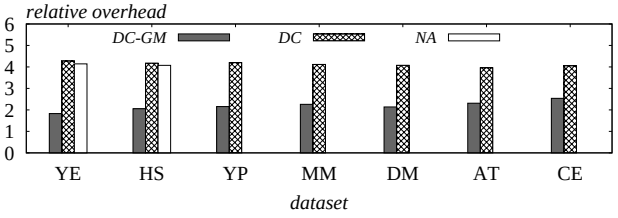
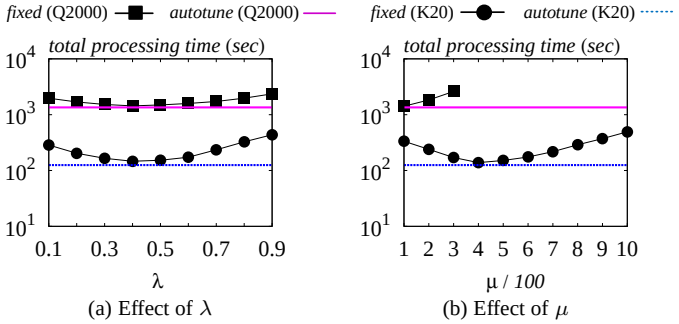


Fig. 8. Effects of optimization vs. dataset.



(a) Effect of λ (b) Effect of μ

Fig. 9. Total processing time on YP by varying λ and μ.

$k \geq 7$. This is because *NemoGPU* engages massive GPU threads to parallelize the intensive tasks, but *DistributedNM* has much smaller number of threads than *NemoGPU*.

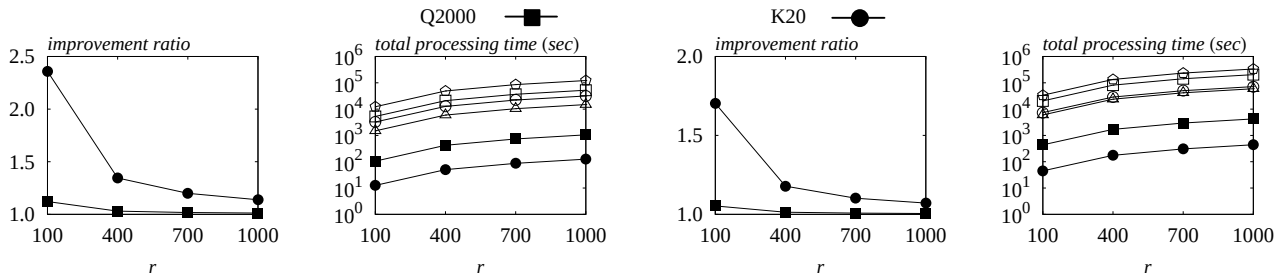
We have also measured the efficiency of all algorithms against r. Figure 7 shows the computation time of each algorithm on datasets YE, HS, YP, and MM, with r varying from 100 to 1000. Observe that our GPU-based methods have much smaller running time than all CPU-based methods do, regardless of the value of r. Specifically, *NemoGPU* consistently outperforms *DistributedNM* by two (resp. one) orders of magnitude when running with K20 (resp. Q2000), which again demonstrates the high capability of GPU for processing massive tasks.

7.3 Effects of Optimizations

In the next section of experiments, we evaluate the effects of the three optimization techniques proposed in Section 5: 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 8 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 5 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.

Recall that (i) *DC* has a parameter $\lambda \in (0, 1)$ that decides the amount of GPU memory used for enumerating size-i ($i < k$) subgraphs, (ii) *GM* has a parameter $\mu \in [1, r]$ that decides the number of random graphs to be processed in one batch, and (iii) we set λ and μ automatically based on our analysis in Section 5. To evaluate our choice of λ and μ, we measure the performance of *NemoGPU* on the dataset YP when λ and μ vary, and we plot the results in Figure 9. Observe that the optimal value for λ is around 0.4 (on both K20 and Q2000), and our automatic choice of



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

Fig. 10. *NemoGPU* (with Q2000 and K20) vs. the hybrid method.

λ results in a performance identical to the optimum. In contrast, $\lambda \leq 0.3$ (resp. $\lambda \geq 0.5$) leads to sub-optimal efficiency, as it makes *NemoGPU* allocate an excessively small amount of GPU memory for the enumeration of small (resp. large) subgraphs. (See the analysis in Section 5.1.) Meanwhile, the optimal value for μ on K20 is around 400, and our automatic choice of μ also yields the optimal result. In contrast, $\mu \geq 500$ overloads the GPU with an excessive number of random graphs to process in a batch, whereas $\mu \leq 300$ under-utilizes the GPU. On Q2000, however, the best value for μ is only 100, since $Q2000$ has a much smaller GPU memory than K20. (We omit the results for $\mu \geq 300$ on Q2000, as Q2000's GPU memory cannot accommodate more than 300 random graphs.)

Finally, to illustrate the importance of optimizing the first phase using GPUs when r is small, we compare *NemoGPU* with a *hybrid* method that runs the first phase by *DistributedNM* and the second phase by *NemoGPU*. We compute the improvement ratio of *NemoGPU* over the hybrid method, defined as the running time of the latter divided by that of the former. Figure 10 shows the improvement ratio of *NemoGPU* as a function of r on YE, HS, YP, and MM. The ratio for K20 (resp. Q2000) increases when r decreases, which indicates that when r is small, the computation cost of the first phase becomes relatively more significant with respect to the second phase. *NemoGPU* on K20 is noticeably more efficient than *Hybrid* in general, and is around two times faster than the latter when $r = 100$. Meanwhile, *NemoGPU* on Q2000 achieves relatively small improvements over the hybrid method. The reason is that, even when $r = 100$, the major overhead of *NemoGPU* on Q2000 is still incurred by the second phase; therefore, even though *NemoGPU* on Q2000 is one order of magnitude faster than the hybrid method in the first phase (see Figure 4), the advantage of the former diminishes when we take both phases into account. That said, we note that *NemoGPU* on Q2000 is never slower than the hybrid method, i.e., it dominates the hybrid method regardless of r.

8 Related Work

A plethora of CPU-based techniques [5], [6], [7], [8], [9], [10], [11], [12], [24] have been proposed for network motif discovery. As discussed in Section 2.3, 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 7, 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 [30], [31], [32], [33] 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 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.

In addition, there are several recent studies [34], [35] 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 [34], [35] 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 compared with those in G. As a consequence, the techniques in [34], [35] are not suitable for network motif discovery.

Furthermore, there exist numerous techniques for *frequent subgraph mining* (see [29] for a survey) and *significant subgraph mining* [36], [37], [38], but those two problems are considerably different from network motif discovery. In particular, in frequent subgraph mining, we are given a set of graphs $\mathcal{G}$, 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.

A preliminary version of this paper appears in [39]. Compared with [39], the current paper features two new contributions. First, we propose a novel GPU-based method for the subgraph enumeration phase of network motif discovery (see Section 6), and it is able to remove the bottleneck that accounts for 99% of the processing cost in subgraph enumeration. In contrast, the technique in [39] relies on an existing CPU-based algorithm [12] in the subgraph enumeration phase. Second, we conduct new experiments that (i) evaluate our GPU-based subgraph enumeration algorithm against CPU-based methods, and (ii) empirically evaluate the internal parameters λ and μ in our technique (see Section 7).

9 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 grant ARC19/14 from MOE, Singapore and a gift from Microsoft Research Asia.

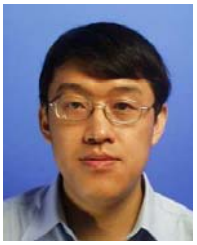
REFERENCES

[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://www.boost.org/doc/libs/1_60_0/libs/graph/doc/compressed_sparse_row.html.
[23] W. Lin, X. Xiao, J. Cheng, and S. S. Bhowmick, "Efficient algorithms for generalized subgraph query processing," in *CIKM*, 2012, pp. 325–334.
[24] 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.
[25] http://thrust.github.com.
[26] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in mapreduce," in *ICDE*, 2014, pp. 844–855.
[27] http://lbb.ut.ac.ir/Download/LBBsoft/QuateXelero/networks/.
[28] http://genemania.org.
[29] 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.
[30] 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.
[31] I. Bordino, D. Donato, A. Gionis, and S. Leonardi, "Mining large networks with subgraph counting," in *ICDM*, 2008, pp. 737–742.
[32] 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.
[33] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe, "Subgraph enumeration in large social contact networks using parallel color coding and streaming," in *ICPP*, 2010, pp. 594–603.
[34] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *ICDE*, 2013, pp. 62–73.
[35] 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.
[36] S. Ranu and A. K. Singh, "Graphsig: A scalable approach to mining significant subgraphs in large graph databases," in *ICDE*, 2009, pp. 844–855.
[37] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *SIGMOD Conference*, 2008, pp. 433–444.
[38] H. He and A. K. Singh, "Graphrank: Statistical modeling and mining of significant subgraphs in the feature space," in *ICDM*, 2006, pp. 885–890.
[39] W. Lin, X. Xiao, X. Xie, and X. Li, "Network motif discovery: A GPU approach," in *ICDE*, 2015, pp. 831–842.

Wenqing Lin received the PhD degree in computer science from the Nanyang Technological University (NTU) in 2015. He is currently a research scientist in the Data Analytics Department at the Institute for Infocomm Research (I^{2}R), Agency for Science, Technology and Research (A*STAR), Singapore. His research interests include graph databases, data mining, and parallel computing.

Xiaokui Xiao received the PhD degree in computer science from the Chinese University of Hong Kong in 2008. He is currently an Associate Professor at the Nanyang Technological University (NTU), Singapore. Before joining NTU in 2009, he was a postdoctoral associate at the Cornell University. His research interests include data privacy, spatial databases, graph databases, and parallel computing.

Xing Xie is currently a senior researcher in Microsoft Research Asia, and a guest PhD advisor with the University of Science and Technology of China. His research interest include spatial data mining, location-based services, social networks, and ubiquitous computing. He is a senior member of the ACM and IEEE.



Xiao-Li Li is currently a department head and a senior scientist in the Data Analytics Department at the Institute for Infocomm Research (I^{2}R), Agency for Science, Technology and Research (A*STAR), Singapore. He also holds adjunct associate professor positions at National University Singapore (NUS) and Nanyang Technological University (NTU). His research interests include data mining, machine learning, and bioinformatics.