Fringe-SGC: Counting Subgraphs with Fringe Vertices

Cameron Lloyd Bradley
clb420@txstate.edu
Texas State University
San Marcos, Texas, USA

Ghadeer Ahmed H. Alabandi
gaa54@txstate.edu
Texas State University
San Marcos, Texas, USA

Martin Burtscher
burtscher@txstate.edu
Texas State University
San Marcos, Texas, USA

Abstract

Subgraph Counting (SGC) is a fundamental component of many important applications, including cybersecurity, drug discovery, social network analysis, and natural language processing. However, current SGC approaches can only handle very small patterns (aka subgraphs) because the computational load increases exponentially with the size of the pattern. To overcome this limitation for certain patterns, we introduce a new technique and algorithm called Fringe-SGC for counting the exact number of times a subgraph occurs in a larger graph. Our approach conventionally searches only for the "core" of the subgraph and then uses set-based methods to compute the number of occurrences that the "fringes" add. Our evaluation shows that Fringe-SGC is able to count the instances of many subgraphs that are too large for state-of-the-art SGC frameworks. Furthermore, Fringe-SGC running on a GPU outperforms the state-of-the-art GPU-based SGC frameworks by up to 20× on average, especially on patterns with many fringes.

CCS Concepts

• Mathematics of computing → Graph enumeration; **• Computing methodologies → Massively parallel algorithms**.

Keywords

Graph pattern mining, subgraph counting, fringe vertices

ACM Reference Format:
Cameron Lloyd Bradley, Ghadeer Ahmed H. Alabandi, and Martin Burtscher. 2025. Fringe-SGC: Counting Subgraphs with Fringe Vertices. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759839

1 Introduction

Subgraph counting (SGC) is the process of counting all occurrences, often called embeddings, of a connected subgraph in a larger graph. Such subgraphs, which are also referred to as patterns or motifs, typically only encompass a few vertices and edges. Fig. 1 shows some examples. Counting how often a pattern appears can help shed light on pattern importance, make predictions, and discover insights. For instance, SGC is used in biological network analysis to study protein-protein interaction networks [15], in disease protein prediction [14], and in chemoinformatics to classify chemical compounds [7].

Moreover, SGC algorithms have been used for mapping human brain networks [9], fraud detection [6], natural language processing [27], cybersecurity [23], social network analysis [8], recommender systems [1], and drug discovery [25]. SGC has gained significant attention in recent years, and many SGC frameworks have been proposed (e.g., [12, 26, 28]). However, these frameworks only support small subgraphs due to the exponentially increasing computational complexity. For example, extending a pattern by just one vertex and one edge generally greatly increases the running time to find and count the pattern. The exponential nature is particularly pronounced for patterns with low-degree "fringe" vertices such as the ones highlighted in orange in Fig. 1. A fringe vertex (or just "fringe" for short) is a vertex that is only connected to the core of the subgraph but not to any other fringe vertices. Adding just a few fringes to a pattern quickly makes counting the occurrences intractable for existing SGC approaches.

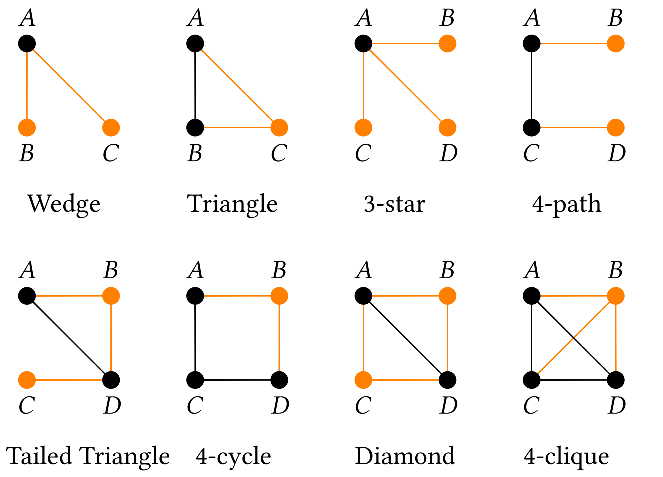


Figure 1: All possible connected 3- and 4-vertex patterns; the "core" is black and the "fringes" are orange

To see why, consider the graph in Fig. 2. There is only one triangle in this graph (formed by vertices 0, 1, and 2), but there are five unique tailed triangles. By unique we mean that none of the matched patterns encompass the same set of vertices. In a relatively small real-world Internet graph with 124,651 vertices and 193,620 edges, there are only 19,523 triangles but 880,555 tailed triangles and 21,095,445 2-tailed triangles (see Fig. 3).

These examples show that extending a pattern can drastically increase the number of occurrences and, concomitantly, the running time of SGC algorithms. Note that tails are the simplest type of fringes. The same trend applies to other types and is exacerbated when there are multiple fringes. Since *every possible pattern with two or more vertices contains at least one fringe*, it is important to develop techniques that can count such patterns efficiently.

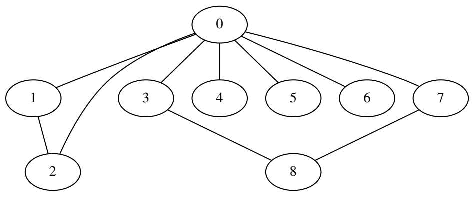


Figure 2: A small graph in which to count patterns

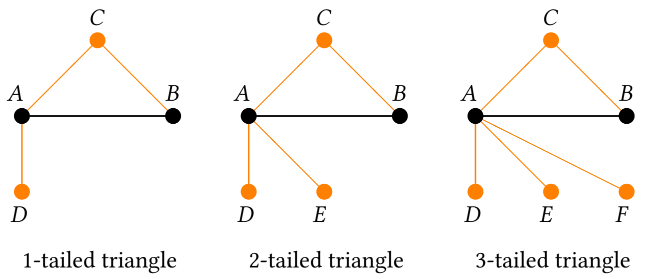


Figure 3: Three examples of k-tailed triangles

This paper presents a new method for counting patterns with fringes called Fringe-SGC. Our approach accomplishes this by conventionally searching only for the core of the pattern (i.e., the pattern without any fringes) and then mathematically accounting for the number of occurrences that the fringes add. Consequently, the performance advantage of Fringe-SGC tends to increase with increasing numbers of fringes for a fixed core. For example, the pattern shown in Fig. 4 with 16 vertices and 25 edges is far beyond the capabilities of other SGC frameworks. However, because the core is small (it only consists of three vertices), Fringe-SGC can count the number of occurrences of this pattern even in large graphs.

This paper makes the following main contributions.

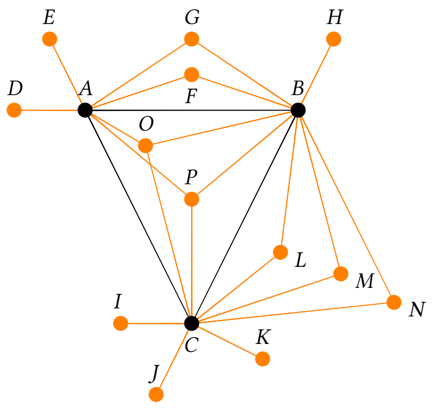


Figure 4: An example pattern with many fringes

- It introduces Fringe-SGC, a framework for counting the number of occurrences of user-defined patterns in graphs, which it decomposes into core and fringe vertices.
- It presents a general formula to compute the occurrences of arbitrarily many fringes and fringe types.
- It presents a GPU-friendly parallelization strategy for algorithms like Fringe-SGC to maintain parallelism in codes with deeply-nested conditional statements.
- It demonstrates that our parallel GPU implementation can be orders of magnitude faster than prior SGC frameworks and handle much larger patterns.

Our Fringe-SGC CUDA code is freely available in open source at https://github.com/burtscher/Fringe-SGC.git

The rest of this paper is organized as follows. Section 2 provides background information. Section 3 explains our approach in detail. Section 4 summarizes related work. Section 5 describes the evaluation methodology. Section 6 presents and discusses the results. Section 7 concludes the paper with a summary.

2 Background

The goal of Fringe-SGC is to count the number of embeddings of a user-provided pattern within a user-provided graph that are isomorphic to the pattern. Important note: by adding a simple print statement, we can change Fringe-SGC to not only count the pattern but also list all identified core locations and the number of patterns that surround each core. Doing so basically changes the code into a subgraph matching application. In this paper, we use Fringe-SGC and all evaluated third-party codes in counting mode only because outputting the matches can be slow and may require a lot of storage.

SGC includes a variety of algorithms such as Triangle Counting (TC) [24] and Clique Finding (CF) [11] and is related to Motif Counting (MC) [17] and Frequent Subgraph Mining (FSM) [10]. TC counts the number of triangles in the input. CF enumerates complete subgraphs (cliques). MC counts the frequency of each possible pattern up to a given size. FSM identifies all frequent patterns that occur at least a specified number of times, which is called "support".

SGC algorithms handle duplicate embeddings or automorphisms by selecting a canonical embedding and recording statistical data, such as the total number of these embeddings. However, determining the canonical embedding can be computationally demanding. Similarly, enumerating embeddings in a graph is an exponential process that can be memory-intensive and slow [4]. Additionally, each embedding must undergo a subgraph isomorphism test to confirm whether it is identical to the pattern, which is NP-complete. Consequently, some SGC algorithms rely on heuristics and approximations to efficiently enumerate embeddings [5]. In contrast, Fringe-SGC is an exact approach.

Most SGC frameworks are based on one of two types of embeddings: vertex induced or edge induced. Vertex-induced embeddings are generated by selecting a group of vertices and identifying the subgraph of interest that contains these vertices, along with the edges that link them in the input graph. In contrast, edge-induced embeddings are created by selecting a set of edges and including all endpoint vertices of those edges in the subgraph. The conventional search for either embedding type is done in a depth-first manner, starting with single-edge graphs and adding an extra edge in each step. This paper targets edge-induced embeddings.

3 Fringe-SGC Approach

Fringe-SGC is based on the observation that it is possible to compute the instances of a k-star pattern, rather than match each instance of the subgraph individually. A k-star is a pattern with a central vertex that is connected to k other vertices. For example, Fig. 1 shows both a 2-star (the wedge) and a 3-star. It is well-known that the number of k-stars in a graph can be determined without visiting the k "spoke" vertices [16, 21]. We only need to visit the central vertex and query its degree to compute the number of occurrences. For instance, to determine the number of 3-stars in the graph from Fig. 2, we visit each vertex and find that only Vertex 0 can be such a central vertex as all other vertices have a degree below 3. Moreover, we can *compute* that Vertex 0 is the central vertex of $\binom{7}{3} = 35$ 3-stars, where the 7 is the degree of Vertex 0 and the 3 is the number of spokes. More generally, for $k > 1$, every vertex v in any graph is the central vertex of exactly $\binom{d}{k}$ k-stars, where d is the degree of v.

Our Fringe-SGC approach generalizes this idea to larger patterns with more than one "central" vertex and more complex "spokes". In particular, the *core* takes the role of the central vertex and the *fringes* take the role of the spokes, meaning we only visit the core and compute how many occurrences the fringes add. The core and fringes of an arbitrary connected pattern are defined as follows.

Definition 3.1 (Core)**.** A core of a pattern is a minimal connected subset of the vertices such that all non-core vertices are only connected to core vertices.

Definition 3.2 (Fringe vertex)**.** A fringe vertex of a pattern is a vertex that is not part of the core.

Note that every pattern vertex either belongs to the core or is a fringe vertex. Moreover, the core is not unique. For example, in the triangle in Fig. 1, the core could just as well have been AC or BC.

To simplify the discussion, it is helpful to define the anchors of the fringes as follows.

Definition 3.3 (Anchors)**.** The anchors, anchor set, or anchor vertices of a fringe are the core vertices to which the fringe is connected.

Fringe-SGC starts with a depth-first search for the core of the pattern. Whenever an instance of the core has been found in the graph, it computes the number of distinct instances of the full pattern with fringes around the core instance. This number is a function of the neighbors of the matched core vertices. We first illustrate what this function looks like on the example of a core with 2 vertices to which we progressively add more fringes.

3.1 2-Vertex Core

The simplest pattern with a 2-vertex core (i.e., an "edge" core) is the triangle. It contains a new type of fringe that we call "wedge" fringe. Wedge-fringe vertices have degree 2 whereas tail-fringe vertices have degree 1. To compute how many wedge fringes a matched edge core has, we need to know how many neighbors the two core vertices have in common, which can be computed in linear time (see Sec. 3.5). With c common neighbors, there are $\binom{c}{1} = c$ possibilities for the wedge fringe. If we have two wedge fringes, that is, a diamond pattern (see Fig. 1), then there are $\binom{c}{2}$ possibilities. These formulas are well known [16, 21]. It is easy to see that, with m wedge fringes, there will be $\binom{c}{m}$ possibilities.

If we use this formula verbatim, we will overcount the number of patterns because of automorphisms, that is, because the pattern appears multiple times in itself. For example, the vertices of a triangle can be rotated into three different configurations (ABC, BCA, and CAB), and each configuration can be mirrored (ACB, CBA, and BAC). Hence, a triangle has six automorphisms. There are two ways to handle the overcounting: we can either divide the final count by the number of automorphisms, or we can introduce symmetry breaking [3], for instance by enforcing that the unique vertex ID of A must be smaller than B's, and B's must be smaller than C's. In the rest of this section, we assume that automorphisms are handled in one way or another.

The above formulas for k-stars, triangles, diamonds, etc. are already published in the literature. However, the following extensions and generalizations are new as far as we know.

Consider the tailed triangle, which has a 2-vertex core, a wedge fringe, and a tail fringe (see Fig. 1). The tail can be any non-core neighbor of the core vertex that has the tail, whereas the wedge fringe must be a common neighbor of both core vertices. The problem is that the two sets of neighbors overlap. If we use one of the common neighbors for the tail, we have one fewer option for the wedge fringe. This is the case because fringes are not allowed to share a vertex. If they did, we would detect a different pattern with fewer vertices. Hence, we must exclude all shared-vertex cases.

Assume we found a match for the 2-vertex core, that the two core vertices are u and v, and that the tail is attached to u. It is helpful to compute the sizes n_u and n_{uv} of the two disjoint vertex sets s_u and s_{uv}. This is done with the function $e(x)$, which returns the *external* neighbors of x, that is, the neighbors that are not vertices of the matched core:

$$s_u = e(u) \setminus e(v),\ n_u = |s_u|$$
$$s_{uv} = e(u) \cap e(v),\ n_{uv} = |s_{uv}|$$

Here, n_u is the number of non-core neighbors of u that are not neighbors of v, and n_{uv} is the number of non-core neighbors that are common to u and v. These two set sizes (note that we do not need the sets, only their sizes) are sufficient to compute the number of possibilities for the tailed triangle:

$$\binom{n_u}{1}\binom{n_{uv}}{1} + \binom{n_{uv}}{1}\binom{n_{uv}-1}{1}$$

The first term reflects the number of possibilities when the tail is chosen from s_u, that is, $\binom{n_u}{1}$ choices for the tail times $\binom{n_{uv}}{1}$ choices for the wedge fringe. The second term reflects the number of possibilities when the tail is chosen from s_{uv}, that is, $\binom{n_{uv}}{1}$ choices for the tail times $\binom{n_{uv}-1}{1}$ choices for the wedge fringe. Note that picking the tail from s_{uv} leaves one fewer choice for picking the wedge fringe, i.e., only $n_{uv} - 1$.

It is immaterial that we first picked the tail and then the wedge fringe because doing it the other way around yields the same result due to the following equality:

$$\binom{n}{a}\binom{n-a}{b} = \binom{n-b}{a}\binom{n}{b}$$

Adding more wedge fringes is easy as these fringes can only be selected from one set, namely s_{uv}. Thus, the total number of possibilities for an edge core with 1 tail and m wedge fringes is:

$$\binom{n_u}{1}\binom{n_{uv}}{m}+\binom{n_{uv}}{1}\binom{n_{uv}-1}{m}$$

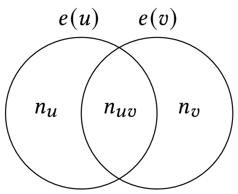
However, adding additional tails to the anchor of the first tail is more complicated because some of the tails may be selected from s_u and the rest from s_{uv}. With k tails attached to vertex u, we can select all k tails from s_u and none from s_{uv}, or we can select $k-1$ tails from s_u and 1 tail from s_{uv}, and so on. This yields the following number of possibilities for an edge core with k tails attached to one core vertex and m wedge fringes (where we choose i tails from s_{uv}):

$$\sum_{i=0}^{k}\binom{n_u}{k-i}\binom{n_{uv}}{i}\binom{n_{uv}-i}{m}$$

Next, let us add $l>0$ tails to the other core vertex. This requires the size n_v of a third disjoint vertex set s_v:

$$s_v=e(v)\setminus e(u),\, n_v=|s_v|$$

Here, n_v is the number of non-core neighbors of v that are not neighbors of u. The relationship between the now three sets can be visualized in a Venn diagram:



We can select the k tails of u from either n_u or n_{uv}, the l tails of v from either n_{uv} or n_v, and the m wedge fringes only from n_{uv}. Using the approach from above, where we sum over all possible ways to draw the tails from two sets, yields the following number of possibilities:

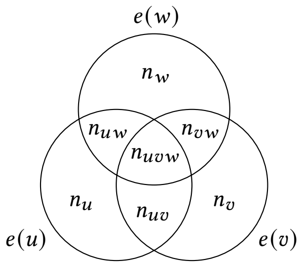
$$\sum_{i=0}^{k}\binom{n_u}{k-i}\binom{n_{uv}}{i}\sum_{j=0}^{l}\binom{n_v}{l-j}\binom{n_{uv}-i}{j}\binom{n_{uv}-i-j}{m}$$

The first two binomials choose the k tails of u as before, the next two binomials choose the l tails of v in a similar manner except the size of s_{uv} has been reduced by i, and the last binomial chooses the m wedges as before except the size of s_{uv} has been further reduced by j. Note that this formula is general and works for all patterns with a 2-vertex core, independent of the values of k, l, and m, that is, it works for infinitely many patterns just like the k-star formula does for 1-vertex cores.

3.2 3-Vertex Cores

There are two distinct 3-vertex cores, the wedge core and the triangle core. For example, the 4-cycle in Fig. 1 has a wedge core and the 4-clique has a triangle core. Aside from some differences in automorphisms, the two types of 3-vertex cores can be handled similarly. Note that 3-vertex cores introduce a new type of fringe, which we call "tri-fringe". Vertices O and P in Fig. 4 are examples of tri-fringes. They have degree 3.

Assuming the 3 core vertices are u, v, and w, we now need to consider the *sizes* of the 7 disjoint vertex sets that $e(u)$, $e(v)$, and $e(w)$ can form, which are visualized in the following Venn diagram:



This gives us four sets from which to draw tails (e.g., s_u, s_{uv}, s_{uw}, and s_{uvw} for the tails of vertex u), two sets from which to draw wedge fringes (e.g., s_{uv} and s_{uvw} for the wedge fringes between vertices u and v), and one set (s_{uvw}) from which to draw tri-fringes, where the sets are defined as above. Hence, our formula for computing the number of occurrences of the pattern has three summations and four binomials for each vertex with tails, one summation and two binomials for each edge with wedge fringes, and one binomial for the tri-fringes. As the formula is lengthy, we describe how the terms look on a few examples.

Since we can start with any fringe type, let us begin with the k tails of vertex u, which yields the following terms:

$$\sum_{a=0}^{k}\binom{n_u}{a}\sum_{b=0}^{k-a}\binom{n_{uv}}{b}\sum_{c=0}^{k-a-b}\binom{n_{uw}}{c}\binom{n_{uvw}}{k-a-b-c}\ldots$$

The last binomial does not require a sum as it must handle whatever number of tails remain.

We can continue with any other fringe type, but we must first subtract the already used elements from the shared sets. For instance, let us continue with the m wedge fringes between u and v. In this case, the next terms of the formula are:

$$\ldots\sum_{d=0}^{m}\binom{n_{uv}-b}{d}\binom{n_{uvw}-(k-a-b-c)}{m-d}\ldots$$

We subtract b from n_{uv} since we already used b vertices for the tails of u. We subtract $k-a-b-c$ from n_{uvw} for the same reason.

This procedure continues for all remaining fringe types. For example, the next binomial that uses n_{uv} must subtract b and d to account for the already used elements by earlier fringe types.

3.3 Larger Cores

Our approach supports arbitrary core sizes as follows. Assuming a matched core with q vertices, we start by computing the Venn diagram based on the q sets of external neighbors. This yields 2^q-1 disjoint set sizes.

Next, we iterate over all present fringe types. For each type, we perform a summation as described above over every Venn diagram entry that covers the corresponding anchor set and subtract the number of used entries before moving on to the next fringe type. We describe this process in more detail in the next subsection.

We also tried an alternative approach, called the inclusion - exclusion principle, that first overcounts and then subtracts the overcounted amount [18]. Although IEP is very efficient in simpler cases, its complexity increases as we apply it to a pattern with multiple fringe types. IEP also proved incompatible with several of our optimizations, resulting in us not seeing a benefit.

3.4 Implementation

Our Fringe-SGC code operates as follows. It starts by analyzing the user-provided pattern and separating it into core and fringe vertices. It employs the following serial heuristic to do this. It first processes all unprocessed degree-1 vertices and marks each of them as a tail fringe and the adjacent vertex as a core vertex. Then, it processes all unprocessed degree-2 vertices and similarly marks them as wedge fringes and their two adjacent vertices as belonging to the core. It continues in this manner until all vertices have been processed. If the resulting core ends up being disconnected, it moves a minimal number of fringe vertices into the core to make it connected. We do this by finding the shortest path between connected components of the core and adding vertices along this path to the core. The result is recorded in a so-called "matching order" that prescribes which core vertex to search for first, which core vertex is next, and so on. All of this work is not performance critical as it only needs to be done once per pattern and is independent of the graph.

Fringe-SGC invokes specialized code for patterns with small cores. If the pattern contains a single vertex, it returns the number of vertices in the input graph. If the pattern contains two vertices, it returns the number of undirected edges in the graph. If the pattern has a 1-vertex core, it uses the k-star formula on each graph vertex to count the number of occurrences. If the pattern has two or three core vertices, it applies the appropriate formula outlined above with specialized code to handle automorphisms and optimized code to compute the Venn diagram entries. For larger cores, it employs a general implementation.

The general Fringe-SGC code starts by searching for the core of the pattern using a conventional approach that is similar to STMatch [26]. Assuming the core has p vertices of which q vertices appear in at least one anchor set, Fringe-SGC proceeds by computing the Venn diagram for the q matched vertices. To boost performance, we only do this for the needed q vertices instead of for all p vertices.

Fringe-SGC stores the resulting $2^q - 1$ disjoint set sizes in an array with 2^q elements, where the first element is unused. Each element has a q-bit index. We interpret this index as a bitset in which each '1' bit indicates the presence and each '0' the absence of the corresponding vertex. For example, in case of $q = 3$ with core vertices u, v, and w, the array holding the Venn diagram looks as follows, where bit position 0 of the index corresponds to u, position 1 to v, and position 2 to w:

Binary Index	Array Element
000	-
001	n_u
010	n_v
011	n_{uv}
100	n_w
101	n_{uw}
110	n_{vw}
111	n_{uvw}

We chose this assignment to simplify the implementation. In particular, for each fringe type, the formula needs to sum over all Venn diagram entries that cover the anchor set, which can now easily be determined. For example, in case of a wedge fringe with an anchor set of u and w, which corresponds to the bitset 101, Fringe-SGC must sum over all array entries whose index has a '1' bit in positions 0 and 2, that is, the elements at indices 101 (n_{uw}) and 111 (n_{uvw}).

To efficiently find the needed indices for any fringe type while skipping over all other indices, we start with a bitset idx that corresponds to the anchor set $anch$ of the fringe type. Then, in each iteration, we compute the next bitset as $idx = (idx + 1) \mid anch$, where the vertical bar indicates a bit-wise OR operation (i.e., set addition). Fringe-SGC terminates the iteration when all bits are '1' in the bitset, i.e., when reaching the last element of the array. This final set is a superset of all fringe types and must always be visited. Recall that this set is special because no summation is needed for it. It simply handles all remaining fringes of the current type.

Fringe-SGC implements this approach using an iterative implementation of the recursive code shown in Listing 5. Whereas the recursive code is shorter and more elegant, we use iterative code because recursion is not well supported on GPUs as each of the many threads only has a very small stack. Before the code executes, the following variables must be computed: q (the number of core vertices that belong to an anchor set), $venn$ (the Venn diagram), s (the number of distinct anchor sets, i.e., fringe types), $anch$ (an array holding s bitsets specifying the anchor sets), and k (an array holding s counts specifying how many fringes of this type are present). Note that q, s, $anch$, and k are pattern specific and only need to be computed once. Only $venn$ needs to be computed for each match of the core in the input graph.

In Listing 5, the variable pos iterates over the s fringe types, rem is the remaining number of fringes of the current type that still need to be drawn from a set, and idx iterates over all sets that cover the current fringe type as described above.

The fringe-counting fc function is called on Line 27 for the first fringe type. The recursion stops on Line 5 when all fringe types have been processed. For each fringe type, the summation is performed in the loop between Lines 17 and 21. Line 19 in the body of this loop moves on to the next Venn diagram entry that covers the current fringe type. The last Venn diagram entry is handled separately on Lines 8 through 14 as it does not require a summation. Instead, it moves on to the next fringe type. Whenever the code draws fringes from a set, the corresponding entry in the Venn diagram is decremented accordingly (Lines 10 and 18) before the next recursive call and then incremented again (Lines 12 and 20) to undo the change after the recursive call returns.

We verified Fringe-SGC's correctness by comparing the number of occurrences it returns to the corresponding number returned by the other codes we evaluate in the results section. The numbers match on all tested inputs and patterns. Furthermore, we exhaustively tested Fringe-SGC on all possible patterns with up to 5 vertices on all possible graphs with up to 5 vertices (as well as some larger graphs).

3.5 Optimization

The code in Listing 5 contains several optimizations. For example, Line 6 moves on to the next fringe type as soon as no more fringes

```
1  long nCk(int n, int k); // computes "n choose k"
2
3  long fc(int pos, int rem, int idx)
4  {
5    if (pos == s) return 1; // end of recursion
6    if (rem == 0) return fc(pos + 1, k[pos + 1], anch[pos
         + 1]); // next fringe type
7    int vc = venn[idx];
8    if (idx == (1 << q) - 1) { // last entry of array
9      if (rem > vc) return 0; // no solution
10     venn[idx] -= rem;
11     long cnt = nCk(vc, rem) * fc(pos + 1, k[pos + 1],
        anch[pos + 1]); // next fringe type
12     venn[idx] += rem;
13     return cnt;
14   }
15   long cnt = 0;
16   int top = std::min(rem, vc);
17   for (int i = 0; i <= top; i++) { // summation loop
18     venn[idx] -= i;
19     cnt += nCk(vc, i) * fc(pos, rem - i, (idx + 1) |
        anch[pos]); // next Venn entry
20     venn[idx] += i;
21   }
22   return cnt;
23 }
24 ...
25 // start of recursion
26 long count = fc(0, k[0], anch[0]);
```

Figure 5: Recursive Fringe-SGC pattern-counting code

of the current type are needed, even before all sets that cover the current type have been visited. Line 9 stops the recursion if not enough elements are available rather than continuing and multiplying the return value on Line 11 by zero. Instead of summing over all *rem* values on Line 17, we only sum over the available values (Line 16) if it is smaller, again minimizing the number of recursive calls. Line 19 skips over non-matching sets as described above.

Note that Fringe-SGC never accesses any fringe vertices in the input graph. It only accesses the core vertices and their adjacency lists, which should improve locality. Moreover, it only reads the graph and adjacency lists while counting patterns. It does not store any information in the graph.

Many SGC frameworks incrementally search for the pattern [4]. They record where in the input graph the first part of the pattern can be found and then gradually "grow" the matches to the full pattern size, deleting any occurrences that do not match. This process is very memory intensive and, therefore, does not work for large patterns, including patterns with many fringes. So as not to run out of memory, Fringe-SGC adopts the stack approach of STMatch [26]. It tries to match the entire pattern for a given starting vertex in the graph before moving on to the next starting vertex. As a consequence, Fringe-SGC only requires enough memory to hold the input graph plus a fixed amount of memory per running thread that is a function of the pattern (see Section 3.7). Importantly, Fringe-SGC's memory consumption does not increase dynamically while it counts patterns.

3.6 Parallelization

Fringe-SGC is implemented in CUDA. Each thread counts the number of occurrences of the pattern when starting from some distinct graph vertices. These counts are then sum-reduced to obtain the total number of occurrences. Since the search for the core and the execution of the fc function can stop at any point and their running times as well as that of computing the Venn diagrams depend on the degrees of the matched vertices, we employ a dynamic schedule to balance the load between the threads.

We studied different ways to compute the needed set sizes (i.e., the Venn diagram entries) on a GPU and settled on the following approach, which seems to deliver good performance. For each anchor vertex on the stack, we use an entire warp (i.e., a group of 32 contiguous threads), to process the entries in its adjacency list. For each entry of the Venn diagram, we search the adjacency lists of the anchor vertices that appear later in the stack to determine which of them also contain the entry. Since the adjacency lists are sorted, we can do so with binary search. This approach is surprisingly efficient because all warp threads search the same adjacency list in parallel and typically search it for similar values (since the values stem from a chunk of another sorted adjacency list). Hence, many of the logarithmic steps of the binary search yield coalesced memory accesses. Once the Venn diagram has been populated with the resulting counts, we computationally correct them to account for the fact that we only compared to adjacency lists of vertices that appear later in the stack. This is about twice as fast as always checking all adjacency lists. Finally, we subtract the counts for the core vertices (i.e., the vertices on the stack) since they are not external neighbors. Note that, for performance reasons, Fringe-SGC does not generate any vertex sets, it only computes their sizes.

Each thread starts from a distinct graph vertex and tries to match the core of the pattern by following the matching order. To speed up the process, the matching order is sorted from most to least constrained core vertex (while maintaining connectivity). This ensures that the matching fails as soon as possible (if at all), which minimizes wasted work. As a simple example, consider the tailed triangle from Fig. 1. Of the two core vertices, vertex D has the higher degree, so we first try to match the current graph vertex to pattern vertex D. This is more likely to fail than first matching the current vertex to pattern vertex A, which has a lower degree. There are other constraints, too. For instance, vertices appearing later in the matching order must be adjacent to one or more earlier vertices, and the matched vertices must all be distinct. This stack-based method is tantamount to a set of nested conditional statements where the probability of reaching a given nesting level decreases with each additional level. Listing 6 provides an excerpt of such code.

This poses a performance problem on GPUs because it causes thread divergence and, consequently, loss of parallelism. To maintain parallelism in such code structures, Fringe-SGC employs the strategy outlined in Listing 7.

In each level, the 32 warp threads concurrently process 32 vertices. Next, they check which threads in the warp (called "lanes") have a vertex that meets the requirements and, crucially, all lanes process the first such vertex together, then the next such vertex, etc. In this manner, the code maintains parallelism at each level and

```
for (v1 = adj[v0].start; v1 < adj[v0].end; v1++) {
  if (v1 meets requirements r1) {

    for (v2 = adj[v1].start; v2 < adj[v1].end; v2++) {
      if (v2 meets requirements r2) {
        ...
      }
    }
  }
}
```

Figure 6: Nested code structure for finding patterns

all lanes reach the innermost level if at least one core match exists. We found this approach to greatly improve performance on GPUs.

When the code reaches the innermost level, the entire warp populates the Venn diagram as described above. The warp does this for up to 32 vertices, creating up to 32 Venn diagrams. Then, each lane runs the iterative fc function that computes the number of possibilities for the fringes based on the counts in the Venn diagram. In summary, all steps of the Fringe-SGC code execute in parallel using a mixture of warp-based and thread-based parallelization.

If the input does not fit on a single GPU, it would have to be partitioned. Each partition would need a ghost region that is as wide as the diameter of the search pattern, which is at most the size of the pattern. This way, multiple GPUs can process the partitions independently and at the same time.

3.7 Storage and Work Complexity

In addition to the shared input graph and the shared k array, each thread in Fringe-SGC requires a stack, *venn* array, and a local copy of the k array that the thread can modify. The storage requirement is $O(2^q)$, which is exponential in the pattern size, but real-world patterns are so small (up to a dozen vertices or so) that the thread-local storage tends to be small in comparison to the memory needed to hold the input graph. In all tested cases, the combined memory consumption of the Venn diagrams across all GPU threads is under 5 MB, irrespective of the size of the input graph, which is a tiny fraction of the GPU’s global memory size.

The work complexity of Fringe-SGC is exponential in the pattern size just like all other SGC approaches we are aware of. At first glance, this appears to be no improvement over conventional SGC approaches. However, Fringe-SGC provides a key benefit: *its exponent is lower for patterns with multiple fringes of the same type.* Whereas other approaches are exponential in the number of pattern vertices, Fringe-SGC is exponential in the number of core vertices plus the number of fringe types, which is necessarily less than or equal to the number of pattern vertices. In other words, Fringe-SGC is exponentially faster than prior approaches on some patterns. For example, its running time remains nearly stable when adding more wedge fringes to the pattern in Fig. 4.

4 Related Work

Chen et al. [4] present Pangolin, the first SGC system that provides high-level abstractions for GPU processing. It targets both shared-memory CPUs and GPUs. Pangolin uses an “extend-reduce-filter” execution model to process embeddings. During the extend phase, the embeddings are expanded, and the resulting embeddings form the output worklist. The reducer phase extracts pattern-based statistical information, such as pattern frequency or support, and the filter phase removes irrelevant embeddings to improve efficiency. Pangolin uses an optimized structure of arrays (SoA) to take advantage of locality when storing embeddings. This approach eliminates the need for creating temporary embeddings, and by blocking the schedule of embedding exploration, it reduces the memory footprint. Additionally, Pangolin utilizes inspection-execution and scalable memory allocation techniques to address the overheads associated with dynamic memory allocation.

Chen et al. [3] further proposed a two-level framework for mining graph patterns, called Sandslash, which is included in GraphMiner. The high-level component autonomously conducts a search for a graph-pattern-mining problem specification, without requiring user input. The low-level component offers an API that allows users to customize the search strategy and improve the efficiency of the framework. At the high level, Sandslash employs a matching order and vertex extension to prune the search tree and avoid expensive isomorphism tests. The framework also uses a standard technique for symmetry breaking to prevent over-counting. To efficiently navigate the search space and identify subgraphs and patterns, Sandslash employs parallel depth-first search (DFS) exploration and degree filtering to terminate the search earlier. It also uses memorization of neighborhood connectivity to avoid repeating lookups in the search graph. At the low level, users only need to understand the subgraph tree abstraction and how to prune it, without needing to know the actual implementation.

It is sometimes possible to compute the number of occurrences of a pattern from the counts of other patterns. This technique is called *local counting*. It works by decomposing the pattern into smaller subpatterns, each of which can be counted independently. For example, the number of 2-stars (wedges) can be counted based on the number of triangles and the degrees of two adjacent vertices. The counts are then combined mathematically to obtain the final count of the subgraph pattern in the original graph. Sandslash supports local counting at the lower level, though the user has to provide the formula to be used [3]. Suganami et al. [21] published over 20 such formulas, and ESCAPE [16] employs local counting for all patterns with up to 5 vertices. Note that local counting does not target fringes and is orthogonal to our approach. It is, however, similar in that it computes the number of occurrences of a pattern rather than counting them.

Xiang et al. [28] proposed a new subgraph isomorphism framework called cuTS, which works for both directed and undirected graphs. Their approach involves constructing a partial path by adding vertices from the actual graph that match the vertices in the pattern, starting with the vertex that has the maximum degree. To store intermediate results efficiently, they propose a space-efficient CSF-based data structure that allows for reusing prefix paths multiple times. They also use a hybrid approach of BFS and DFS to scan partial paths efficiently. For efficient neighbor intersection, they load all children of a vertex into a buffer and use a warp to read them in a coalesced manner from global to shared memory.

Shi et al. [12] proposed a parallel CPU graph-pattern-matching framework called Dryadic. It utilizes a computation tree, i.e., a

```
for (v1 = adj[v0].start + lane; v1 < adj[v0].end; v1 += warpSize) { // warp-based
  active1 = (v1 meets requirements r1);
  bal1 = __ballot_sync(~0, active1); // determines which lanes have a vertex that meets the requirements
  while (bal1 != 0) {
    old1 = bal1;
    bal1 &= bal1 - 1; // removes least-significant set bit
    who = __ffs(bal1 ^ old1) - 1; // determines location of removed bit
    if (who == lane) stack[1] = v1; // saves v1 of the currently selected lane of the warp
    __syncwarp(); // forces remaining lanes to wait for stack update to be visible

    x = stack[1];
    for (v2 = adj[x].start + lane; v2 < adj[x].end; v2 += warpSize) { // warp-based
      active2 = (v2 meets requirements r2);
      bal2 = __ballot_sync(~0, active2); // determines which lanes have a vertex that meets requirements
      while (bal2 != 0) {
        old2 = bal2;
        bal2 &= bal2 - 1; // removes least-significant set bit
        who = __ffs(bal2 ^ old2) - 1; // determines location of removed bit
        if (who == lane) stack[2] = v2; // saves v2 of the currently selected lane of the warp
        __syncwarp(); // forces remaining lanes to wait for stack update to be visible
        ...
      }
    }
  }
}
```

Listing 7: Modified nested code structure for finding patterns that maintains parallelism

tree-structured intermediate representation, to enable the detection of multiple patterns simultaneously. The computation tree serves as the foundation for Dryadic's three main components: tree construction, tree optimization, and execution. Dryadic's pattern matching process computes a matching order and restrictions to break symmetry. The matching order identifies the dependencies of computations to find the vertices in an embedding, while the restrictions enforce "ID-is-larger" relations between some vertices to avoid identifying the same embedding multiple times. To minimize redundant computations, Dryadic merges computation trees of different patterns. To eliminate the redundancy completely, they move the set operations to the upper level of the tree, which they refer to as operation motion. Moreover, they employ a 3-field data structure. The data structure consists of the in-field, which includes all pattern vertices connected to vertex i that appear before i in the matching order, the out-field, which contains all pattern vertices disconnected from vertex i that appear before i in the matching order, and the res-field, which contains the ID of the pattern vertices restricting vertex i. Dryadic supports DFS to minimize the memory footprint and improves load balance by using fine-grained work-stealing based on the computation tree and embedding tree abstraction. It also offers two modes for executing the computation tree: "Galois-based Interpretation" and "Code Generation for Parallel and Distributed Execution".

Wei et al. [26] recently presented STMatch, the first stack-based GPU graph-pattern-matching system that requires no synchronization during the search and avoids the memory consumption issues of previous systems. Its stack-based approach simulates a recursive procedure and maintains three arrays to store candidate vertices, candidate sizes, and loop iterations. It also employs a pre-processing technique called backward search, which reduces the search space and improves overall performance. In STMatch, each GPU warp runs a while loop independently, and a two-level work-stealing technique is used to balance the workload among the warps. They use two-level stealing because selecting the best target to steal from all warps is expensive, whereas finding a good target within the thread block is much easier. Since each warp will perform one while loop at a time, each loop might not have enough operations for all threads in the warp, which may cause some or most of the threads to be idle. To improve thread utilization, loop unrolling is used to allow threads to perform operations together. The operation motion technique from Dryadic is also implemented to reduce redundancy.

Yuan et al. [29] improved upon STMatch. Their approach, called T-DFS, decomposes the computation into tasks and distributes them in parallel. The tasks are initially distributed evenly across all warps. Straggler tasks are then assigned based on a timeout mechanism. T-DFS also improves upon the task queue of STMatch by making enqueue and dequeue operations atomic and lock-free.

Shi et al. [19] recently proposed GraphSet, a CPU and GPU supported set-based approach. It employs the inclusion-exclusion principle (IEP) to more efficiently count the instances of patterns rather than iterating through all nested loops. In order to accomplish this on a general SGC application, GraphSet leverages set properties to generate a transformation-friendly schedule based on the entire input pattern. Then, with dependence analysis, GraphSet performs rescheduling and extracts loop variables that can be reduced. These loop variables are vertices in the input pattern that are not directly connected and are said to not require an intersection operation in their innermost loops. The control flow of these vertices is then transformed into set operations that result in better performance for patterns with many opportunities for this optimization. This extraction of unconnected loop variables is a step in the direction

Table 1: Information about the input graphs

Graph name	Type	Source	Vertices	Edges	d_avg	d_max
amazon0601	co-purchases	SNAP	403,394	2,443,408	12.1	2,752
coPapersDBLP	publication citations	SMC	540,486	30,491,458	56.4	3,299
delaunay_n22	triangluation	SMC	4,194,304	25,165,738	6.0	23
in-2004	web links	SMC	1,382,908	13,591,473	19.7	21,869
internet	Internet topology	SMC	124,651	193,620	3.1	151
kron_g500-logn20	Kronecker	SMC	1,048,576	89,238,804	85.1	131,503
rmat16.sym	RMAT	Galois	65,536	483,933	14.8	569
soc-LiveJournal1	journal community	SNAP	4,847,571	85,702,474	17.7	20,333
uk-2002	Web links	SMC	18,520,486	523,574,516	28.3	194,955
USA-road-d.NY	road map	Dimacs	264,346	730,100	2.8	3

of our fringe vertices, making GraphSet probably the most closely related work. However, our approach is more general and often yields higher performance, as our results show.

We compare the performance of Fringe-SGC to several of the above frameworks in the result section. We omit Dryadic because its code is not open-sourced. There are many additional SGC frameworks in the literature, including AutoMine [13], Kaleido [31], G-miner [2], Arabesque [22], and GSI [30]. They either only support predefined patterns or have been shown to be slower than the codes we do compare with. For example, GraphSet has been shown to be faster than Sandslash [3], GraphPi [20], cuTs [28], and G-Miner [2]. As is commonly done in most of the related work, Fringe-SGC also employs a matching order, symmetry breaking, and degree filtering to speed up the search. Its engine that searches for the pattern's core is based on STMatch's stack-based approach to minimize memory usage. However, no prior work incorporates a formula to quickly count the fringes of patterns.

5 Evaluation Methodology

We compare the performance of Fringe-SGC with that of the GPU implementations of STMatch [26], GraphSet [19], and T-DFS [29]. We instrumented each code to measure the subgraph counting time, excluding reading in the graph. We evaluated the codes on an RTX 3080 Ti with 10,240 cores distributed over 80 multiprocessors. Each multiprocessor has 128 kB of L1 cache, and all have access to 6 MB of shared L2 cache. The global memory has a capacity of 12 GB. We also experimented with other GPU generations, which yielded consistent results. Hence, we only show results for the 3080 Ti.

We use the evaluated SGC codes to count all edge-induced subgraphs that are isomorphic to the search pattern in a given input graph. All codes that we compare to only count instances of these subgraphs, and do not list their embeddings. Given a core of vertices, fringe vertices are added in additional user-defined subgraphs for tests with incrementally larger subgraphs. We start with a single core vertex, and add fringes until there are 6 fringe-vertices, and a total of 7 in the subgraph. We then move to a 2-vertex core that is connected, and add fringes to all anchor sets incrementally, and observe how runtimes change as we count larger subgraphs. We do this for all connected cores with up to 3 vertices. We are limited in the number of vertices in a subgraph for these tests, the other codes place a hard limit to 7 vertices. We use the 10 graphs shown in Table 1 as inputs. We selected them because they represent a range of types, sizes, average degrees, and maximum degrees.

6 Results

In this section, we evaluate the performance of Fringe-SGC and compare it to that of the leading GPU codes from the literature. We show throughputs (i.e., the number of edges in the input graph divided by the measured running time) to normalize the results and because throughput is a higher-is-better metric, which is more intuitive. Some throughputs are not shown, as these codes did not finish counting a given subgraph in the allotted time of half an hour per graph input.

6.1 Performance Comparison

First, we compare our Fringe-SGC code to STMatch, T-DFS, and GraphSet, the fastest GPU codes from the literature. Figure 8 shows the results for patterns with a 1-vertex core, i.e., for k-stars. The x-axis lists the subgraphs and the y-axis the geometric-mean throughputs across our 10 graph inputs. Note that the y-axis uses a logarithmic scale to account for the large differences in throughput. We do not show results for codes where more than one input times out.

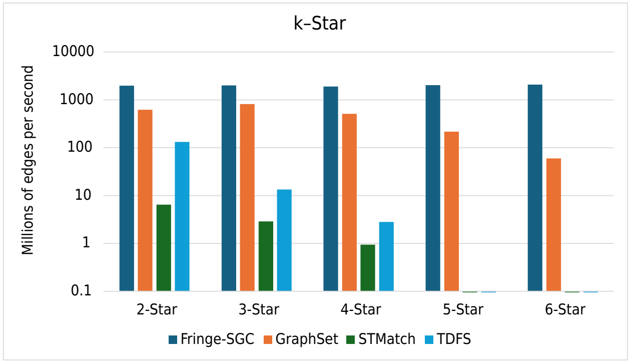


Figure 8: Throughputs for Patterns with a Vertex Core

Our approach consistently outperforms the other three GPU codes on the k-stars. Moreover, the throughput of Fringe-SGC is approximately constant whereas the throughputs of GraphSet, T-DFS and STMatch decrease as k (the number of spokes in the stars) increases. Hence, the performance advantage of Fringe-SGC increases with increasing number of fringes. The resulting geometric-mean speedups over GraphSet range from 1.64× with 2-stars to 18.76× with 6-stars. The speedup over STMatch is 164.99× with 2-stars and 1064× on 4-stars. The speedup over T-DFS ranges from 5.99× to 516.23×. Looking at the individual inputs (results not shown), we find that the performance advantage of Fringe-SGC increases for larger graphs. On our largest input, it is over an order of magnitude faster than any other tested code.

Figure 9 shows throughputs for patterns with a 2-vertex core. We systematically add fringe vertices until the 7-vertex limit of the other codes is reached. The x-axis lists the subgraphs, and the y-axis displays the throughputs in the millions of edges per second.

The trends are the same as for the k-stars. Fringe-SGC delivers a near-constant throughput when we add more fringe vertices to the 2-vertex core, whereas the other codes' performance drops. Note that we could add more fringe vertices without significantly affecting Fringe-SGC's throughput, but the other codes do not

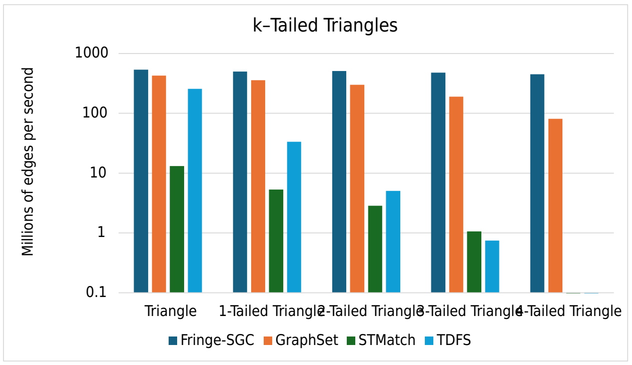


Figure 9: Throughputs for Patterns with an Edge Core

support larger patterns. The geometric-mean speedups range from 1.07× to 4.7× compared to Graphset, 41.72× to 465.25× compared to STMatch, and 1.96× to 664× compared to T-DFS.

Figure 10 shows throughputs for patterns with a triangle core. The geometric-mean speedups compared to GraphSet range from 0.6× (a slowdown) for the 4-clique, which only has a single fringe, to a speedup of 2.89× for the 3-tailed 4-clique, which has 4 fringe vertices. The speedups compared to STMatch range from 10.37× to 158.39×. Compared to T-DFS, the speedups range from 2.2× to 49×. It is important to note that, although the speedups are lower than with a single-core vertex, we are now testing a larger core with fewer fringe vertices due to the other codes' limit of 7 vertices per subgraph. We would expect to see higher speedups if we were able to run the third-party codes with more fringe vertices.

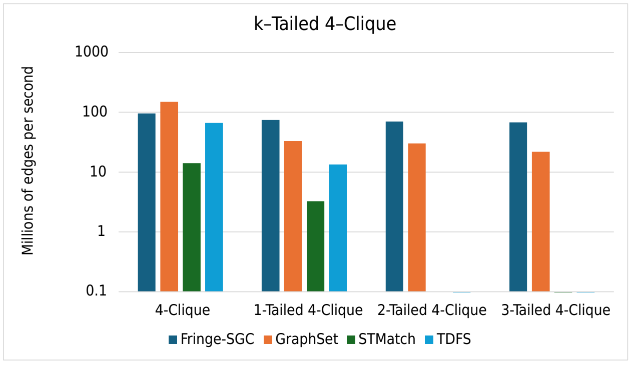


Figure 10: Throughputs for Patterns with a Triangle Core

Figure 11 shows throughputs for patterns with a wedge core. We see similar results to the triangle core, in which speedups range from 0.6× to 4.35× compared to GraphSet, 88.77× to 534.95× compared to STMatch, and 40.64× to 155.97× compared to T-DFS. Again, the performance benefit increases with increasing fringe counts.

In summary, the results in this subsection show that Fringe-SGC is often faster than GraphSet and always faster than STMatch and T-DFS. Importantly, the performance advantage of Fringe-SGC tends to increase with increasing numbers of fringes. When more fringe vertices are added to a pattern, GraphSet's, T-DFS's, and STMatch's

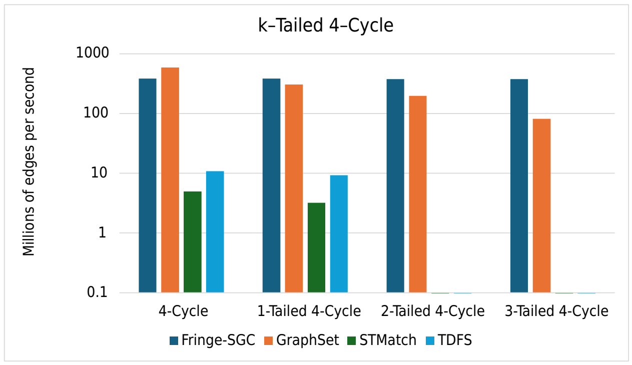


Figure 11: Throughputs for Patterns with a Wedge Core

throughputs tend to decrease whereas Fringe-SGC's throughput remains roughly stable as expected.

6.2 Systematic Addition of Fringes

The key strength of Fringe-SGC is that its performance degrades exponentially less when adding more fringes of the same type compared to any other state-of-the-art SGC approach. Unfortunately, this benefit mostly manifests on patterns that are too large for other SGC frameworks. Hence, we can only show throughputs for Fringe-SGC in this subsection.

To study this behavior, we systematically add more and more fringes of a specific type to a triangle core. We start with the subgraph from Figure 4, a pattern that is already too large to be counted by any other state-of-the-art SGC code, and progressively add different types of fringe vertices. Figure 12 shows the throughputs when adding more tails, Figure 13 when adding more wedge fringes, and Figure 14 when adding more tri-fringes.

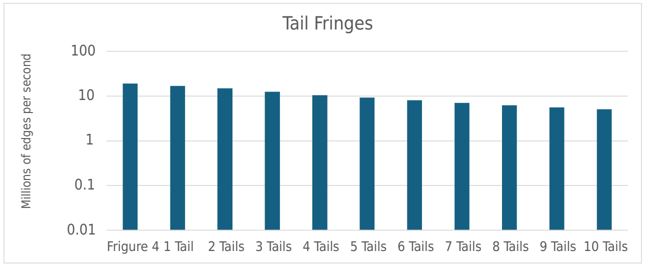


Figure 12: Throughput when Adding Tail Fringes

In all three cases, the throughput only changes marginally when adding up to 10 additional vertices to the pattern, demonstrating that Fringe-SGC's performance remains largely stable when including more fringes. This is in contrast to other SGC frameworks, where the performance drops so much that they do not support patterns of this size. In fact, these codes do not even support the starting subgraph from Figure 4 because it already has 16 vertices. Fringe-SGC is easily able to support 10 more vertices without an exponential change in runtime. Nevertheless, its performance does drop a little when adding more fringes. This is because, for example, adding a fringe vertex to the pattern from Figure 4 requires

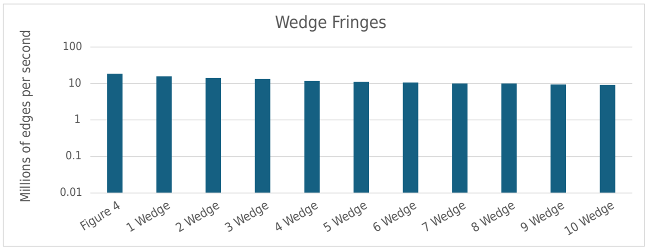


Figure 13: Throughput when Adding Wedge Fringes

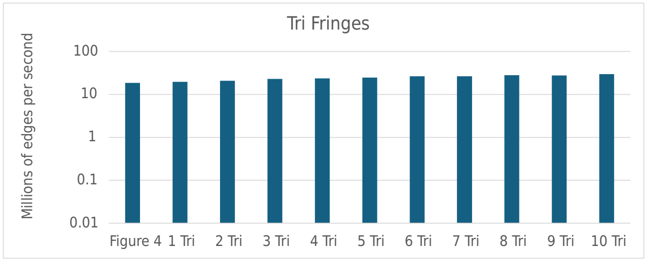


Figure 14: Throughput when Adding Tri-Fringes

an additional iteration when performing the summation over the corresponding anchor sets. In the case of adding tails, the 10 extra fringes lower the performance by under a factor of 3.5, which is less than the drop due to adding just 4 fringe vertices with any of the other studied SGC codes (see Section 6.1). Adding 10 wedge fringes results in an even smaller drop, and adding 10 tri-fringes yields a speedup of 1.56× as there are fewer core matches.

6.3 Per-Input Results

In this subsection, we present detailed results for kron_g500-logn20 in Figure 15. We selected this input because it has the highest average degree, the widest degree distribution, and one of the largest sizes of all evaluated inputs. These features help expose weaknesses and highlight optimizations. The figure combines results for patterns containing vertex, edge, and triangle cores.

Across all codes, the throughput drops whenever we add a vertex to a search pattern. For GraphSet, STMatch, and TDFS, this happens regardless of whether the added vertex is a core or fringe vertex. However, for Fringe-SGC, a significant throughput drop only occurs when adding a core vertex, showcasing the benefit of our approach.

On this input, Fringe-SGC is at least 1.06×, 7.78×, and 1.99× faster and at most 240.1×, 2334.0×, and 961.1× faster than GraphSet, STMatch, and TDFS, respectively. The geometric-mean speedup over these codes is 15.66×, 134.8×, and 43.82×. Note that there is not a single pattern where Fringe-SGC is slower. This is in contrast to our inputs with smaller average and maximum degrees, especially on patterns with few fringe vertices, where other codes are sometimes faster. On such inputs, there is often not enough parallelism for an entire warp due to the lower degrees. Moreover, any advantage of using Fringe-SGC's summations is lost on vertices with insufficient degree to match a vertex in the search pattern's core. On higher-degree inputs like kron_g500-logn20, hub vertices introduce combinatorial explosions for fringe vertices, making it the ideal topology to demonstrate the strengths of Fringe-SGC.

7 Conclusion

Subgraph Counting (SGC) is an important graph analysis that has broad applications in various domains. Most existing SGC algorithms incrementally search for the pattern, which is memory intensive and has exponential work complexity. This problem is particularly pronounced for patterns with fringes. As a consequence, most existing frameworks only support standard patterns like triangles, cliques, and small motifs. In this paper, we introduce Fringe-SGC, a framework for efficiently counting patterns with fringes. Our approach accomplishes this by conventionally searching only for the core of the pattern and employing a formula to compute the occurrences that the remaining pattern vertices add, which we call fringes. We evaluated Fringe-SGC both on standard patterns as well as on several special patterns with fringes. Note that all patterns with at least 2 vertices have at least one fringe vertex. The experimental results show that our approach outperforms the leading GPU implementations on various kinds of patterns. When counting patterns with multiple fringes, Fringe-SGC outperforms the three fastest GPU implementations by an average of 21.91×. We cannot evaluate the speedup on more complex patterns with fringes because none of the other frameworks support them. However, Fringe-SGC is not only able to process them but does so at near constant throughput, regardless of how many fringes are present.

Acknowledgments

We thank the reviewers for their valuable feedback. This work has been supported by the National Science Foundation under Award Number 1955367.

References

[1] Hongxu Chen, Yicong Li, and Haoran Yang. 2021. Graph Data Mining in Recommender Systems. In *Web Information Systems Engineering – WISE 2021*, Wenjie Zhang, Lei Zou, Zakaria Maamar, and Lu Chen (Eds.). Springer International Publishing, Cham, 491–496. https://doi.org/10.1007/978-3-030-91560-5_36

[2] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large scale graph mining with g-miner. In *Proceedings of the 2019 International Conference on Management of Data*. 1881–1884. https://doi.org/10.1145/3299869.3320219

[3] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/3447818.3460359

[4] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (April 2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[5] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 581–594. https://doi.org/10.1109/ISCA52012.2021.00052

[6] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. https://doi.org/10.5441/002/edbt.2015.15

[7] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1036–1050. https://doi.org/10.1109/TKDE.2005.127

[8] Katherine Faust. 2010. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks* 32, 3 (2010), 221–233. https://doi.org/10.1016/j.socnet.2010.03.004

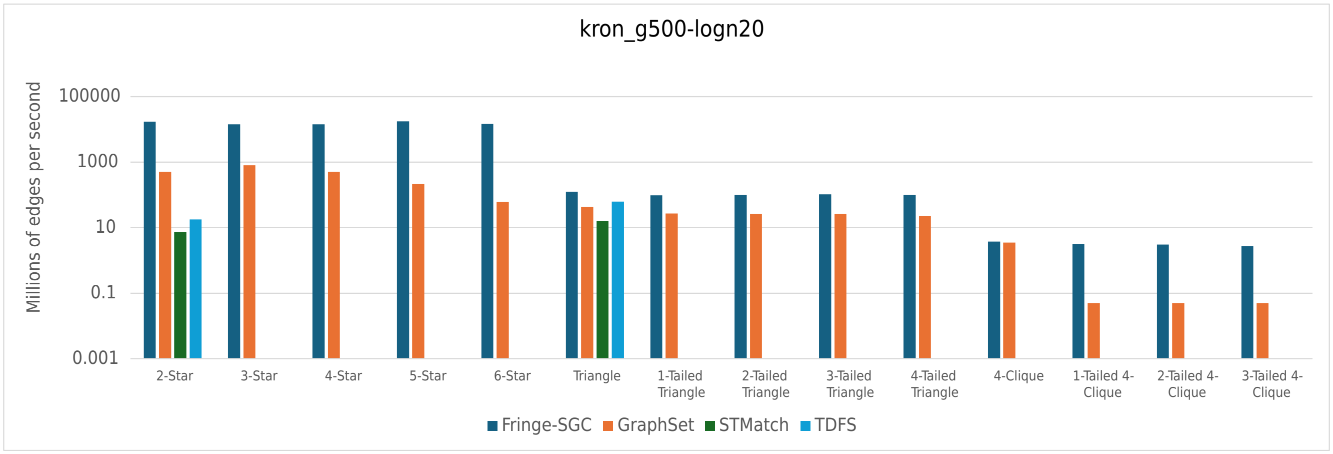


Figure 15: Throughput of all tested cores in kron_g500-logn20

[9] Máté Fellner, Bálint Varga, and Vince Grolmusz. 2019. The frequent subgraphs of the connectome of the human brain. *Cognitive Neurodynamics* 13, 5 (May 2019), 453–460. https://doi.org/10.1007/s11571-019-09535-y

[10] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105. https://doi.org/10.1017/S0269888912000331

[11] Mohammad Mehdi Daliri Khomami, Alireza Rezvanian, Ali Mohammad Saghiri, and Mohammad Reza Meybodi. 2020. Distributed Learning Automata-Based Algorithm for Finding K-Clique in Complex Social Networks. In *2020 11th International Conference on Information and Knowledge Technology (IKT)*. 139–143. https://doi.org/10.1109/IKT51791.2020.9345622

[12] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. 2024. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques* (Atlanta, GA, USA) *(PACT '21)*. IEEE Press, 289–303. https://doi.org/10.1109/PACT52795.2021.00028

[13] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 509–523. https://doi.org/10.1145/3341301.3359633

[14] Tijana Milenković and Natasa Przulj. 2008. Uncovering biological network function via graphlet degree signatures. *Cancer Inform* 6 (2008), 257–273. https://doi.org/10.4137/CIN.S680

[15] Phuong Dao Noga Alon, Fereydoun Hormozdiari Iman Hajirasouliha, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discover by color coding. https://doi.org/10.1093/bioinformatics/btn163

[16] Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. 2017. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. (2017), 1431–1440. https://doi.org/10.1145/3038912.3052597

[17] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. 2021. A Survey on Subgraph Counting: Concepts, Algorithms, and Applications to Network Motifs and Graphlets. *ACM Comput. Surv.* 54, 2, Article 28 (March 2021). https://doi.org/10.1145/3433652

[18] S Sankaranarayanan and H Bennett. 2015. *Counting problems and the inclusion-exclusion principle.* Technical Report. Department of Computer Science, University of Colorado.

[19] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (, Denver, CO, USA,) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 32. https://doi.org/10.1145/3581784.3613213

[20] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC41405.2020.00104

[21] Shuya Suganami, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2020. Accelerating All 5-Vertex Subgraphs Counting Using GPUs. In *Database and Expert Systems Applications*, Sven Hartmann, Josef Küng, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil (Eds.). Springer International Publishing, Cham, 55–70. https://doi.org/10.1007/978-3-030-59003-1_4

[22] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. https://doi.org/10.1145/2815400.2815410

[23] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. 2017. Parallel triangle counting and k-truss identification using graph-centric methods. *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (2017). https://doi.org/10.1109/HPEC.2017.8091037

[24] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. 2016. A Comparative Study on Exact Triangle Counting Algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing* (Kyoto, Japan) *(HPGP '16)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2915516.2915521

[25] Yuanxu Wang, Song Jinmiao, Mingjie Wei, and Xiaodong Duan. 2023. Predicting Potential Drug–Disease Associations Based on Hypergraph Learning with Subgraph Matching. *Interdisciplinary Sciences: Computational Life Sciences* 15 (03 2023). https://doi.org/10.1007/s12539-023-00556-0

[26] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41404.2022.00058

[27] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, and Bo Long. 2021. Graph Neural Networks for Natural Language Processing: A Survey. https://doi.org/10.48550/ARXIV.2106.06090

[28] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 69. https://doi.org/10.1145/3458817.3476214

[29] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster Depth-First Subgraph Matching on GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3151–3163. https://doi.org/10.1109/ICDE60146.2024.00244

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

[31] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. 2020. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 673–684. https://doi.org/10.1109/ICDE48307.2020.00064