SubGraph2Vec: Highly-Vectorized Tree-like Subgraph Counting

Anonymous Author(s)

ABSTRACT

Subgraph counting aims to count occurrences of a template T in a given network G(V,E). It is a powerful graph analysis tool and has found real-world applications in diverse domains. Scaling subgraph counting problems is known to be memory bounded and computationally challenging with an exponential complexity. Although scalable parallel algorithms are known for several graph problems such as Triangle Counting and PageRank, this is not common for counting complex subgraphs. Here we address this challenge and study connected acyclic graphs, or trees. We propose a novel vectorized subgraph counting algorithm, named SubGraph2Vec, as well as both shared memory and distributed implementations: 1) reducing algorithmic complexity by minimizing neighbor traversal; 2) achieving a highly-vectorized implementation upon linear algebra kernels to significantly improve performance and hardware utilization. SubGraph2Vec improves the overall performance over the state-of-the-art work by orders of magnitude and up to 660x on a single node. 4) SubGraph2Vec in distributed mode can scale up the template size to 20 and maintain a good strong scalability. 5) enabling portability to both CPU and GPU.

KEYWORDS

Subgraph Counting, Vectorization, Portability

ACM Reference Format:
Anonymous Author(s). 2019. SubGraph2Vec: Highly-Vectorized Tree-like Subgraph Counting. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC'19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Counting subgraphs from a large network is fundamental in graphs problems. It has been used in real world applications across a range of disciplines, such as in bioinformatics [5], social networks analysis, neuroscience [13] Online social network has billion- or trillion-sized network, where certain group of users may share specific interests[42]. Studying these groups improves the searching algorithm and [22] enables an estimate of graphlet (size up to 5) counts in social networks with 50 million of vertices and 200 million of edges. In bioinformatics, the frequency or distribution of the occurrence of each different testing templates may characterize a protein-protein interaction network [5][37], where repeated subgraphs are crucial in understanding cell physiology as well as developing new drugs.

SC'19, November 17-22, 2019, Denver, Colorado

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Counting the exact number of subgraphs of size k in a n-vertex network takes $O(k^2 n^k)$ time, which is computationally challenging even for moderate values of n and k. In fact, determining whether a graph G contains a subgraph to H is a related graph isomorphic problem that is NP-complete [23]. Arvind et al.[9] provides an approximate algorithm, called color coding to estimate the exact count with statistical guarantees bounded treewidth graphs. Although the color-coding algorithm in [5] has a time complexity linear in network size, it is exponential to subgraph size. Therefore, efficient parallel implementations are the only viable way to count subgraphs from large-scale networks. To the best of our knowledge, a multi-threaded implementation named Fascia [37] is considered to be the state-of-the-art work in this area. Still, it takes Fascia more than 4 days (105 hours) to count a 17-vertex subgraph from the RMAT-1M network (1M vertices, 200M edges) on a 48-core Intel (R) Skylake processor. While our proposed algorithm named **SubGraph2Vec** takes only 9.5 minutes to complete the same task on the same hardware.

The primary contributions of this paper are as follows:

- **Algorithmic Design.** We identify and reduce the computation complexity of the sequential color-coding algorithm, which also helps reduce communication overhead in distributed systems.
- **System design and optimization.** We design the data structure as well as the execution order to maximize the hardware efficiency in terms of vector register and memory bandwidth usage. The new design replaces the vertex-programming model by using linear algebra kernels.
- **Performance evaluation and comparison to prior work.** We characterize the performance compared to state-of-the-art work Fascia, and our solution attains the full hardware efficiency according to a roofline model analysis.

2 PRELIMINARIES

2.1 Motivation

Recent research has achieved substantial progress on counting the occurrences of subgraphs such as triangles or 4-cycle subgraphs: not only have very efficient algorithms been developed, but also a theoretical explanation of graph models and their applications in real-world use.

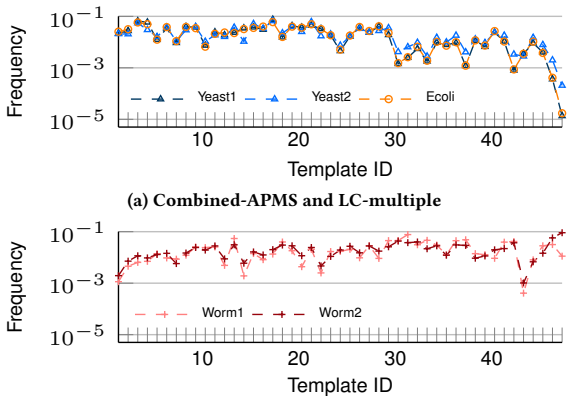
Comparing two graph networks via subgraph counting to measure topological features how similar any given pair of networks could be.

Note that an induced subgraph of a graph G is a subset of the vertices of the graph G as well as with any edges connecting pairs of vertices in that subset. If G is a fully connect graph of size n, then a circle goes through every vertex in G is not an induced subgraph of G; it is a non-induced subgraph of G since there are missing edges between vertex pairs. There are many more non-induced subgraphs isomorphic to a given topology and thus it is

more difficult to count non-induced subgraphs of a network. The motivation of considering non-induced subgraph is

Counting non-induced subgraphs from a large network is fundamental in numerous applications such as the protein-protein interaction network (PPIN), where repeated subgraphs (motifs) are are crucial in understanding cell physiology as well as developing new drugs. As PPINs usually include many false (positive and negative) and missing interactions[5], counting non-induced subgraph is more suitable to get reliable and robust results [5].

Alon et al. [5, 7] provide a practical algorithm, named color coding, to count trees and graphs of bounded treewidth (size less than 10) from PPINs of unicellular and 41 multicellular organisms by using the color-coding technique developed in [8]. Figure 1



(a) Combined-APMS and LC-multiple

(b) WI-2004 and WI-2007

Figure 1: A comparison of treelet distributions of five PPIN networks by SubGraph2Vec

illustrates such a real-world application of PPIN, where we count tree-like motifs with different sizes to estimate their frequencies and observe that the distribution for the unicellular Ecoli and yeasts are very close, while the more complex C. elegans (a kind of worm) is quite different.

2.2 Statement of Problem

2.2.1 Subgraph Counting. Subgraph finding and counting is a widely studied subject. A (non-induced) subgraph of a simple unweighted graph $G(V, E)$ is a graph $H(V_H, E_H)$ satisfying $V_H \subset V$ and $E_H \subset E$. H is an embedding of a template graph T if T is isomorphic to H. The subgraph counting problem is to count the number of all embeddings of a given template T in a network G. We use $emb(T, G)$ to denote the number of all embeddings of template T in network G.

2.2.2 Color coding method and its precision. Color-coding [8] is an algorithmic technique which is useful in the discovery of network motifs. The basic idea of color-coding is: given a k-node template T, we assign random colors between 0 and $k-1$ to the vertices of the network graph G, and count the number of occurrences of the template that are colorful, meaning the vertices matched to the template have distinct colors. Both theoretical proof [6, 8, 19] and experiments [5, 39] show that with proper normalization, the colorful count is an unbiased estimator of the actual count.

SubGraph2Vec is a vectorized implementation of the color coding technique. Alon et al. [8] proved that to guarantee a count bound of $(1 \pm \epsilon)emb(T, G)$ with probability $1 - 2\delta$, we would need to run at most N iterations, as defined in Algorithm 1.

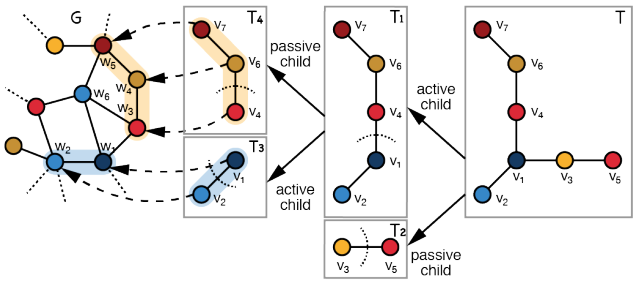


Figure 2: Illustration of the template partitioning within a colored input $G = (V, E)$

2.3 Sequential Algorithm

Color coding [5] provides a *fixed parameter tractable* algorithm to address the subgraph counting problem where T is a tree. It has a time complexity of $O(c^k \text{poly}(n))$, which is exponential to template size k but polynomial to vertex number n. Algorithm 1 describes the standard sequential algorithm with definition and notation shown in Table 1, which contains three important steps as follows.

Algorithm 1: Standard Sequential Algorithm

$N = O(\frac{e^k \log(1/\delta)}{\epsilon^2})$ `// required iterations to converge`
Partition T into sub-templates T_s
for $j = 1$ *to* N **do**
  color all $V_i \in G(V, E)$ randomly
  counting T_s in a dynamic programming procedure
  $P \leftarrow$ probability that the template is colorful
  $\alpha \leftarrow$ number of automorphisms of T_0
  $finalCount[j] \leftarrow \frac{1}{P\alpha} \sum_i \sum_C \mathbf{M}_0(i, I_C)$
Output the average of all $finalCount$.

2.3.1 Template Partitioning. For tree-like templates, we can recursively partition T into a chain of sub-templates T_s until the sub-template containing only one vertex (see Figure 2). When partitioning a template T, a single vertex ρ is selected as the root while $T_s(\rho)$ refers to the s-th sub-template rooted at ρ. Secondly, one of the edges (ρ, τ) adjacent to root ρ is cut, creating two child sub-templates. The child holding ρ as its root is named *active child* and denoted as $T_{s,a}$. The child rooted at τ of the cutting edge is named *passive child* and denoted as $T_{s,p}$.

2.3.2 Random Coloring. Each vertex $v \in G(V, E)$ is given an integer value of color randomly selected between 0 and $k-1$, where $k \geq |V_T|$ (we consider $k = |V_T|$ for simplicity). $G(V, E)$ is therefore converted to a labeled graph. We consider an embedding H as "colorful" if each of its vertices has a distinct color value. In [8],

Alon proves that the probability of H being colorful is $\frac{k!}{k^k}$, and color coding approximates the exact number of H by using the count of colorful H.

Algorithm 2: Dynamic Programming Procedure

input : $G(V, E), T$
output : $\mathbf{M}_s$
forall *sub-templates* T_s *in reverse order of their partitioning* **do**
  if T_s *consists of a single vertex* **then**
    forall $V_i \in V$ **do**
      $\mathbf{M}_s(i, \text{color of } V_i) \leftarrow 1$
  else
    `// $T_s$ has an active child $T_{s,a}$ and a passive child $T_{s,p}$`
    forall *vertices* $V_i \in V$ **do**
      forall *color sets* C_s *satisfying* $|C| = |T_s|$ **do**
        forall *color sets* $C_{s,a}$ *and* $C_{s,p}$ *created by splitting* C_s *satisfying* $|C_{s,a}| = |T_{s,a}|$ *and* $|C_{s,p}| = |T_{s,p}|$ **do**
          $\mathbf{M}_s(i, I_s) \leftarrow \sum_{V_j \in N(V_i)} \mathbf{M}_{s,a}(i, I_{s,a}) \mathbf{M}_{s,p}(j, I_{s,p})$

2.3.3 Counting by Dynamic Programming. Algorithm 2 describes the dynamic programming procedure to count partitioned template T from the randomly colored $G(V, E)$. For bottom sub-template $|T_s| = 1$, $M_s(i, I_s)$ is 1 only if C_s equals the color randomly assigned to V_i, and otherwise it is 0. For non-bottom cases where $|T_s| > 1$, we obtain $M_s(i, I_s)$ by multiplying the count values from its two children, which have been calculated in previous steps of dynamic programming.

Table 1: Definitions and Notations

Notation	Definition
$G(V, E)$ or G	The input network
$\mathbf{A}_G$	$\lvert V\rvert \times \lvert V\rvert$ sparse adjacency matrix of $G(V, E)$
$N(V_i)$ or $N(i)$	Neighbors of vertex V_i
T, T_s	The input template and the s-th sub-template
$\lvert T_s\rvert$	Number of vertices in T_s
$T_{s,a}$, $T_{s,p}$	Active and passive child of T_s
n	$n = \lvert V\rvert$ is the number of vertices in G
k	$k = \lvert V_T\rvert$ is the number of vertices in T
C_s	Color set for T_s
$\mathbf{M}_s$	$\lvert V\rvert \times \binom{k}{\lvert T_s\rvert}$ dense matrix to store counts for T_s
$\mathbf{M}_{s,a}$, $\mathbf{M}_{s,p}$	Dense matrix to store counts for $T_{s,a}$, $T_{s,p}$
$\mathbf{B}$	$\mathbf{B} = \mathbf{A}_G \mathbf{M}_{s,p}$, the sum of the counts of all neighbors.
I_{C_s} or, I_s	Column index of color set C_s calculated by Equation 1

The index I_s requires a bijection between the color sets and integers. The combinatorial number system [36] is such a correspondence between natural numbers and k-combinations. For a sub-template T_s of size h with k possible colors, the color set C_s would be $C_s = \{c_1, c_2, \ldots, c_h\} \subset \{0, 1, \ldots, k-1\}$, where $c_1 \leq c_2 \leq \cdots \leq c_h$. The corresponding index of C_s is

$$I_{C_s} = \binom{c_1}{1} + \binom{c_2}{2} + \ldots \binom{c_h}{h}. \quad (1)$$

3 RELATED WORK

A tree subgraph enumeration algorithm by combining color coding with a stream-based cover decomposition was developed in [48]. To process massive networks, [49] developed a distributed version of color-coding based tree counting solution upon MapReduce framework in Hadoop, [38] implemented a MPI-based solution, and [47] [21] pushed the limit of subgraph counting to process billion-edged networks and trees up to 15 vertices.

Beyond counting trees, a sampling and random-walk based technique has been applied to count graphlets, a small induced graph with size up to 4 or 5, which include the work of [4] and [22]. Later, [19] extends color coding to count any graph with a treewidth of 2 in a distributed system.

Color coding is a very general method and [8] showed that color coding applies to any subgraph of tree, circle and bounded tree width. [19] is a color coding implementation that can be applied to all templates with a tree width of no more than 2.

Also, [34, 35] provides a pruning method on labeled networks and graphlets to reduce the vertex number by orders of magnitude prior to the actual counting.

Other subgraph topics include: 1) *subgraph finding*. As in [24], paths and trees with size up to 18 could be detected by using multi-linear detection; 2) *Graphlet Frequency Distribution* estimates relative frequency among all subgraphs with the same size [17] [31]; 3) *clustering* networks by using the relative frequency of their subgraphs [33]. *Subgraph Matching* finds and enumerates all isomorphic subgraphs to a given template from input network. [40] contributes an online algorithm to query subgraph templates from billion-node network by using intelligent graph exploration to replace expensive join operations. [28] compares and summarizes subgraph isomorphism algorithms in graph databases. Later on [15] improves the performance of subgraph matching up to three orders of magnitude by postponing the Cartesian products based on the structure of a query to minimize the redundant Cartesian products.

Subgraph counting can also be used to define the similarity between graphs. The graphlet frequency distance (GFD) was proposed by Przulj et. al[32] as a global comparative measure based on the local structural characteristics of different networks. Bordino et al. [16] demonstrates that one can use the relative frequency of subgraphs within networks to distinguish and cluster different networks. Using the relative frequencies of undirected subgraphs up to four vertices and other topological properties such as in-degree, out-degree, and PageRank as representative features for a network, they show up to 75% clustering accuracy for networks chosen from seven distinct categories. Using directed edges and 284 features in total, they achieved just over 90% clustering accuracy.

The GraphBLAS project was inspired by the Basic Linear Algebra Subprograms (BLAS) with the goal of building graph algorithms upon a small set of kernels. GrpahBLAS libraries includes [18]

[41][26] [45][43].GraphBLAS operations have been successfully employed to implement a suite of traditional graph algorithms including Breadth-first traversal (BFS) [41], Single-source shortest path (SSSP) [41], Triangle Counting [10], and so forth. More complex algorithms have also been developed with GraphBLAS primitives. For example, the high-performance Markov clustering algorithm (HipMCL) [11] that is used to cluster large-scale protein-similarity networks is centered around a distributed-memory SpGEMM algorithm.

4 ALGORITHMIC DESIGN OF SUBGRAPH2VEC

The dynamic programming in Algorithm 2 requires $\binom{|T|}{|T_s|}\binom{|T_s|}{|T_{s,a}|}$ times of vertex neighbor traversal for each V_i from line 7 to 9. However, we find the redundancy in this traversal, which is shown in Figure 3, where it counts a two-vertex sub-template with a total

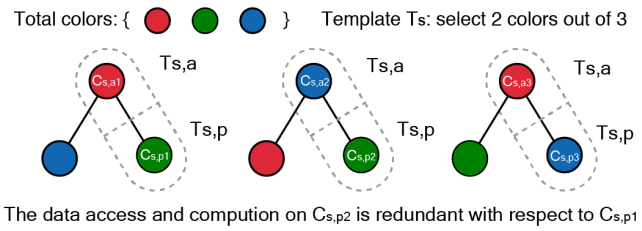


Figure 3: Identify the redundancy of standard color coding in a two-vertex sub-template T_s, which is further split into an active child and a passive child.

of three colors. The left case and the middle case have the same color set (the same $I_{s,p}$) assigned to their passive child $T_{s,p}$, which causes redundant access to $\mathbf{M}_{s,p}(j, I_{s,p})$ when traversing neighbor vertices.

Algorithm 3: Dynamic Programming in SubGraph2Vec

input : $G(V,E), \mathbf{M}_{s,p}, T_s$
output: $\mathbf{M}_s$: matrix storing traversal results

1 **for** $V_i \in G(V,E)$ **do**
2   **for** *color sets* $C_{s,p}$ *satisfying* $|C_{s,p}| = |T_{s,p}|$ **do**
3   **forall** $V_j \in N(V_i)$ **do**
4       $\mathbf{B}(i, I_{s,p}) \leftarrow \mathbf{B}(i, I_{s,p}) + \mathbf{M}_{s,p}(j, I_{s,p})$
5 **for** $V_i \in G(V,E)$ **do**
6   **for** *color set* C_s *satisfying* $|C_s| = |T_s|$ **do**
7   **for** *color sets* $C_{s,a}, C_{s,p}$ *split from* C_s **do**
8       $\mathbf{M}_s(i, I_s) \leftarrow \mathbf{M}_s(i, I_s) + \mathbf{M}_{s,a}(i, I_{s,a})\mathbf{B}(i, I_{s,p})$

On the contrary, SubGraph2Vec proposes a new way to accomplish the vertex neighbor traversal described from 1 to 4 of Algorithm 3:

(1) The vertex neighbor traversal is decoupled from line 9 of Algorithm 2.

(2) Only $\binom{|T|}{|T_{s,p}|}$ times of traversal is applied on each vertex, which is proven to be the minimum amount.

According to distributive property of addition and multiplication, line 9 of Algorithm 2 can be re-written as

$$\sum_{V_j \in N(i)} \mathbf{M}_{s,a}(i, I_{s,a})\mathbf{M}_{s,p}(j, I_{s,p}) = \mathbf{M}_{s,a}(i, I_{s,a}) \sum_{V_j \in N(i)} \mathbf{M}_{s,p}(j, I_{s,p}) \tag{2}$$

, where the first item $\mathbf{M}_{s,a}(i, I_{s,a})$ at right-hand only contains count values of V_i while the second item $\sum_{V_j \in N(i)} \mathbf{M}_{s,p}(j, I_{s,p})$ only involves traversing neighbors of V_i. This decoupled design enables a caching and re-using of the traversal results (the summation of $\mathbf{M}_{s,p}(j, I_{s,p})$ shown in Figure 4), which allows us to reduce the traversal times.

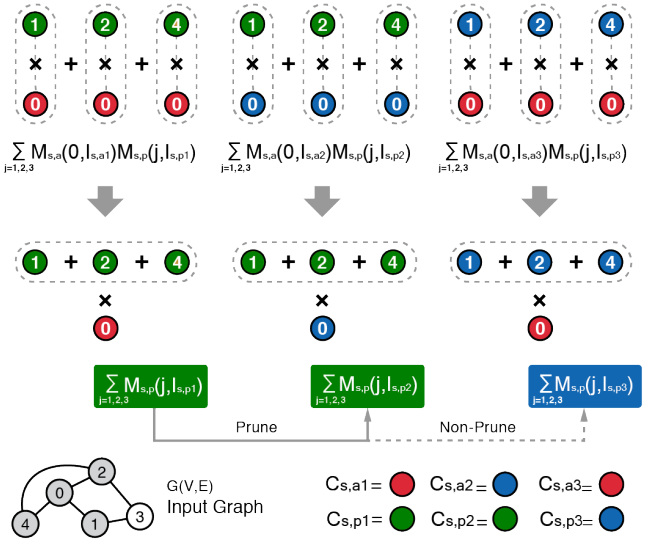


Figure 4: Decouple the vertex neighbor traversal from updating of the count value according to distributive property of addition and multiplication in Equation 2.

Secondly, we prove that $\binom{|T|}{|T_{s,p}|}$ is the minimal required amount of vertex neighbor traversal. Supposing that $|T_{s,p}| \leq |T_s| \leq |T|$, we have $\binom{|T|}{|T_s|}\binom{|T_s|}{|T_{s,p}|}/\binom{|T|}{|T_{s,p}|} = \binom{|T|-|T_{s,p}|}{|T|-|T_s|} \geq 1$ Hence, we have $\binom{|T|}{|T_{s,p}|} \leq \binom{|T|}{|T_s|}\binom{|T_s|}{|T_{s,p}|}$. On the other hand, all color set combinations of C_s are required at line 7 of Algorithm 2, meaning that all color set combinations of $C_{s,p}$ will be covered because $C_s = C_{s,a} \cup C_{s,p}$. Therefore, $\binom{|T|}{|T_{s,p}|}$ is the minimal times of required vertex neighbor traversal. In addition, the operation of updating M_s at line 9 of Algorithm 2 can be re-written in Algorithm 3 from line 5 to 8.

The minimization of vertex neighbor traversal also reduces the complexity of the standard sequential algorithm of color coding. A single time of vertex neighbor traversal by all $V_i \in G(V,E)$ has a complexity of $O(|E|)$, and a single time of counting template at line 4 of Algorithm 3 is $O(|V|)$. We assume that the sizes of sub-templates are uniformly distributed between 1 and n, and we have the following lemmas.

Lemma 4.1. *For a template or a sub-template T_s, if the counting of the sub-templates of T_s has been completed, then the time complexity of counting T_s is:*

$$O(|E|\binom{|T|}{|T_{s,p}|} + |V|\binom{|T|}{|T_s|}\binom{|T_s|}{|T_{s,p}|}). \tag{3}$$

Proof. First, $T_{s,p}$ has $\binom{|T|}{|T_{s,p}|}$ color combinations, which requires $\binom{|T|}{|T_{s,p}|}$ times of vertex neighbor traversal having a time complexity of $O(|E|\binom{|T|}{|T_{s,p}|})$. Secondly, T_s has a total of $\binom{|T|}{|T_s|}$ different color combinations, and each sub-template has a total of $\binom{|T_s|}{|T_{s,p}|}$ splits, thus we have a complexity of $O(|V|\binom{|T|}{|T_s|}\binom{|T_s|}{|T_{s,p}|})$ for line 8 of Algorithm 3. □

Lemma 4.2. *For integers $l \le m \le n$, and $l_0 \le 0, l_1 \le 1, \ldots, l_n \le n$, the following equations hold:*

(1) $\max_m \binom{n}{m} = O(n^{-1/2}2^n)$
(2) $\max_{\{m,l\}} \binom{n}{m}\binom{m}{l} = O(n^{-1}3^n)$
(3) $\max_{\{l_0,l_1,\ldots,l_n\}} \sum_{m=0}^{n} \binom{n}{m}\binom{m}{l_m} = O(n^{-1/2}3^n)$

Lemma 4.3. *In the worst case, the total time complexity of counting a k-node template using SubGraph2Vec is in Equation 4.*

$$O((e^k \log(\frac{1}{\delta})\frac{1}{\epsilon^2})(|E|2^k + |V|k^{-1/2}3^k)). \tag{4}$$

Proof. A k-node template generates up to $O(k)$ sub-templates. And $O(e^k \log(\frac{1}{\delta})\frac{1}{\epsilon^2})$ iterations are performed in order to get the (ϵ, δ)-approximation. □

Whereas, the total time complexity of the standard sequential algorithm is shown in Equation 5.

$$O((e^k \log(\frac{1}{\delta})\frac{1}{\epsilon^2})(|E|k^{-1/2}3^k)). \tag{5}$$

By comparing the time complexities of 4 and 5, we observe an algorithmic improvement of SubGraph2Vec to the standard sequential algorithm in terms of the template size $k = |T|$ and the average degree $|E|/|V|$ of $G(V, E)$. With $|E|/|V| \gg 1$, we have an improvement proportional to $(3/2)^k k^{-1/2}$. For large templates of $k \gg 1$, we have an improvement proportional to $|E|/|V|$. In general, SubGraph2Vec has an algorithmic improvement over the standard sequential algorithm with large templates and dense input networks, and we have experiments results in Section 7 to support this implication. This complexity reduction is also crucial to ensure that the algorithm is scalable in distributed system, where the traversal of vertex neighbour usually curses expensive inter-machine communication overhead.

5 COALESCED MEMORY ACCESS AND VECTORIZATION

To parallelize the dynamic programming procedure in Algorithm 2, a naive implementation is to parallelize the *for* loop over $V_i \in G(V, E)$ by using threads (the graph traversal model). However, modern high-end processors are normally equipped with vector register units supporting single instruction multiple data (SIMD) programming paradigm. Modern compilers could automatically vectorize the codes within a *for* loop if the loop has consecutive

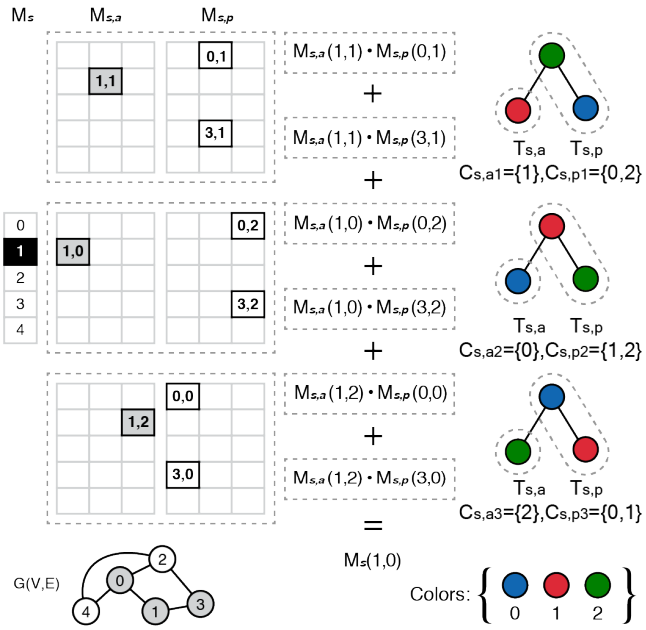


Figure 5: Illustrate Line 6 to 9 of Algorithm 2) by a five-vertex $G = (V, E)$ and a three-vertex template T. The column indices in $\mathbf{M}_{s,a}$ and $\mathbf{M}_{s,p}$ are calculated from their color combinations by Equation 1.

indices and regular memory access pattern (e.g., access memory by stride 1). Unfortunately, we find that the graph traversal model of Algorithm 2 does not meet the SIMD paradigm. In Figure 5, we examine a simple case where a thread is traversing its two neighbors and updating count values in columns of $\mathbf{M}_s$. There are two barriers:

(1) The count values from neighbors are not adjacent in the memory layout. See $\mathbf{M}_{s,p}(0, 1)$ and $\mathbf{M}_{s,p}(3, 1)$ in Figure 5.
(2) The indices like I_s are not consecutive because of using the combinatorial number system in Equation 1. For instance, I_s equals 1, 0, 2 in up, middle, and bottom parts of Figure 5.

These two barriers still exist even in our decoupled procedures shown in Algorithm 3. By carefully selecting and modifying data structure and thread workflow, we propose a new scheme in SubGraph2Vec to achieve highly-vectorized codes and coalesced memory access.

5.1 Vectorization in Vertex Neighbour Traversal

SubGraph2Vec uses an adjacency matrix, notated as $\mathbf{A}_G$ for the input network $G(V, E)$. $\mathbf{A}_G$ is a sparse 0-1 matrix satisfying $\mathbf{A}_G(i, j) = 1$ if and only if $V_j \in N(V_i)$. There are various data formats to represent a sparse matrix, for instance, the Sparse Row Compressed (CSR) format utilizes three dense arrays:

(1) *val* stores the values of nonzero entries
(2) *colIdx* stores the column indices of each nonzero entry.
(3) *rowIdx* stores the offset of the first nonzero entry in *values* for each row.

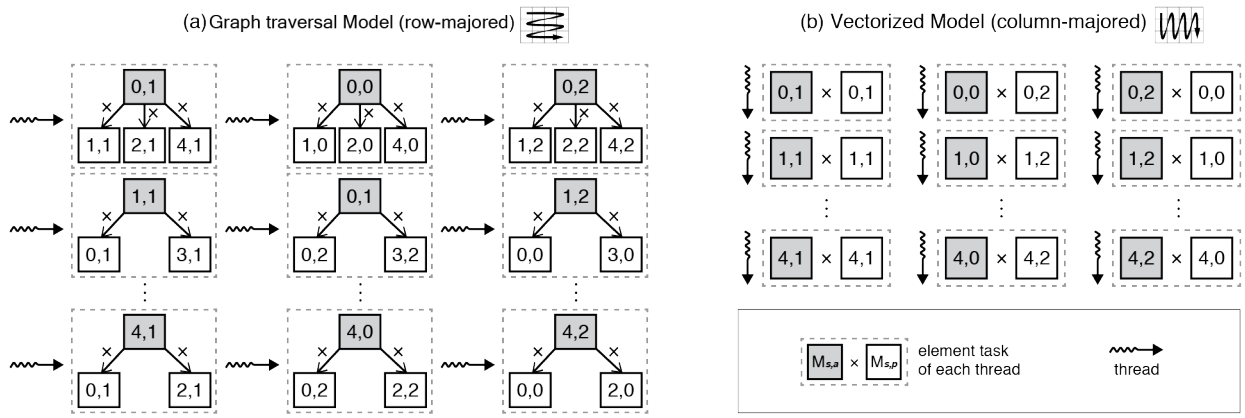


Figure 6: Comparing the thread execution order, where (a) Graph traversal Model has count data stored in memory with a row-majored layout, and (b) SubGraph2Vec (Vectorized Model) has count data stored in memory with a column-majored layout.

Therefore, we re-write line 1 to 4 of Algorithm 3 by Algorithm 4, where for each $I_{s,p}$, we schedule loops of V_i to threads while each thread is vectorizing its own work. We observe that j has successive values from $A_G.rowIdx[i]$ to $\mathbf{A}_G.rowIdx[i+1]$ resulting in coalesced data access to three dense arrays of A_G. Unfortunately, $\mathbf{A}_G.colIdx[j]$ does not guarantee successive values due to the sparsity of $\mathbf{A}_G$. However, advanced compilers still provide partial vectorization support to this indexed access pattern. We will introduce our customization in addressing this partial vectorization issue in Section 6.

Algorithm 4: Vectorized Neighbor Traversal in SubGraph2Vec

input : $\mathbf{A}_G, T_s, \mathbf{M}_{s,p}$
output : $\mathbf{B}$

1 **forall** *color sets* $C_{s,p}$ *satisfying* $|C_{s,p}| = |T_{s,p}|$ **do**
2 **forall** $V_i \in \mathbf{A}_G$ **do** `// loop is scheduled to threads`
3 **forall** $j = \mathbf{A}_G.rowIdx[i]$ *to* $\mathbf{A}_G.rowIndex[i+1]$ **do**
 `// thread workload is vectorized`
4 $\mathbf{B}(i, I_{s,p}) \leftarrow$
 $\mathbf{B}(i, I_{s,p}) + \mathbf{A}_G.val[j]\mathbf{M}_{s,p}(\mathbf{A}_G.colIdx[j], I_{s,p})$

5.2 Thread Workflow for Vectorization

In Algorithm 3, counting templates at line 8 cannot be vectorized because the indices are not successive. One solution is to find another index system to provide successive index values. Nevertheless, there would be varying column numbers in $\mathbf{M}_s$ for different sub-template T_s. The same variation occurs at $\mathbf{M}_{s,a}$ and $\mathbf{M}_{s,p}$. For small T_s, the length of vectorization could be less than 10, which causes an under utilization of hardware resource.

To address this issue, we propose a new scheme illustrated in Figure 6.

(1) Change memory layout from row-majored order to column-majored order.
(2) All threads are working on same columns of $\mathbf{M}_s, \mathbf{M}_{s,a}, \mathbf{M}_{s,p}$ concurrently and processing the matrices column by column.
(3) All rows of a column are evenly distributed to threads.
(4) Each thread vectorizes the work on its own portion of rows.

Compared to the graph traversal model, the length of vectorization is converted from $\binom{|T|}{|T_s|}$ to the million-level number of vertices in $G(V, E)$, which is sufficient to fully utilize the hardware and invariant to different sub-templates. Furthermore, the stride one regular memory access is efficient in prefetching data from memory to cache lines.

6 INVOCATION OF LINEAR ALGEBRA KERNELS

Compared to the graph traversal model, SubGraph2Vec is designed to be portable among hardware platforms while keeping high performance. The vectorized vertex neighbor traversal module in Algorithm 4 mathematically equals to an operation of sparse matrix dense vector multiplication (SpMV), which is an essential sparse linear solver on different hardware platforms. Correspondingly, line 8 of Algorithm 3 equals an element-wised multiplication and addition of dense vectors (eMA). A complete SubGraph2Vec made of SpMV and eMA kernels is described in Algorithm 5, which also applies an in-place storage of the SpMV results from the column vector buffer $\mathbf{B}$ back to $\mathbf{M}_{s,p}$ to reduce the memory footprint.

To achieve better kernel performance than by using public libraries, we customize both of SpMV and eMA kernels. For SpMV, we combine a bundle of SpMV operations in Algorithm 5 into a Sparse matrix dense matrix (SpMM) operation shown in Algorithm 6, where the right-hand dense matrix is $\mathbf{M}_{s,p}$. To save peak memory utilization, we also split columns of $\mathbf{M}_{s,p}$ into batches with a pre-selected batch size. To improve the load balancing and data locality, we utilize a split compressed sparse column (CSC-Split) format instead of the default compressed sparse row (CSR) format that is widely used by public libraries. CSC-Split format converts the standard CSC format into a fixed number of partitions. Entries of CSC matrix are distributed to a partition when their row

Algorithm 5: SubGraph2Vec with Linear Algebra Kernels

input : $\mathbf{A}_G, T, \epsilon, \delta$
output: A (ϵ, δ)-approximation to $emb(T, G)$

1 $N = O(\frac{e^k \log(1/\delta)}{\epsilon^2})$ `// required iterations to converge`
2 Partition T into sub-templates T_s
3 **for** $j = 1$ *to* N **do**
4 **forall** $V_i \in G(V, E)$ **do**
5 Color V_i by a value randomly drawn from 1 to $k = |T|$
6 **for** $s = 0, 1, \ldots, S-1$ **do**
7 **forall** *color sets* $C_{s,p}$ *satisfying* $|C_{s,p}| = |T_{s,p}|$ **do**
8 $\mathbf{B} \leftarrow \mathbf{A}_G \mathbf{M}_{s,p}(:, I_{s,p})$ `// SpMV kernel`
9 $\mathbf{M}_{s,p}(:, I_{s,p}) \leftarrow \mathbf{B}$ `// Sum of neighbor counts`
10 **forall** *color sets* C_s *satisfying* $|C_s| = |T_s|$ **do**
11 $\mathbf{M}_s(:, I_s) \leftarrow 0$
12 **forall** *color sets* $C_{s,a}$ *and* $C_{s,p}$, *created by splitting* C_s *satisfying* $|C_{s,a}| = |T_{s,a}|$ *and* $|C_{s,p}| = |T_{s,p}|$ **do**
13 $\mathbf{M}_s(:, I_s) \leftarrow \mathbf{M}_s(:, I_s) + \mathbf{M}_{s,a}(:, I_{s,a}) \odot \mathbf{M}_{s,p}(:, I_{s,p})$ `// eMA kernel`
14 $P \leftarrow$ probability that the template is colorful
15 $\alpha \leftarrow$ number of automorphisms of T_0
16 $finalCount[j] \leftarrow \frac{1}{P\alpha} \sum_i \sum_C \mathbf{M}_0(i, I_C)$
17 Output the average of all $finalCount$.

IDs fit into a pre-defined range of that partition. Inside a partition, the entries are ordered by their column IDs of CSC format, and therefore entries sharing the same column ID and adjacent row IDs are bundled together to improve the data locality and cache usage. Meanwhile, we store batches of right-hand vectors from $\mathbf{M}_{s,p}$ in a row-majored memory layout, and we set up the batch size to the maximal concurrent element number of the hardware SIMD unit. Finally, when a partition is assigned to a thread, the thread processes its entries one by one while vectorizing the computation work on a batch of row entries from $\mathbf{M}_{s,p}$.

To customize the eMA kernel, we utilize Intel (R) AVX intrinsics, where multiplication and addition are implemented by using the fused multiply-add (FMA) instruction, which cuts the computation instructions by half.

In addition, there are already substantial research work in developing high performance linear algebra kernels. The invocation of linear algebra kernels in SubGraph2Vec benefits from: 1) using formats and kernel implementations tailored for different input datasets; 2) increasingly improved kernel performance on various hardware platforms.

7 EXPERIMENTS AND RESULTS

7.1 Datasets and Templates

The datasets in our experiments are listed in Table 2, where *Graph500 Scale=20, 21, 22* are collected from [25]; *Miami*, *Orkut*, and *NYC* are from [12] [29] [44]; RMAT are widely used synthetic datasets generated by the RMAT model [20], where we increase parameter K to generate datasets with increasingly skewed degree distribution. The templates in Figure 7 are from the tests in [37] or created by us. The template size increases from 10 to 17 while some templates have two different shapes.

Algorithm 6: CSCSplit SpMM for sub-template T_s in SubGraph2Vec

input : $A_G, \mathbf{M}_{s,p}, sIdx, BSize, SplitPars$
output: Out

1 **forall** $Par \in SplitPars$ **do** `// partition per thread`
2 **forall** $e \in Par$ **do** `// workload per thread`
3 **for** $j = sIdx, \ldots, sIdx + BSize$ **do**
4 $Out(e.rowId, j) \leftarrow Out(e.rowId, j) + \mathbf{A}_G(e.rowId, e.colId)\mathbf{M}_{s,p}(e.colId, j)$ `// rowId, colId are row and column indices in CSC-Split compressed sparse format`

7.2 Hardware and Software

In the experiments, we use: 1) a single node of a dual-socket Intel(R) Xeon(R) CPU E5-2670 v3 (architecture Haswell), 2) a single node of a dual-socket Intel(R) Xeon(R) Platinum 8160 CPU (architecture Skylake-SP) processors, and 3) a single node of Tesla V100 SXM2 paired with an Intel(R) Xeon(R) CPU E5-2630 v4. More details of the testbed hardware as well as the computation environment are released in the Artifact Description file.

We use the following implementations.

- **Fascia** implements the graph traversal model of color-coding algorithm with multi-threading on a single CPU [37], which serves as a performance baseline.
- **SubGraph2Vec** implements SubGraph2Vec on CPU by using our in-house CSC-Split format with a SpMM kernel and eMA kernel (threaded by OpenMP). It is the default implementation of SubGraph2Vec and supports distributed systems.
- **SubGraph2Vec-MKL** implements SubGraph2Vec on a single CPU by using CSR based SpMV kernel from Intel MKL and eMA kernel (threaded by OpenMP). It also supports distributed systems.
- **SubGraph2Vec-cuSPARSE** implements SubGraph2Vec on GPU by using CSR based SpMV kernel from NVIDIA cuSPARSE and eMA kernel (threaded by CUDA). It can support distributed systems by using the CSR format API from distributed mode of SubGraph2Vec-MKL.

Binaries on CPU are compiled by the Intel(R) C++ compiler for Intel(R) 64 target platform from Intel(R) Parallel Studio XE 2019, with compilation flags of "-O3", "-xCore-AVX2", "-xCore-AVX512", and the Intel(R) OpenMP. Binaries on GPU are compiled by CUDA release 9.1 (V9.1.85). The distributed binaries are compiled by Intel MPI 2019. We use, by default, a thread number equal to the physical core number of CPU, i.e., 48 threads on a Skylake node and 24 threads on a Haswell node. The threads are bind to cores with a spread affinity. For GPU, we use a thread block with a size of 1024 for the eMA kernel. For kernel invoked by Intel MKL and NVIDIA cuSPARSE, we use the default setup. We mainly use the

Table 2: Datasets used in the experiments (K=10^3, M=10^6)

Data	Vertices	Edges	Avg Deg	Max Deg	Abbreviation	Source
EcoliGO-BP	1,474	6,896	9.36	72	Ecoli	Biology [1]
WI-2004	1,239	1,736	2.8	74	Worm1	Biology [2]
WI-2007	1,498	1,817	2.43	86	Worm2	Biology [2]
Combined-APMS	1,622	9,070	11.18	127	Yeast1	Biology [3]
LC-multiple	1,536	2,925	3.81	40	Yeast2	Biology [3]
Graph500 Scale=20	600K	31M	48	67K	GS20	Graph500 [25]
Graph500 Scale=21	1M	63M	51	107K	GS21	Graph500 [25]
Graph500 Scale=22	2M	128M	53	170K	GS22	Graph500 [25]
Miami	2.1M	200M	49	10K	MI	Social network [12]
Orkut	3M	230M	76	33K	OR	Social network [29]
NYC	18M	960M	54	429	NY	Social network [44]
RMAT-1M	1M	200M	201	47K	RT1M	Synthetic data [20]
RMAT(K=3)	4M	200M	52	26K	RTK3	Synthetic data [20]
RMAT(K=5)	4M	200M	73	144K	RTK5	Synthetic data [20]
RMAT(K=8)	4M	200M	127	252K	RTK8	Synthetic data [20]

Table 3: Execution time (s) of SubGraph2Vec versus Fascia with increasing template sizes from U12 to U17.

Dataset	Algorithm	u12	u13	u14	u15-1	u15-2	u16	u17
Miami	F	163	400	944	2663	2435		
Miami	S	18	38	55	160	150		
Orkut	F	642	2006	4347	1.5e4	1.2e4		
Orkut	S	30	67	80	238	230		
RMAT 1M	F	1535	5378	1.2e4	3.4e4	3.2e4	1.1e5	3.8e5
RMAT 1M	S	16	32	34	97	97	224	573
Graph500 20	F	132	452	923	3379	2679		
Graph500 20	S	7	14	21	63	56		
Graph500 21	F	289	1044	2036	7535	5914		
Graph500 21	S	12	26	36	105	102		
Graph500 22	F	764	2814	5477	1.9e4	1.6e4		
Graph500 22	S	26	53	74	220	194		
RMAT K=3	F	1191	4890	9711	3.0e4	3.2e4		
RMAT K=3	S	39	110	170	377	262		
RMAT K=5	F	2860	9906	2.0e4	9.0e4	5.4e4		
RMAT K=5	S	29	60	82	233	240		
RMAT K=8	F	5620	2.0e4	3.3e4	9.4e4	8.5e4		
RMAT K=8	S	25	51	67	217	234		

Tests run on a Skylake node. F = Fascia and S = SubGraph2Vec.

SubGraph2Vec to evaluate our work except for Section 7.8, where SubGraph2Vec with public library kernels are evaluated against Fascia.

7.3 Performance Evaluation

We first examine the performance improvement of SubGraph2Vec over the state-of-the-art Fascia on a Skylake node. The best performance we can obtain is by using customized matrix format and SpMM kernel. Note that we scale the template size up to the memory limitation on our Skylake testbed for each dataset in Table 3. The reduction of execution time is significant particularly for template sizes larger than 14. For instance, Fascia spends four days to process a million-vertex dataset RMAT-1M with template u17 while SubGraph2Vec only spends 9.5 minutes. For relatively smaller templates such as u12, SubGraph2Vec still achieves 10x to 100x of improvement on datasets Miami, Orkut, and RMAT-1M.

In Table 3, we observe that the improvement is approximately proportional to the average degree of datasets. For instance, SubGraph2Vec achieves 10x and 20x improvements on datasets Miami (average degree of 49) and Orkut (average degree of 76), respectively. It implies that our optimization works better on dense graph network when compared to Fascia.

The three Graph500 datasets in Table 2 have comparable average degrees but growing vertex number and edge number. For the same template, SubGraph2Vec obtains similar improvements over Fascia across the three datasets implying that SubGraph2Vec has a scalable performance improvement with respect to the dataset size.

Finally, we compare RMAT datasets with increasingly skewed degree distribution, which causes a thread-level workload imbalance. The results show that SubGraph2Vec has comparable execution time regardless of the degree distribution. On the contrary, Fascia spends significantly (2x to 3x) more time on datasets with skewed degree distribution.

7.4 Non-Vectorized versus Vectorized

To test the performance gains of vectorization, we implemented a non-vectorized SubGraph2Vec, keeping the same computation complexity as SubGraph2Vecand uses multi-threading while not applying the vectorization techniques introduced in Section 56. We tested the performance improvements shown in Figure 9 on Miami dataset.

We see that vectorization brings a performance boost of at least 8 times. When With increasing thread number, the performance improvement will decrease, which means SubGraph2Vectakes full advantage of the VPU, so the gain from the increase in the number of threads is relatively less. Despite this, the performance of fully vectorized SubGraph2Vecis still far beyond the non-vectorized version.

7.5 Hardware Utilization

In addition to the improvement in execution time, SubGraph2Vec also enjoys a better hardware utilization over Fascia, which is expected because of our co-design approach that explores the hardware horsepower from multiple aspects. It is worth noting that we only evaluate the codes doing the counting workload and exclude other code sections such as data loading. The evaluated codes of Fascia is a *for* loop parallelized by OpenMP threads while the evaluated codes of SubGraph2Vec are the SpMM and eMA kernels.

7.5.1 ***CPU and VPU Utilization.*** Figure 8(a) first compares the CPU utilization defined as the average number of concurrently running physical cores. For Miami, Fascia achieves 60% of CPU utilization. However, the CPU utilization drops down below 50% on Orkut and NYC. Conversely, SpMM kernel keeps a high CPU utilization from 65% to 78% for all datasets. The eMA kernel has a growing CPU utilization from Miami (46%) to NYC (88%). We have two explanations: 1) the SpMM kernel splits and regroups the nonzero entries by their row IDs, which mitigates the imbalance of nonzero entries among rows; 2) the eMA kernel has its computation workload for each column of $\mathbf{M}_{s,a}, \mathbf{M}_{s,p}$ evenly dispatched among threads.

Secondly, we examine the code vectorization in Figure 8. VPU in a Skylake node is a group of 512-bit registers. The scalar instruction also utilizes the VPU but it cannot fully exploit its 512-bit length. Figure 8 refers to the portion of instructions vectorized with a full vector capacity. For all of the three datasets, Fascia only has 6.7% to 12.5% VPU utilization implying that the codes are not vectorized. While for SpMM and eMA kernels of SubGraph2Vec, the VPU utilization is 100%. A further metric of packed float point instruction ratio (Packed FP) justifies the implication that Fascia has zero vectorized instructions but SubGraph2Vec has all of its float point operations vectorized.

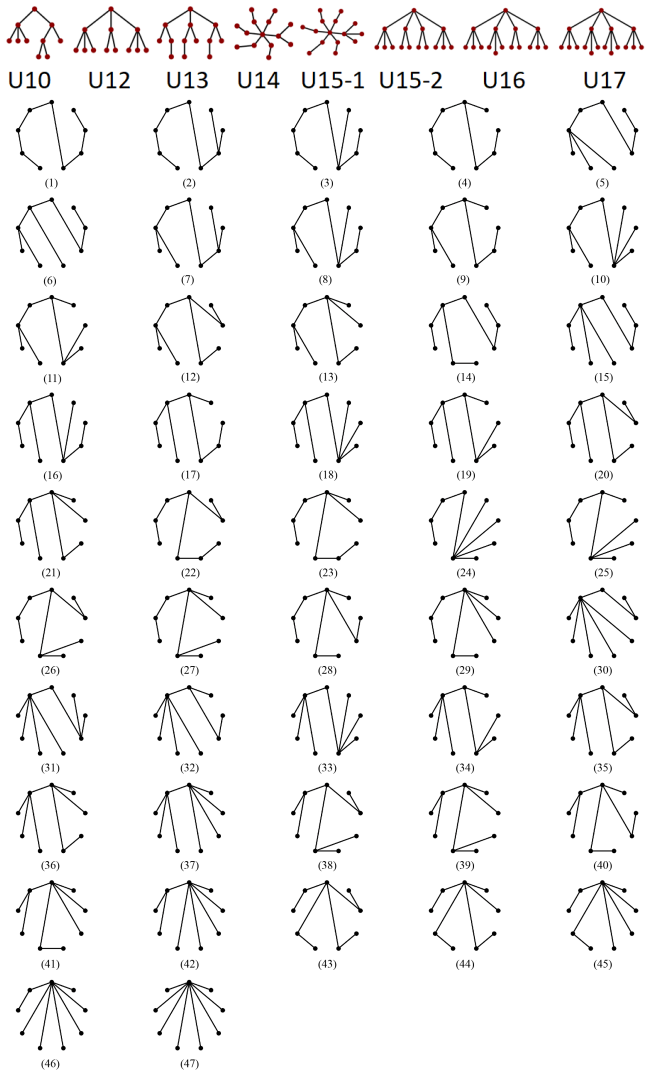


Figure 7: Templates used in experiments

Table 4: Memory and Cache Usage of Fascia, SpMM, and eMA of SubGraph2Vec on a Skylake Node

Miami	Bandwidth	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate
Fascia	**6 GB/s**	4.1%	1.8%	**85%**
SpMM	**86.95 GB/s**	8.3%	51.2%	**36.8%**
eMA	**106 GB/s**	0.3%	20.6%	**9.9%**
Orkut	**Bandwidth**	**L1 Miss Rate**	**L2 Miss Rate**	**L3 Miss Rate**
Fascia	**8 GB/s**	9.6%	5.3%	**46%**
SpMM	**59.5 GB/s**	6.7%	42.8%	**45%**
eMA	**116 GB/s**	0.32%	22.2%	**9.0%**
NYC	**Bandwidth**	**L1 Miss Rate**	**L2 Miss Rate**	**L3 Miss Rate**
Fascia	**7 GB/s**	2.4%	8.1%	**87%**
SpMM	**96 GB/s**	7.7%	76%	**74%**
eMA	**122 GB/s**	0.1%	99%	**14.8%**

7.5.2 ***Memory Bandwidth and Cache Usage.*** Because of the sparsity, subgraph counting is memory bounded within a shared memory system. Therefore, the utilization of memory and cache are critical to the overall performance. In Table 4, we compare SpMM and eMA of SubGraph2Vec to Fascia. It shows that the eMA kernel has the highest bandwidth value around 110 GB/s for the three datasets, which is due to the highly vectorized codes and regular memory access pattern. The data is prefetched into cache lines, which mitigates the cache miss rate as low as 0.1%.

The SpMM kernel also enjoys a decent bandwidth usage around 70 to 80 GB/s by average when compared to Fascia. Although SpMM has an L3 miss rate as high as 74% in dataset NYC because the memory footprint is larger than the L3 cache capacity. The optimized thread level and instruction level vectorization ensures a concurrent data loading from memory, which leverages the high memory bandwidth. Fascia has the lowest memory bandwidth usage because of the thread imbalance and the irregular memory access.

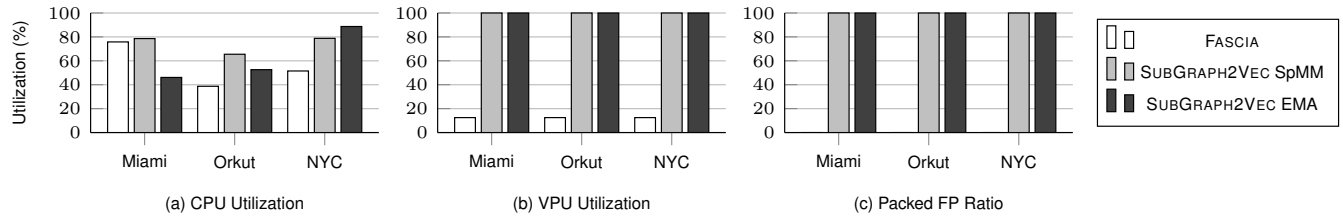


Figure 8: The hardware utilization on one Skylake node for template u12.

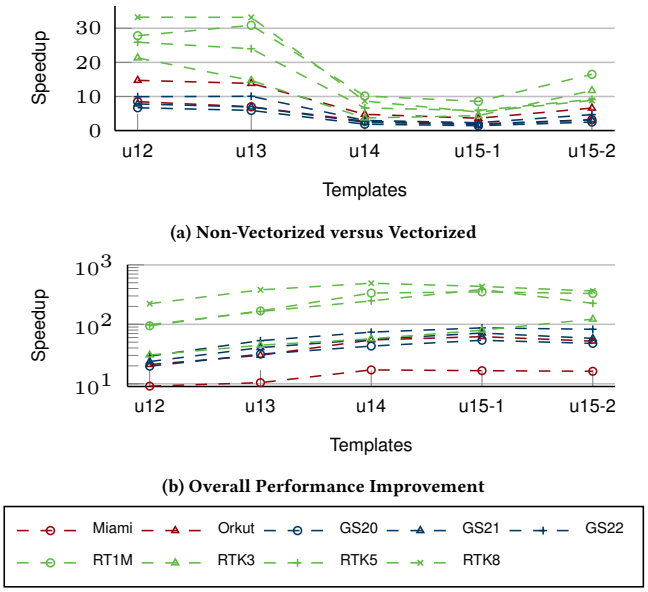


Figure 9: The performance improvement brought by vectorization and the overall performance improvement. The Scaling tests run on a Skylake node.

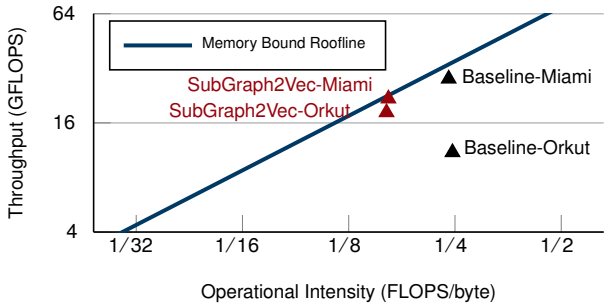


Figure 10: Apply roofline model to FASCIA and SUBGRAPH2VEC. Dataset Miami, Orkut for template u15-1. Tests are done on a Skylake node.

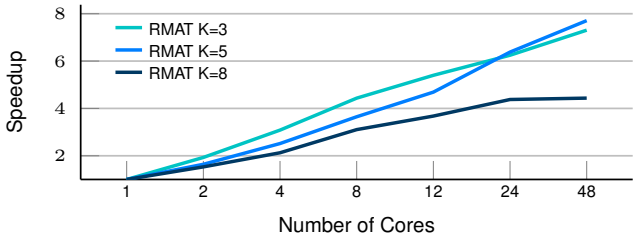


Figure 11: Thread scaling for RMAT datasets with increasing skewness on a Skylake node.

7.5.3 ***Roofline Model.*** The roofline model in Figure 10 reflects the hardware efficiency. The horizontal axis is the operational intensity (FLOP/byte) and the vertical axis refers to the measured throughput performance (FLOP/second). The solid roofline is the maximal performance the hardware can deliver under a certain operational intensity. Because of the low operational intensity, the performance of FASCIA and SUBGRAPH2VEC are bounded by the memory bandwidth and we consider it as a memory-bound roofline. For a relatively small dataset like Miami, both of FASCIA and SUBGRAPH2VEC are close to the memory-bound roofline because the data can be fit into the 33 MB L3 cache. For dataset Orkut, whose data size is beyond the capacity of L3 cache, SUBGRAPH2VEC is much closer to the memory-bound roofline than that of FASCIA because of its regular and vectorized memory access pattern.

7.6 Parallelization of Single Node

We perform a strong scaling test using up to 48 threads on Skylake node in Figure 11. We choose RMAT generated datasets with increasing skewness parameters of $K = 3, 5, 8$. When $K = 3$.

As the performance is bounded by memory, which has 6 memory channels per socket, we have a total of 12 memory channels on a Skylake node that bounds the thread scaling. Eventually, SUBGRAPH2VEC obtain a 7.5x speedup at 48 threads. When increasing the skewness of datasets to $K = 5, 8$, the thread scalability of SUBGRAPH2VEC drop down because the skewed data distribution brings workload imbalance when looping vertex neighbors.

7.7 Parallelization of Distributed Nodes

A distributed SUBGRAPH2VEC extends the memory capacity of a single node to run large templates. In Table 3, dataset RT1M can only run template with size up to u17 on a single node, however, we can scale up the template size to u20 by using 16 nodes shown in Figure 12.

In addition, our distributed SUBGRAPH2VEC has a good strong scalability, which even achieves superlinear speedup in Figure 13 from 1 node to 8 nodes. According to the analysis made in Section 7.5.3, SUBGRAPH2VEC is memory bounded, and increasing the

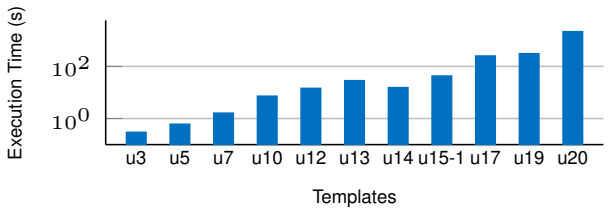


Figure 12: Scaling up the templates up to u20 by distributed SubGraph2Vec running on 16 Haswell nodes

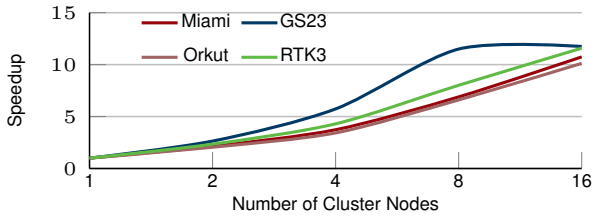


Figure 13: Strong scaling test on distributed SubGraph2Vec for four datasets and template u14.

number of nodes scales out not only computation resources but also memory bandwidth and cache resource. Having less data on each node can increase the percentage of data held by last level of CPU cache.

7.8 Portability to Other Platforms

Hardware platforms such as NVIDIA GPU requires higher programming skills to exploit their performance. However, there are already highly-optimized public libraries of linear algebra kernels. For SpMV operation, we have the *mkl_sparse_s_mv* kernel from Intel MKL library on CPU and the *cusparseScsrmv* kernel from NVIDIA cuSPARSE library on GPU. For eMA kernel, we can use a combination of *vsMul* and *vsAdd* kernels from Intel MKL or hand-implement such kernels whenever the kernel is absent because of its simplicity. Hence, we have ported SubGraph2Vec to GPU by keeping the CPU codes other than SpMV and eMA on the host side while invoking cuSPARSE and CUDA kernels for the two linear algebra operations.

In Figure 14, we port the performance of SubGraph2Vec to three platforms by using CSR-SpMV libraries kernels. When the template size is small, SubGraph2Vec-cuSPARSE has comparable or even better performance than SubGraph2Vec-MKL. However, the performance of SubGraph2Vec-cuSPARSE drops down when the template size grows up. As NVIDIA-V100 only has 16GB of device memory, it is probable that the large memory footprint of M_s brought by large template size cannot fit into the device memory, and the bi-directional data transfer between the host and device memory compromises the performance of SubGraph2Vec-cuSPARSE. Nevertheless, both of Intel MKL and NVIDIA cuSPARSE are not open sourced, and we cannot conclude on their performance gap. Also, the three hardware platforms have different theoretical peak performances and memory bandwidths, this result is only meant to demonstrate the portability of our SubGraph2Vec across hardware platforms.

7.9 Error Discussion

We implement the standard color coding algorithm that Alon et al. [5] prove to run at most N iterations to control approximation quality as in Algorithm 1. In practice, the subgraph counting with color coding requires only 100 iterations for a 7 node template on H.pylori with an error of less than 1% in FASCIA [39]. SubGraph2Vec with its pruning and vectorization optimization only differs from the Fascia due to the restructuring of the computation from Algorithm 2 to Algorithm 3. It should give identical results with exact arithmetic in Equation 2.

However, when dealing with large graphs, the counted value will exceed the range of integer variables. As a consequence, both Fascia and our SubGraph2Vec use 32-bit floating point numbers to avoid overflow. Hence, slightly different results are observed between Fascia and SubGraph2Vec due to the rounding error consequent from floating point arithmetic operations. Figure 15 reports such relative errors between SubGraph2Vec and Fascia in the range of 10^{-6} across all the tests on a Graph500 GS20 dataset with increasing template sizes, which is negligible.

8 CONCLUSION

Although a single machine with big shared memory and many cores is becoming an attractive solution to graph analysis problems [30], the irregularity of memory access remains a roadblock to improve the hardware utilization. For fundamental algorithms, such as PageRank, the fixed data structure and predictable execution order are explored to improve data locality either in graph traversal approach [27][46] or in linear algebra approach [14]. Subgraph counting, with random access to the vast memory region and dynamic programming workflow, requires much more effort to exploit the cache efficiency and hardware vectorization. In this paper, we fully vectorize a sophisticated algorithm of subgraph analysis, and the novelty is a co-design approach with pattern identification of linear algebra kernels that leverage hardware vectorization of Intel CPU and NVIDIA GPU architectures.

The overall performance achieves a promising improvement over the state-of-the-art work by orders of magnitude by average and up to 660x (RMAT1M with u17) within a shared-memory multi-threaded system. We will explore in our future work: 1) enabling counting of tree subgraph with size larger than 30 and subgraphs beyond tree; 2) extending the shared-memory implementation to a distributed system; 3) exploring other graph and machine learning problems by this co-design approach; 4) adding support to more emerging hardware architectures.

REFERENCES

[1] [n. d.]. EColiNet. https://www.inetbio.org/ecolinet/downloadnetwork.php.
[2] [n. d.]. Worm Interactome database. http://interactome.dfci.harvard.edu/C_elegans/index.php.
[3] [n. d.]. Yeast Interactome database. http://interactome.dfci.harvard.edu/S_cerevisiae/index.php.
[4] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. 2015. Efficient graphlet counting for large networks. In *Data Mining (ICDM), 2015 IEEE International Conference on*. IEEE, 1–10.

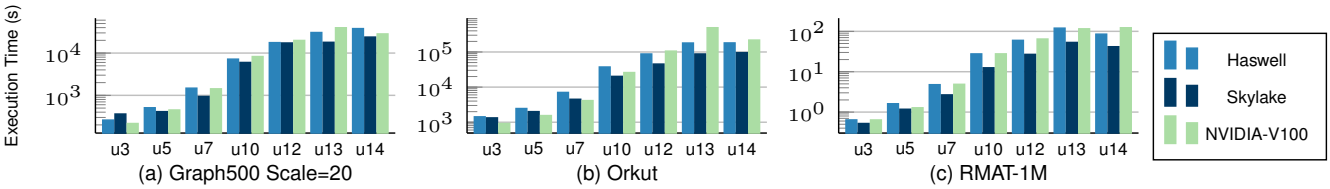


Figure 14: Execution Time of SubGraph2Vec on three platforms. On Haswell and Skylake nodes, we use CSR based SpMV kernel from Intel MKL; On Volta GPU V100, we use CSR based SpMV kernel from NVIDIA cuSPARSE

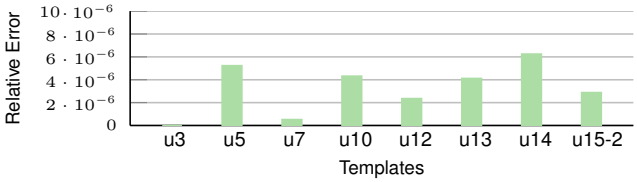


Figure 15: Relative error on dataset Graph500 Scale=20. Tests are done on a Skylake node.

[5] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[6] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[7] Noga Alon and Shai Gutner. 2007. Balanced families of perfect hash functions and their applications. In *International Colloquium on Automata, Languages, and Programming*. Springer, 435–446.

[8] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *Journal of the ACM (JACM)* 42, 4 (1995), 844–856.

[9] Vikraman Arvind and Venkatesh Raman. 2002. Approximation algorithms for some parameterized counting problems. In *International Symposium on Algorithms and Computation*. Springer, 453–464.

[10] A. Azad, A. BuluÃğ, and J. Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop* (2015-05). 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[11] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.

[12] Christopher L Barrett, Richard J Beckman, Maleq Khan, VS Anil Kumar, Madhav V Marathe, Paula E Stretz, Tridib Dutta, and Bryan Lewis. 2009. Generation and analysis of large synthetic social contact networks. In *Winter Simulation Conference*. Winter Simulation Conference, 1003–1014.

[13] Federico Battiston, Vincenzo Nicosia, Mario Chavez, and Vito Latora. 2017. Multilayer motif analysis of brain networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 27, 4 (2017), 047404.

[14] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing PageRank communication via propagation blocking. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 820–831.

[15] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1199–1214.

[16] Ilaria Bordino, Debora Donato, Aristides Gionis, and Stefano Leonardi. 2008. Mining large networks with subgraph counting. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 737–742.

[17] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting graphlets: Space vs time. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 557–566.

[18] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[19] Venkatesan T Chakaravarthy, Michael Kapralov, Prakash Murali, Fabrizio Petrini, Xinyu Que, Yogish Sabharwal, and Baruch Schieber. 2016. Subgraph counting: Color coding beyond trees. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. Ieee, 2–11.

[20] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[21] Langshi Chen, Bo Peng, Sabra Ossen, Anil Vullikanti, Madhav Marathe, Lei Jiang, and Judy Qiu. 2018. High-Performance Massive Subgraph Counting Using Pipelined Adaptive-Group Communication. *Big Data and HPC: Ecosystem and Convergence* 33 (2018), 173.

[22] Xiaowei Chen and John Lui. 2018. Mining graphlet counts in online social networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 4 (2018), 41.

[23] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 151–158.

[24] Saliya Ekanayake, Jose Cadena, Udayanga Wickramasinghe, and Anil Vullikanti. 2018. MIDAS: Multilinear Detection at Scale. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2–11.

[25] GraphChallenge. 2019. Graph Challenge MIT. https://graphchallenge.mit.edu/data-sets

[26] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Bill Howe. 2016. From NoSQL Accumulo to NewSQL Graphulo: Design and utility of graph algorithms inside a BigTable database. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 1–9.

[27] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2017. Accelerating PageRank using Partition-Centric Processing. (2017).

[28] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 133–144.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[30] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. 2015. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1105–1110.

[31] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.

[32] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[33] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. 2014. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2466–2478.

[34] Tahsin Reza, Christine Klymko, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2017. Towards practical and robust labeled pattern matching in trillion-edge graphs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

[35] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. 2018. PruneJuice: pruning trillion-edge graphs to a precise pattern-matching solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 21.

[36] Abu Bakar Siddique, Saadia Farid, and Muhammad Tahir. 2016. Proof of bijection for combinatorial number system. *arXiv preprint arXiv:1601.05794* (2016).

[37] George M Slota and Kamesh Madduri. 2013. Fast approximate subgraph counting and enumeration. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 210–219.

[38] George M Slota and Kamesh Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 405–414.

[39] George M. Slota and Kamesh Madduri. 2015. Parallel color-coding. *Parallel Comput.* 47 (2015), 51–69. https://doi.org/10.1016/j.parco.2015.02.004 bibtex: slota_parallel_2015.

[40] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.

[41] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[42] Johan Ugander, Lars Backstrom, and Jon Kleinberg. 2013. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 1307–1318.

[43] Carl Yang, Aydin Buluc, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. (2018).

[44] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. *Knowledge & Information Systems* 42, 1 (2012), 181–213.

[45] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan. 2016. GBTL-CUDA: Graph Algorithms and Primitives for GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016-05). 912–920. https://doi.org/10.1109/IPDPSW.2016.185

[46] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 293–302.

[47] Zhao Zhao, Langshi Chen, Mihai Avram, Meng Li, Guanying Wang, Ali Butt, Maleq Khan, Madhav Marathe, Judy Qiu, and Anil Vullikanti. 2018. Finding and counting tree-like subgraphs using MapReduce. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018), 217–230.

[48] Zhao Zhao, Maleq Khan, VS Anil Kumar, and Madhav V Marathe. 2010. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 594–603.

[49] Zhao Zhao, Guanying Wang, Ali R. Butt, Maleq Khan, V. S. Anil Kumar, and Madhav V. Marathe. 2012. SAHAD: Subgraph Analysis in Massive Networks Using Hadoop. In *IEEE International Parallel & Distributed Processing Symposium*.