Exact and Parallel Triangle Counting in Dynamic Graphs

Devavret Makkar, David A. Bader, Oded Green
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332

Abstract— **Triangle counting is an important building block for finding key players in a graph. It is an integral part of the popular clustering coefficient analytic and can be used for pattern matching in social networks. A triangle, which is also a 3-clique, represents a strong connection between three players that are all connected. While counting triangles is not overly expensive from a computational standpoint, especially in comparison to centrality metrics (such as betweenness centrality and closeness centrality), it can still prove to be prohibitive for large scale networks, especially for those with a power-law distribution. This problem only deepens for dynamic graphs where the network is constantly changing, requiring constant updating of the graph and the analytic. In this paper, we present a new dynamic graph algorithm for counting triangles that is based on an inclusion-exclusion formulation. While our algorithm is independent of the computing platform, we show performance results on an NVIDIA GPU. Our approach handles 32 million updates per second, or up to 11 million updates per second if the graph data structure is also updated. In past approaches, when a vertex was affected due to an edge insertion or deletion, it was necessary to find the triangles from scratch for that given vertex. Our new formulation does not need this and only requires considering the affected edges. As such our algorithm is typically several hundred times faster than the past approach - in some cases up to 819X faster.**

I. Introduction

Dynamic graphs are ubiquitous and used to represent evolving data sets in numerous applications. This can include representing transactions between entities in a financial network, players in a social network, or message passing in a communication network. Constant changes to the graph require that the graph data structure be mutable. A dynamic graph data structure is only one out of two components needed for updating an analytic. The second component is the dynamic graph **algorithm** which is responsible for updating the metric of interest. Ideally, the dynamic graph algorithm should be computationally inexpensive in comparison with its static graph counterpart (which typically starts the computation from scratch). In some cases, approximation are a viable solution, yet there are also applications where the exact result is necessary. Thus, it is expected that the dynamic graph algorithm **produce the same result** as that of a static graph algorithm.

In this work we show a new parallel and computationally efficient algorithm for finding the **exact number** of triangles in a graph in a dynamic setting. This new algorithm uses an **inclusion-exclusion** formulation, introduced later in this paper, to ensure that triangles are counted accurately. Furthermore, our algorithm can process batches of edge updates, including when a triangle is either formed (in the case of edge insertion) or removed (in the case of edge deletions) within the batch. This requires special attention. The inclusion-exclusion formulations helps reduce the computational requirements in comparison to past solutions.

There are numerous approaches and algorithms to count triangles in the static setting (Sec. II); however, most of these approaches can be grouped into a small number of computational approaches [37]: iterating through node triplets, using linear algebra operations, and intersecting the adjacency lists of two connected vertices. Our new dynamic graph algorithm uses the latter. A dynamic graph analytic can start its execution from an empty graph or from some initial snapshot. For the latter case, typically, a static graph algorithm is executed and is then followed by the dynamic graph algorithm after each update. The new algorithms supports both insertions and deletions. In the literature, these updates might be called incremental and decremental operations.

Contributions

While the new algorithm in this paper is hardware independent, our implementation is on an NVIDIA GPU using the cuSTINGER [16] dynamic graph library. cuSTINGER is a scalable dynamic graph data structure for the GPU (a necessary feature for our dynamic graph algorithm). Our reasons for selecting cuSTINGER are as follows: 1) the data layout of cuSTINGER enables efficient adjacency list intersections, 2) the GPU is a massively multi-threaded system with high memory bandwidth allowing to test the scalability of our new algorithm, and 3) it is open-source.

In summary, the contributions of this paper are:
• We present a new dynamic graph algorithm for triangle counting based on an inclusion-exclusion formulation. Our new algorithm has better complexity bounds than prior approaches (Sec. IV).
• Our algorithm handles very high update rates, tens of millions of updates per second (Sec. V).
• Unlike the update process in [16], which does not ensure that the adjacency lists are sorted, our update process does ensure this. We show an efficient edge update processes that ensures this (Sec. III).
• All the source code of our algorithm and the modifications made to the dynamic graph data structure are open-source.

While the implementation and performance analysis of our new algorithm are presented for the GPU, we emphasize that the new algorithm can be implemented for

additional systems (CPU included) given an appropriate dynamic graph data structure.

Lastly, we note that the new algorithm is significantly faster than prior dynamic triangle counting algorithms. We typically see speedups of well over $100X$ and in some cases as much as $819X$ over past algorithms.

II. Related Work

The applications in which triangle counting and triangle listing are used is broad, though its adoption by data scientists as an important metric came after the introduction of the clustering coefficients [42] metric. Triangle counting was also part of the HPEC Graph Challenge [33]. Other applications for triangle counting are: finding transitivity [29], spam detection in email networks [4], finding tightly knit communities [32], finding trusses k-trusses [7], [39], [17] , and evaluating the quality of different community detection algorithms [26], [45]. An extended discussion of triangle counting applications can also be found in [6].

a) Computational Approaches: Given a graph, $G = (V, E)$, where V are the vertices and E are the edges in the graph, the three simplest and mostly widely used approaches for counting triangles in a graph [37]; enumerating over all node triplets $O(V^3)$, using linear algebra operations $O(V^w)$ (where $w < 2.376$), and adjacency list intersection (this can also be done using hash tables). The time complexity for the adjacency list intersection is implementation dependent. Using a set intersection operation for all edges in the graph gives an upper-bound time complexity of $O(|E| \cdot d_{max})$ where d_{max} is the degree of the largest vertex in the graph. A similar time complexity can be found for hash-table based intersections. Hash tables can be used when the adjacency lists are not sorted; however these tend to incur additional runtime and storage overheads. List intersections can also be completed in the Gather-Apply-Scatter (GAS) programming models [14], [27], [44]. While the GAS model enables good scalability, in practice performance and utilization tend to be relatively low due to communications overheads. A comparison of triangle counting in a vertex centric setting using sorted adjacency lists and a GAS implementation can be found in [9]. In this work, it was shown that the vertex centric approach is approximately $10X$ faster than its GAS counterpart. This performance difference is only likely to increase for a distributed system, as the GAS model requires more communication.

The applicability of triangle counting and clustering coefficients has lead to a proliferation of algorithms and unique implementations. Static graph implementations can be found in GraphX [44], GraphLab [27], Gunrock [41], and STINGER [2], [11]. Dynamic graph implementations can be found in STINGER and GraphIn [34].

b) Algorithmic Optimization: Numerous computational optimizations can be applied to triangle counting algorithms for **static graphs** to help reduce the overall execution time. For example, [15], presents a combinatorial optimization that reduces the number of necessary intersections - offering a better complexity bound. Parallelization is another approach for increasing performance. Ediger *et al.* [10] evaluate the performance of two algorithms on the Cray XMT (a massively multi-threaded system). One algorithm gives the exact number of triangles while the other algorithms give an approximation using Bloom Filters. Green *et al.* [19] show a scalable technique for load-balancing the triangle counting on shared-memory systems. Leist *et al.* [24] show the first GPU algorithm for triangle counting. In this approach each GPU thread is responsible for a different intersection. In contrast, Green *et al.* [20] offer a different parallelization scheme for the GPU that uses numerous GPU threads for each adjacency intersection based on the Merge-Path formulation [30], [18]. This improves the performance over [24] by an order of magnitude. Wang *et al.* [40] compare several different strategies for implementing triangle counting on the GPU. Shun & Tangwongsan [35] and Polak [31] explore vertex re-ordering to reduce the number of intersections. This re-ordering also creates an improved memory access pattern and reduces the number of cache misses. Lastly, the Doulion [38] framework shows an approximation technique for finding the number of triangles in the graph by sparsifying the network before running a static graph triangle counting algorithm.

c) Streaming and Dynamic Graph Triangle Counting: There are numerous streaming[1] graph algorithms for triangle counting. Most streaming graph algorithms limit the permitted number of memory accesses per update. This usually requires approximating the number of triangles in the graph rather than outputting the exact count. There are numerous applications where these approximations are still not accurate enough [4]. In [5] two variations of a streaming graph algorithm are given: one algorithm assumes nothing about the edge stream; while the other assumes the stream of edges is partially sorted (based on the source vertex). In addition to this, some algorithms may only support edge insertions, where edge deletions are not considered due to storage or computational requirements [23].

In [10] an exact (accurate) dynamic graph algorithm is shown. This algorithm makes use of STINGER and modifies the edges in the graph using the batch update followed by an execution of the analytic. A similar approach can be found in GraphIn [34]. It's worth noting that the dynamic triangle counting algorithm in GraphIn [34] does not explain how the triangles are updated given the fact that the graph updates are stored in separate data structure than the input graph. For batch updates consisting of one edge the update process is simple. However, for larger batches a situation can arise where numerous edges in a batch will create a triangle. Without the inclusion-exclusion formulation presented in our work which can find triangles within the batch update, it could very well be necessary to recompute the intersections for all the edges of an affected source vertex (as is done in

[1]We distinguish between the classic streaming graph model and dynamic graph models such that the streaming graph model refers to the fact that the graph evolves one edge at time with a limited number of memory operations per update and the dynamic graph model refers to the fact that the graph can simply change over time.

[10]). This increases the computational requirements of these algorithm. Our new algorithm can be hundreds of times faster than these.

A recent framework by Ediger and Fairbanks [12] shows how to formulate a dynamic graph problems using linear algebra operations. Specifically, they outline how to update triangle counts in a dynamic graph.

d) Graph Frameworks: The need for scalable streaming graph data structures has brought a plethora of data-structures and data-bases that enable graph updates. Each of these graph data structures supports different features and programming models. Some work only on shared-memory systems while others work with distributed memory systems. Some are memory based while others are disk based. Some support transactional memory operation and are fault tolerant. An extended discussion of these graph data structures and graph databases can be found in [28]. We briefly discuss some of the key findings of [28]: many (though not all) of the graph data bases, such as Pegasus [8], Giraph [21], GraphLab [27], and Boost [36], support updates. In practice the STINGER data structure [11] has the highest update rate. Further, STINGER also has the smallest memory footprint. In some cases the difference is orders of magnitude smaller than other frameworks systems. Furthermore, several different analytics, page-rank and connected components, were implemented for each of these systems. Once again, STINGER usually was the best performer.

e) STINGER: The STINGER data structure was first introduced in [2] as a high performance extensible data structure for dynamic graphs. STINGER uses blocked linked-lists to store the adjacency lists of vertices. This allows for vertices to grow and shrink in size. This means 1) STINGER is more flexible than Compressed Sparse Row (CSR) which does not support update operations, 2) has better locality than a linked-list which stores only one edge per element, and 3) has lower storage bounds than an adjacency matrix. However, the blocked linked-list representation makes efficient list intersection challenging and their respective sorting even more challenging. A distributed version of STINGER, called DISTINGER can be found in [13], though it seems that the source code for DISTINGER is not open-source and that no analytics have been implemented for this version.

III. Dynamic Graph Data Structures

Table I summarizes the symbols and notations used throughout the update process.

The cuSTINGER data structure is an extension of STINGER for the GPU [16]. While cuSTINGER supports much of the same functionality as STINGER, the internal data structure has been modified to better target the GPU's architecture. cuSTINGER uses dynamically growing arrays rather than blocked linked-lists for the adjacency arrays. This allows for improved locality and increased parallel scalability. cuSTINGER is a key component of our algorithm. The original version of cuSTINGER did not support sorted batch updates. This is a requirement of our algorithm. As such we add sorted batch updates to the data structure. In this section, we present the sorting update process that we added to cuSTINGER. This gives cuSTINGER an unique feature that is not in other dynamic graph data structures such as STINGER [2], DISTINGER [13], AIMS [43], and GraphIN [34], just to name a few.

Lastly, note, that as part of the new sorted graph update we construct a temporary graph called the update-graph. The update-graph is used for two different roles in our algorithm. The first role is straightforward - updating the graph. The second role is for updating the triangle count and is discussed in the next section.

Sorted Graph Updates

In many real world applications the need to modify the graph with multiple edge updates arises. These are known as *batches*[2]. Batches can increase scalability by dealing with multiple edges within a batch in concurrent manner. The adjacency list intersection approach for triangle counting requires that the adjacency lists be sorted. This also means that after a batch has been processed (edge insertions or deletions), all the adjacency arrays in the graph also have to be sorted to allow using adjacency list intersections. While the sorting operations increase the time complexity of the algorithm, most of these sorts are for relatively small arrays; therefore, they can be done in parallel (as they are independent of each other). We outline an efficient algorithm for this and present its complexity. In practice, our implementation can use different sorting algorithms, if these prove to be more efficient.

First of all, we assume that all the adjacency lists in the graph are sorted prior to the batch update. By default, if the graph is empty all the adjacency lists are trivially sorted. We separate the update process for edge insertions and edge deletions as these require slightly different assumptions from the dynamic graph data structure. Given a batch update, the **update-graph** $G' = (V, E')$ is created, where E' are the edges in the batch update and V are the vertices in the graph. The symbols used for this new graph can be found in Table I. The update graph itself is also a sparse network and as such can be represented by a CSR graph.

a) Update-Graph construction: the following outlines a simple and practical way to construct G'. First, given all edges in the update $e = \langle src, dst \rangle \in E'$, count the number of times src appears ($O(|E'|)$). Using this information, create a CSR representation of G' using a parallel prefix summation [22] (time complexity of $O(log|V|)$ and work complexity of $O(|V|)$). Note, these complexities are dependent on V rather than V' has the update-graph includes all the vertices in the original graph (even if they are empty in the update-graph).

The edge array contains all the destination points in the graph while the offset array points to a location in the edge array. Lastly, the adjacency lists in G' also need to be sorted. This can be done either by sorting each list an separate fashion or by using two radix sorts on the batch-update itself (which is also an edge list). The sorted batch-update is then

[2]A batch can also consist of a single edge

Algorithm 1 Sorting processing for batch insertion and deletions. The insertion algorithm uses an in-place (into $adj(u, G)$) merging algorithm where the elements are copied from the end to beginning. See Fig 1 for additional examples.

```
Require: Graph $G(V, E)$, $G'(V, E')$
procedure INSERTION
    parallel for $u \in V$ do
        $i \leftarrow d_u^G$                                         ▷ degree of $u$ in $G$
        $j \leftarrow d_u^{G'}$                                       ▷ degree of $u$ in $G'$
        while $i \geq 0 \wedge j \geq 0$ do
            diff $\leftarrow adj(u, G)[i] - adj(u, G')[j]$
            if diff $> 0$ then                               ▷ Copy from original.
                $adj(u, G)[i + j + 1] \leftarrow adj(u, G')[i]$
                $i \leftarrow i - 1$
            else                                          ▷ Copy from batch graph.
                $adj(u, G)[i + j + 1] \leftarrow adj(u, G')[j]$
                $j \leftarrow j - 1$
            end if
        end while
        while $j \geq 0$ do
            $adj(u, G)[i + j + 1] \leftarrow adj(u, G')[j]$
            $j \leftarrow j - 1$
        end while
end procedure
procedure DELETION
    parallel for $u \in V$ do
        $i \leftarrow d_u^G$
        $j \leftarrow d_u^{G'}$
        while $i \geq 0 \wedge j \geq 0$ do
            diff $\leftarrow adj(u, G)[i] - adj(u, G')[j]$
            if diff $= 0$ then $adj(u, G)[i] \leftarrow$ NULL
            end if
            if diff $\geq 0$ then $i \leftarrow i - 1$
            end if
            if diff $\leq 0$ then $j \leftarrow j - 1$
            end if
        end while
        $i \leftarrow 0$
        $j \leftarrow 0$
        while $i < d_u$ do                                    ▷ Stream compaction
            if $adj(u, G)[i] \neq$ NULL then
                $adj(u, G)[i] \leftarrow adj(u, G)[j]$; $j \leftarrow j + 1$
            end if
            $i \leftarrow i + 1$
        end while
        $d_u \leftarrow j$
end procedure
```

copied into the edge array of the CSR data structure. The time complexity of this radix sort is $O(|E'|)$. Pseudo code for the update-graph construction can be found in Alg. 1.

Lastly, the work complexity for update-graph construction is $O(|V|+|E'|)$. Usually, the $O(|V|)$ is the dominant factor in the work complexity. Fortunately, most phases in the update-graph construction process are scalable, allowing for a time complexity of $O(\frac{|V|+|E'|}{P})$, where P denotes the number of available processors. In most cases $|V| << P$.

b) Sorting a batch of edge insertions: given the update graph G' and the input graph G, the adjacency lists of both these graphs need to be **merged** (in a sorted fashion) into the final output graph, $\widehat{G}$. Given a batch consisting of multiple edges, the merging of each adjacency list can be done in parallel by allocating a single thread for each merge. For vertices in G' that do not have any edges, meaning no edges have been added to that vertex, the merging process is not required and is not executed. Thus, given a vertex $u \in V$, the time and storage complexity for merging the two adjacency lists is $O(d_u^G + d_u^{G'})$. The number of affected vertices, such that $d_u^{G'} > 0$ is dependent on the incoming batch. The total

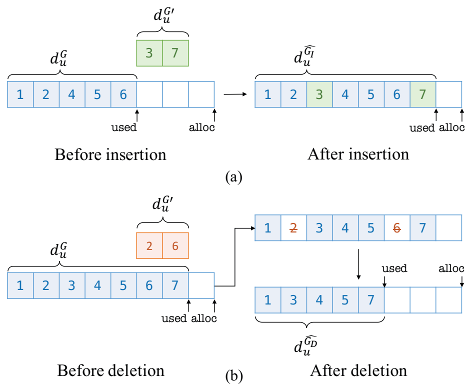


Fig. 1. Adjacency array before and after (a) insertion and (b) deletions. Inserted edges are denoted in green and deleted edges are denoted in red.

time complexity for this entire process is:

$$\sum_{(u,v)\in E'} O(d_u^G + d_u^{G'}) \qquad (1)$$

Implementation Detail - merging two sorted arrays into a single array can be done both in or out of place. Many merging algorithms will use a third array to store the results of the merged array. Using a third array requires allocating additional memory to store the output and is also costly in time. As the number of active vertices in batch can be high, this can be undesirable. Instead, an inplace merging can be done by starting at the tail of merge (Fig 1); recall, we know the final number of elements that need to be merged. Given this number we can ensure that the adjacency array of the vertex has enough elements to store the merged results. Recall, that in the cuSTINGER data structure additional edges are allocated per vertex to enable growth. For some vertices, the extra space will be enough to add merge the arrays. For other vertices, cuSTINGER has functionality to allocate additional storage to ensure that the merging can proceed. Note, the time complexity of the merging operation is linear to the size of the adjacency array.

c) Sorting a batch of edge deletions: unlike the edge insertion process, which had vertices that required additional storage due to an increase in the adjacency array sizes. the opposite can happen with edge deletions. A large number of edges might be removed such that the number of used edges versus allocated edges is so small that the memory is being under-utilized. For such cases, a new, yet smaller, adjacency block will be allocated for the update edge block. Similar to the edge insertion process, the edge deletion process is also done in place using a two phase process. The post deletion graph is denoted as $\widehat{G_D} = G \setminus G'$. E_v' denotes all the edges in the batch update. Phase 1: all edges $e \in E_v'$ edges are found and marked as being deleted, this requires $O(d_u^G + d_u^{G'})$ steps. Phase 2: the edge list is compacted by moving edges forward one at a time in a sequential fashion and can be done within $O(d_u^G)$ steps. For vertices that have allocated a new adjacency array, the compacted array is copied into the new memory and the older array can be reclamated (to improve

TABLE I

LIST OF SYMBOLS AND NOTATIONS.

Symbol	Description
G	Input graph.
G'	Update-graph constructed from batch update.
$\widehat{G_I}$	Post insertion-graph $\widehat{G_I} = G \cup G'$.
$\widehat{G_D}$	Post deletion-graph $\widehat{G_D} = G \setminus G'$.
V	Vertices in input graph.
E	Edges in input graph.
E'	Set of edges in batch update.
$adj(u, g)$	Adjacency of u in graph g.
d_u^g	Degree of u in graph g. $\lvert adj(u, g)\rvert$
T_n	Per vertex triangle array.
	Unique symbols for insertions
Δ_1^i	Triangles consisting of 1 new edge and 2 old edges.
Δ_2^i	Triangles consisting of 2 new edges and 1 old edge.
Δ_3^i	Triangles with 3 new edges.
	Unique symbols for Deletions
Δ_1^r	Open triangle with 1 deleted and 2 remaining edges.
Δ_2^r	Open triangle with 2 deleted and 1 remaining edge.
Δ_3^r	A former triangle with 3 edges deleted.

storage utilization). Based on the performance analysis in Section V, the fact that only a single thread is used for both phases does not seem to be a problem.

IV. Dynamic Triangle Counting

In this section, we present our new algorithm for triangle counting in dynamic graphs. The new algorithm uses the sorted batch update functionality discussed in the previous section. We use the following example to highlight key challenges of finding triangles in a batch. Consider the toy input graph in Fig 2 (a). There is a single triangle consisting of the triplet $\langle 1, 2, 7\rangle$. Note, all the edges in the input graph are shown with solid black lines. Fig 2 (b) depicts the graph after a batch of edges has been inserted into the graph. The new edges, (E'), are denoted with a dashed orange line. As a result of the edge insertion, new edges triangles have been created in the graph. Three new triangles, marked with blue dashed lines, are depicted in Fig 2 (c): $\langle 1, 4, 5\rangle$, $\langle 1, 2, 3\rangle$, and $\langle 1, 3, 4\rangle$. Note, the following differences between these triangles: $\langle 1, 4, 5\rangle$ has two edges from the original graph and one newly inserted edge, $\langle 1, 2, 3\rangle$ consists of two newly inserted edges and one original edges, and $\langle 1, 3, 4\rangle$ is made up of three newly inserted edges. The difference between these types of triangles plays a role in the counting process and is discussed below.

A. Insertions

We start off by highlighting the types of triangles formed by the insertion. We then show a counting correction mechanism that is based on an inclusion-exclusion formulation that correctly counts the number of triangles in the graph. The counting correction is necessary as not all intersection are computed in practice - this allows to significantly reduce the computational overhead.

First of all, we use the terms "new edge" and "old edge" to differentiate between newly inserted edges in the graph and an existing edges in the graph, respectively. There are **three types of triangles** that can be created following an

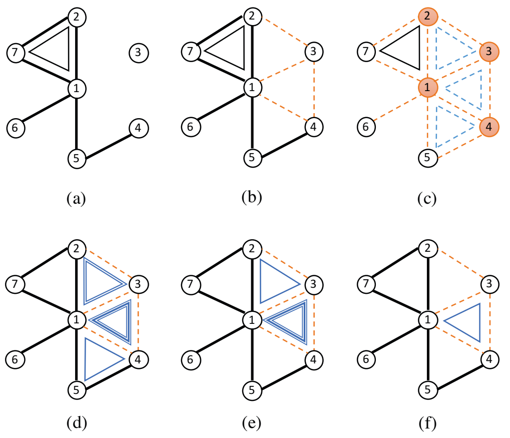


Fig. 2. Phases in new dynamic triangle counting algorithms for a small example graph. (a) Input graph. (b) Update graph after edge insertion - new edges denoted with dashed orange lines. (c) Affected vertices and edges are marked in dashed orange lines. Subplots (d), (e) and (f) show the triangles counted using the phases of our inclusion-exclusion formulation: S_1, S_2 and S_3 respectively.

edge insertion (each of these types of triangles is computed separately): 1) Δ_1^i - triangles with 1 new edge and 2 old edges, 2) Δ_2^i - triangles with 2 new edges and 1 old edge, and 3) Δ_3^i - triangles with 3 new edges.

The number of *new and unique* triangles that need to be counted, due to edge insertions, is:

$$NewTriangles = |\Delta_1^i| + |\Delta_2^i| + |\Delta_3^i| \qquad (2)$$

Recall, triangles are found by adjacency lists intersection. Each triangle can be represented by six unique ordered triplets. Give three vertices (u, v, w) of a triangle, this triangle can be represented by the following ordered triplets: $\langle u, v, w\rangle$,$\langle u, w, v\rangle$, $\langle v, w, u\rangle$, $\langle v, u, w\rangle$, $\langle w, u, v\rangle$, and $\langle w, v, u\rangle$. Many static graph algorithm can avoid counting the same triangle multiple times. This is more challenging for dynamic graphs. One challenge is that some of vertices in the formed triangle might be in the batch update and other vertices may not. Alg. 2 presents pseudo code for the dynamic triangle counting. In this algorithm triangle may be counted multiple times. In practice, we avoid counting some triangles multiple times and discuss this in Sec. IV-D.

Fig 2 (c) depicts the vertices and edges that have been affected and require recounting due to the graph update - this approach was taken in prior algorithm. In contrast, our algorithm only requires recounting triangles for the edges in the update batch, Fig 2 (b).

a) Δ_1^i: initially we find the triangles consisting of one new edge and two old edges. For each edge in the batch update, $e = \langle u, v\rangle \in E'$, intersect the two adjacency lists in the post insertion-graph , $\widehat{G_I}$ [3]:

$$s_{e,1} = adj(u, \widehat{G_I}) \cap adj(v, \widehat{G_I}) \qquad (3)$$

[3]In Sec. IV-D we show how to avoid doing both the intersection of u with v as well as the intersection of v with u.

Triangles found through this intersection includes **at least one** new edge, but might also consist of multiple new edges. We define S_1^i to be the total number of triangles found for all the intersections of E':

$$S_1^i = \sum_{e \in E'} |s_{e,1}| \tag{4}$$

Consider once again the three new triangles Fig 2 (c). While there are only three triangles, in practice we find six triangles: 1) the intersection of the adjacencies of $\langle 1,4 \rangle$ will find the triangles: $\langle 1,4,5 \rangle \in \Delta_1^i$ and $\langle 1,3,4 \rangle \Delta_3^i$, 2) the intersection of the adjacencies of $\langle 1,3 \rangle$ will find the triangles: $\langle 1,2,3 \rangle \in \Delta_2^i$ and $\langle 1,3,4 \rangle \in \Delta_3^i$. 3) the intersection of the adjacencies of $\langle 2,3 \rangle$ will find the triangle: $\langle 1,2,3 \rangle \in \Delta_2^i$. 4) the intersection of the adjacencies of $\langle 3,4 \rangle$ will find the triangle: $\langle 1,3,4 \rangle \in \Delta_3^i$.

Note, the number of times a triangle is found is based on the number of new edges it has. As such (4) can be rewritten as follows (counting duplications due to the undirected nature of the graph):

$$S_1^i = 2 \cdot |\Delta_1^i| + 4 \cdot |\Delta_2^i| + 6 \cdot |\Delta_3^i| \tag{5}$$

b) Δ_2^i: , next we find triangles with two new edges. These triangles must share a single end-point, otherwise, they will not form a triangle. We denote this end-point as p. We know that $\langle p,u \rangle, \langle p,v \rangle \in E'$. Unfortunately, when looking for these triangles, triangles made up of three new edges might be found as well. We show a counting correction scheme for this. Given an edge $e = \langle u,v \rangle \in E'$, we denote the triangles found in this intersection as:

$$s_{e,2} = adj(u, \widehat{G_I}) \cap adj(v, G') \tag{6}$$

To find these triangles, the following four intersections will be computed[4] : $\langle p,u \rangle, \langle u,p \rangle, \langle p,v \rangle, \langle v,p \rangle$. However, only the following intersections will find new triangles (causing a single triangle to be counted twice): $adj(u, \widehat{G_I}) \cap adj(p, G')$ and $adj(v, \widehat{G_I}) \cap adj(p, G')$. Ideally, the other two intersections can be avoided; however, in practice we don't know the identity of p and need to compute all intersections.

c) Δ_3^i: lastly we find triangles made up of three new edges. An example of this can be found in Fig 2 (e) and (f) where $\langle 1,2,3 \rangle$ is counted twice and $\langle 1,3,4 \rangle$ is counted six times (due to the edge ordering). This over-counting is true for all triangles made up of three new edges. Given the above we can formulate the total number of triangles counted here as:

$$S_2^i = \sum_{e \in E'} |s_{e,2}| = 2 \cdot |\Delta_2^i| + 6 \cdot |\Delta_3^i| \tag{7}$$

These triangles are found by intersecting all the edges in the update graph G'. This gives:

$$S_3^i = 6 \cdot |\Delta_3^i| \tag{8}$$

[4]The ordering of these edges is important as the adjacency lists are taken from two different graphs G' (update-graph) and $\widehat{G_I}$ (post insertion graph) - one adjacency is taken from G' and the other from $\widehat{G_I}$.

Algorithm 2 Algorithm to compute and update triangle counts - updates both the global and local triangle count. T_n denotes the local triangle count. Each intersection can be done in parallel [20].

```
Require: $\widehat{G_I}, \widehat{G_D}, G', E', T_n$
 1: function COUNT($g_1, g_2, E', T_n, m$)
 2:     $totalcount \leftarrow 0$
                                        ▷ Intersect adjacency lists for all edges $E'$
 3:     parallel for $\langle u,v \rangle \in E'$ do
 4:         count $\leftarrow 0$
 5:         $i \leftarrow 0; j \leftarrow 0$
                                ▷ Individual intersections can also be done in parallel
 6:         while $i < d_u \wedge j < d_v$ do
 7:             diff $\leftarrow adj(u, g_1)[i] - adj(v, g_2)[j]$
 8:             if diff $= 0$ then
 9:                 $count \leftarrow count + 1$
10:                 $w \leftarrow adj(u, g_1)[i]$
11:                 $T_w \leftarrow T_w + m$
12:             end if
13:             if diff $\leq 0$ then $i \leftarrow i + 1$
14:             end if
15:             if diff $\geq 0$ then $j \leftarrow j + 1$
16:             end if
17:         end while
18:         $T_u \leftarrow T_u + m * count$
19:         $T_v \leftarrow T_v + m * count$
20:         $totalcount \leftarrow totalcount + 3 \cdot count$
21:     end parallel forreturn $totalcount$
22: end function
23: procedure INSERTIONCOUNT
24:     $S_1^i \leftarrow$ COUNT($\widehat{G_I}, \widehat{G_I}, E', T_n, 1/2$)        ▷ Count $S_1^i$
25:     $S_2^i \leftarrow$ COUNT($\widehat{G_I}, G', E', T_n, -1/2$)       ▷ Count $S_2^i$
26:     $S_3^i \leftarrow$ COUNT($G', G', E', T_n, 1/6$)           ▷ Count $S_3^i$
27:     return $\frac{1}{2}(S_1^i - S_2^i + \frac{S_3^i}{3})$
28: end procedure
29: procedure DELETIONCOUNT
30:     $S_1^d \leftarrow$ COUNT($\widehat{G_D}, \widehat{G_D}, E', T_n, -1/2$)      ▷ Count $S_1^d$
31:     $S_2^d \leftarrow$ COUNT($\widehat{G_D}, G', E', T_n, -1/2$)       ▷ Count $S_2^d$
32:     $S_3^d \leftarrow$ COUNT($G', G', E', T_n, -1/2$)          ▷ Count $S_3^d$
33:     return $\frac{1}{2}(S_1^i + S_2^i + S_3^i)$
34: end procedure
```

Note, triangles from Δ_1^i and Δ_2^i cannot and will not be found in this intersection as these require information from the post-update graph. This is shown in Fig 2 (f) where only triangle $\langle 1,3,4 \rangle$ is found.

d) All Triangles: Finally, using an inclusion-exclusion formula that takes into account how the different stages of the algorithm over count the number of triangles we show that the final number of new triangles is:

$$|\Delta_1^i| + |\Delta_2^i| + |\Delta_3^i| = \frac{1}{2}\left(S_1^i - S_2^i + \frac{S_3^i}{3}\right) \tag{9}$$

The pseudo-code for our new algorithm can be found in Alg. 2 where both the insertion and deletion processes are shown.

B. Deletions

Deletions are conceptually similar to insertions, however the final triangle count does not require the inclusion-exclusion formulation as there is no over counting. Counting triangles after edge deletions can be computed using a simple formula, which is outlined below. We use the terms "remaining edges" and "deleted edges" for the edges left in the modified graph and the edges deleted in the batch update, respectively. This leads to the following classifications of **deleted triangles**: 1)Δ_1^d - Open triangle with 1 deleted edge and 2 remaining edges, 2)Δ_2^d - Open triangle with 2 deleted edges and 1 remaining edge, and 3)Δ_3^d - A former triangle with 3 edges deleted.

For deletions, rather than looking for triangles after the update, we need to detect triangles that existed before their removals. This means that if a triangle with 2 deleted edges is removed, it won't be detected in the intersections done in S_1^d. Also, if a triangle with 3 deleted edges is removed, then it won't be found in S_1^d or S_2^d. This means that the formula to compute deleted triangle is considerably simpler than the formula for inserted triangles.

By calculating S_1^d, S_2^d and S_3^d in the same fashion and using a slightly different formulation, we find that no over-counting occurs. Thus,

$$S_1^d = 2 \cdot |\Delta_1^d| \tag{10}$$

$$S_2^d = 2 \cdot |\Delta_2^d| \tag{11}$$

$$S_3^d = 2 \cdot |\Delta_3^d| \tag{12}$$

And from $(10) + (11) + (12)$ we get

$$|\Delta_1^d| + |\Delta_2^d| + |\Delta_3^d| = \frac{1}{2}(S_1^d + S_2^d + S_3^d) \tag{13}$$

which is our final required deleted triangle count.

The pseudo-code for deletions can be found in Alg. 2. Note, that the key difference between insertions and deletions are the *multipliers* used. Negative constants state that triangles are being removed from the count and positive constants imply that triangles are being added.

C. Complexity Analysis

Each of the intersections in the triangle counting algorithms requires $O(d_u^{g1} + d_v^{g2})$ time where d_x^g is degree of vertex x in graph g. There is a total of E' edges that are processed and for each there are three different intersections that are computed as part of the inclusion-exclusion formulation: $(\widehat{G_I}, \widehat{G_I})$, $(\widehat{G_I}, G')$ and (G', G'). The largest of these intersections will be $(\widehat{G_I}, \widehat{G_I})$, as $\forall u \in \widehat{G_I}, adj(u, G') \subseteq adj(u, \widehat{G_I})$. The time complexity, dominated by the largest intersection, is therefore:

$$O(|E'| \cdot (d_{max}^{\widehat{G_I}} + d_{max}^{\widehat{G_I}})) = O(|E'| \cdot d_{max}^{\widehat{G_I}})$$

Where $d_{max}^{\widehat{G_I}}$ is the vertex with the largest degree within the graph. The deletions time complexity is similar. Note, our algorithm does not require additional space as we reuse the update-graph, G', during the graph update.

D. Additional Algorithmic Optimizations

Lastly, we discuss a few algorithmic optimizations that make the new dynamic triangle counting algorithm more efficient. First of all, the vertex reordering scheme used by Shun & Tangwongsan [35] can also be adopted to the dynamic graph algorithm. Our objective is to find each triangle only once in a fixed order. Therefore, corresponding to a triangle consisting of u, v, w, only the ordered triplet $\langle w, v, u\rangle$ (where $w > v > u$) should be found in an intersection and not the remaining combinations.

To do this, we create two arrays of size $|V|$ each. Each vertex, $u \in V$ stores a *reduced adjacency vertex degree*: d_u^r. This degree, d_u^r, represents the number of neighbors u has with have a vertex ID less than u. Since the adjacency arrays are sorted, splitting the adjacency array from 0 to d_u^r gives us a reduced adjacency array $adj_r(u, g)$ without modifying the original. For a triplet $\langle w, v, u\rangle$ such that $w > v > u$, u and v are found in $adj_r(w, g)$, u is in $adj_r(v, g)$ but w is not and, neither v nor w are in $adj_r(u, g)$. The two arrays contain these d_u^r values for graphs $\widehat{G_I}$ (or $\widehat{G_D}$ for deletions) and G'.

This optimization is applied slightly differently in each of the three phase. For counting $S_1^{i/d}$ we use the reduced adjacency lists for G'. However, we do not use these reduced degree lists for $\widehat{G_I}$ (or $\widehat{G_D}$). In practice, this means that size of E' is reduced by half, as edges where the source ID is smaller than the destination ID are ignored. For counting $S_2^{i/d}$ we use the reduced adjacency lists for $\widehat{G_I}$ (or $\widehat{G_D}$) but not for G'. Lastly, counting $S_3^{i/d}$ does not involve the use of $\widehat{G_I}$ (or $\widehat{G_D}$). By using the above optimizations, we reduce the previous formulations for S^i to and the new and final formulation:

$$|\Delta_1^i| + |\Delta_2^i| + |\Delta_3^i| = (S_1^i - S_2^i + S_3^i) \tag{14}$$

For deletions, each of S^d is reduced by a factor of 2 (in comparison with Eq. (13):

$$|\Delta_1^d| + |\Delta_2^d| + |\Delta_3^d| = (S_1^d + S_2^d + S_3^d) \tag{15}$$

V. Performance Analysis

A. Experiment Setup

Our experiments are conducted on an NVIDIA K40 GPU which has 15 SMs and 192 SPs per SM, for a total of 2880 SPs and has 12GB of GDDR5 memory. The CPU is a quad-core Intel i7-4770K, which is a Haswell based processor, running at 3.5 GHz with 8MB L3 cache. This system has 32GB of DDR3-1600 memory. To test the algorithm we use real world graphs and networks taken from the 10th DIMACS Graph Implementation Challenge and [25]. Details of these graphs can be found in Table II. In the case of directed graphs, all edges have been duplicated to ensure that the graph is undirected.

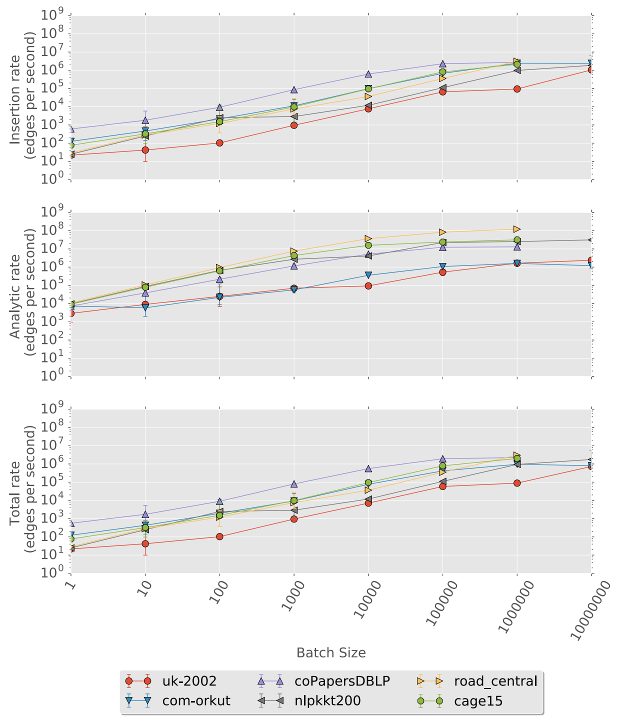
B. Batch Creation

To test the throughput rate of our analytic, we check its ability to handle different batch update sizes. For each graph, the largest batch size is no larger than the original graph. All batch sizes are in powers of ten. The edges in the batches are taken from the original input graph - as such real edges are inserted into the graph. For edge deletions, prior to loading the graph to the GPU a subset of existing edges are sampled into a batch. Note that the batch sizes refer to undirected edges. In practice, both directions of the edge update appear

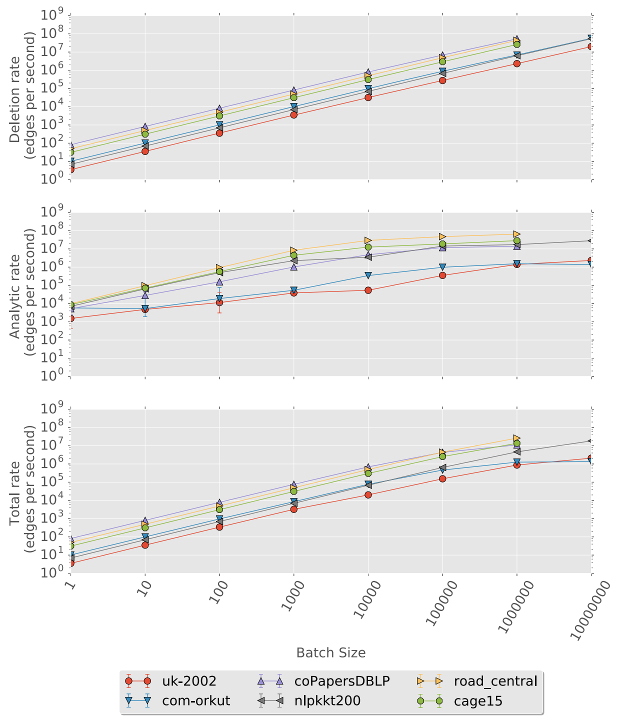
TABLE II

NETWORKS USED IN OUR EXPERIMENTS. $|E|$ REFERS TO DIRECTED EDGES. NETWORKS ARE SORTED BASED ON THE NUMBER OF VERTICES.

Name	Network Type	$\|V\|$	$\|E\|$	Ref.	Static (sec.)	Insertion (sec)			Deletion (sec)		
						100k	1M	10M	100k	1M	10M
coPapersDBLP	Social	540k	30M	[3]	1.032	0.053	0.452	-	0.025	0.098	-
in-2004	Webcrawl	1.38M	27M	[3]	18.176	0.213	2.208	-	0.117	1.805	-
com-orkut	Social	3M	234M	[25]	90.164	0.242	1.107	10.440	0.218	0.807	8.451
com-LiveJournal	Social	4M	69M	[25]	8.975	0.168	0.765	-	0.067	0.191	-
cage15	Matrix	5.15M	94M	[3]	1.638	0.132	0.651	-	0.043	0.091	-
nlpkkt160	Matrix	8.3M	221M	[3]	1.778	0.192	0.329	7.537	0.089	0.156	0.332
road_central	Road	14M	33M	[3]	1.348	0.288	0.348	-	0.029	0.057	-
nlpkkt200	Matrix	16.2M	432M	[3]	3.460	0.910	1.081	2.016	0.164	0.238	0.732
uk-2002	Webcrawl	18.52M	523M	[3]	522.586	1.653	10.875	12.416	0.629	1.170	5.981
road_usa	Road	24M	58M	[3]	2.188	0.480	0.550	-	0.046	0.074	-



(a) Insertions



(b) Deletions

Fig. 3. This figure depicts the update rate, number of edges per second, at which we can update the graph and analytic. This shown for both edge insertions (a) and deletions (b). For each operation, the update rate is plotted for different phases of the algorithm: (*top*) graph modification, (*middle*) dynamic graph triangle counting using new algorithms, and (*bottom*) the update rate for entire process from start to end.

in the batch. The batch update rates reported in this paper refer to undirected edges[5].

Each graph is tested with five different batches for each batch size. For each benchmark the original input graph is used. Note, that the batches are not stored in the GPUs main memory, rather they are stored in the CPU memory. The batches are transfered to the GPU as part of the update process. The time spent transferring the batch is included in our timing. Its worth noting that all the batches are created in advance and as such they do not reside in the CPU's cache.

[5]As such, the effective update rate supported by the data structure is essentially twice the reported rate as both directions are added.

C. Update Rates

Recall that updating the dynamic triangle counts consists of three stages: 1) constructing the update graph G' from the batch update edge list (including sorting the batch update), 2) modifying the cuSTINGER graph, and 3) updating the triangle count. Creating the graph G' actually serves two purposes: 1) preparing the batch update to be inserted/deleted from the graph in a sorted fashion and 2) updating the triangle count.

Fig. 3 depicts the update rates for both edge insertions (a) and deletions (b). The abscissa represents the size of the batch update size. The following update rates are shown in Fig. 3: (top) just modifying the graph, (middle) updating the triangle counts using the dynamic algorithm, and (bottom)

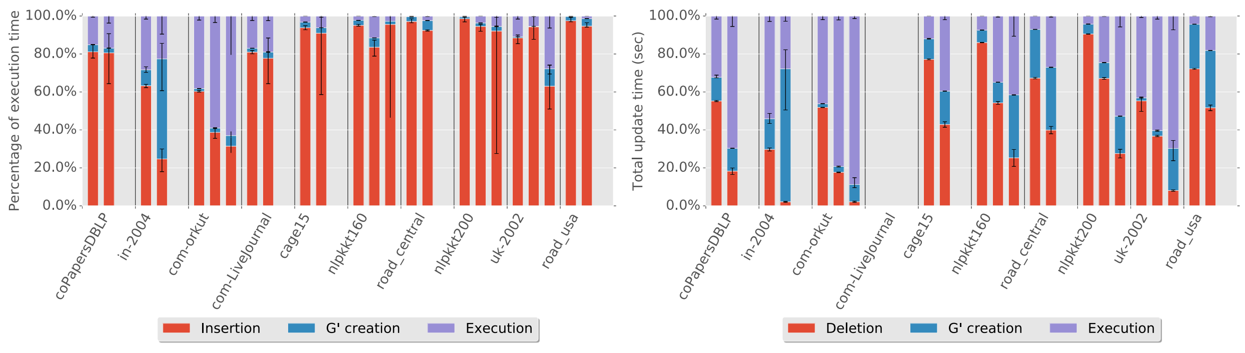


Fig. 4. This figure depicts the execution breakdown (in percentage) for the three stages in the execution: 1) creating the update graph G' from the batch update, 2) inserting (or deleting) the batches into the graph (modification of cuSTINGER), and 3) running the dynamic graph triangle counting.

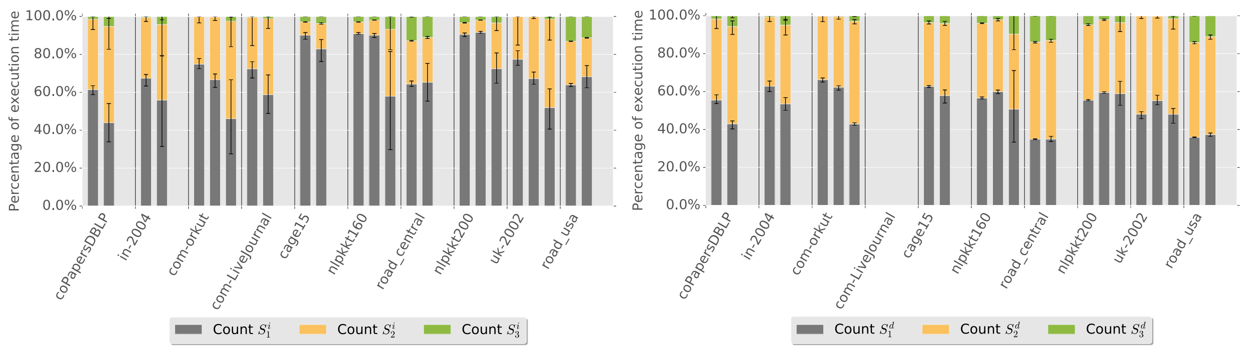


Fig. 5. This figure depicts the execution breakdown (in percentage) of only the dynamic triangle counting analytic using the inclusion-exclusion formulation. For both the insertion (a) and deletions (b) there are three phases. The execution time of the triangle counting accounts for the purple bars in Fig. 4.

the overall rate combining both updating the graph and the analytic.

For small batch sizes, especially up to to 1000 edges, there are numerous *large* overheads that limit the update rate; we highlight two of these. 1) Building and sorting G' - requires a parallel prefix summation operation and is dependent on the number of vertices in the graph. While this operation can be highly optimized for the GPU, it still can take a several milliseconds, which is relatively high in comparison to the update process. 2) Memory re-allocation - this problem was reported in cuSTINGER [16] and is also problematic here[6]. These overheads explain the performance restrictions of the sorting and update process (Fig. 3 top left and top right). For batch sizes greater than 10k edges, this overhead becomes negligible. Insertions can be processed at a rate well over one million per second and the deletions can be processed at well over 30 *million* updates per second. Table II shows the update times for both insertions and deletions for several different batch sizes. Recall, the key difference between insertions and deletions is the additional memory re-allocations phase in the insertion process.

The middle subplots in Fig. 3 depict the update rate for counting triangles as a function of the batch size. For small batch sizes the GPU is under-utilized. As the batch size increases there is more work to keep the GPU saturated - this is true for both insertions and deletions. The analytic can be updated at rates of over ten million updates per second. Further performance factors include the graph structure and the number of vertices affected by the updates. The graphs with the slowest updates, *uk-2002* and *com-orkut* are denser than the other graphs leading to slower intersections.

The bottom subplot Fig. 3 depicts the overall update rate. Note, for insertions the top and bottom most plots are similar due to the overhead of the memory allocation phase. For deletions, this is not the case.

D. Execution Time Analysis

Fig. 4 and Fig. 5 depict the percentage of time spent in the various phases of the algorithm for three different batch sizes[7]: 0.1M, 1M, and 10M. Fig. 4 depicts the three main phases of the algorithm: graph G' creation (blue), updating

[6]The cuSTINGER data structure has been replaced by the Hornet data structure [1]. Hornet no longer has the aforementioned problems and is about 4X-10X faster for mid-size and large batches. At time of publication this algorithm has not been ported into Hornet.

[7]A missing bar implies that the algorithm was not tested with a specific configuration. For example numerous 10M bars are missing as there were not enough edges in the original graph to remove.

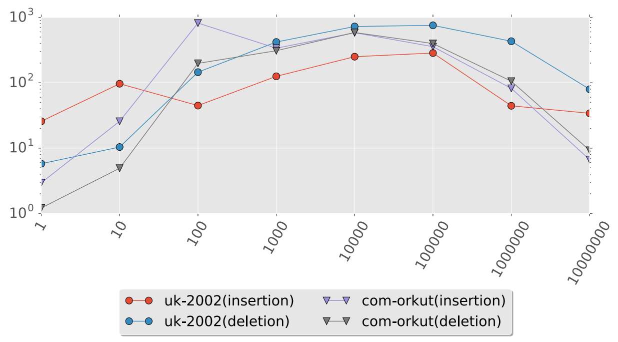


Fig. 6. Speedup of our new algorithm over the previous algorithm which required recounting all triangles for vertices affected by an update. In all cases, our new algorithm is faster than the past case as would be expected given the reduced time complexity of our approach.

the graph data structure (red), and analytic execution time (purple). For edge insertions, a minority of the execution is spent in the analytic update. For deletions, the percentage of time spent in updating the triangle counts is significantly larger. Yet, the actual execution times of the analytic update is similar. Once again, this has to do with the various overheads, including that of cuSTINGER.

Fig. 5 shows the execution time breakdown of only the analytic for the three different phases, discussed in Section IV. Finding new triangles that are entirely within the update graph (green bars) - uses only the adjacency lists within G'. These adjacency lists are a smaller subset than the ones found in the updated graph $\widehat{G}$. This explains why the other two types of triangles account for a larger amount of the analytic execution time.

E. Speedup Over Previous Approaches

Since our exact and dynamic algorithm is also the first for the GPU, we do not have any other algorithms to benchmark. While it is be possible to compare our GPU implementation with a CPU implementation, this would require us to compare: implementations, dynamic graph data structures, and architectures. To avoid this we implement the approach taken in [10] on the GPU. Fig. 6 depicts the speedup of our new algorithm over prior approaches. Given that our algorithm requires far less work than past approaches, it is perhaps not surprising that our new algorithm is faster in all cases.

For small batch sizes, the amount of work is small. This means that the GPU is under-utilized leading to relative lower speedups of our algorithm. For batch sizes of 100 edges and up to batches of 1M edges, the new algorithm is almost always $100X$ faster than the past approach and as much as $819X$ faster. For batch sizes of 10M edges, it seems that execution time is being dominated by the larger list intersection executed by both algorithms. Nonetheless, the new algorithm is still anywhere from $10X - 80X$ faster than the prior approach.

VI. Conclusions

In this paper we provide a new algorithm for finding triangles in dynamic graphs using an inclusion-exclusion formulation. Our algorithm gives the exact number of triangles in the graph (global triangle count) and also updates the number of triangles each vertex belongs to (local triangle count). Our algorithm is computationally more efficient than prior algorithms which required "re-finding" all triangles for vertices affected by an edge update. For large batches, this typically meant going over all vertices and edges to find the accurate number of triangles. In contrast, we only need to access the affected edges. This results in significant speedups - for most cases over $100X$ faster and as much as $819X$ faster.

While we implemented our algorithm for the GPU, it is architecture independent and can be implemented on additional parallel programming platforms, with the constraint that a dynamic graph data structure be used. Our new algorithm supports tens of millions of updates per second on a single GPU. When the overhead of updating the graph is also considered, we are able to update both the graph and analytic at several millions of updates per second for insertions and tens of millions of updates per second for deletions.

While the cuSTINGER data structure was modified as part of this work (namely, we added sorted updates), we continued to use its internal memory allocation process. Given the overhead it added to our algorithm, we will investigate more efficients ways to reduce the time spent in the graph update phase. We are able to support graph update rates that are sufficient for many real world applications.

Acknowledgments

Funding was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This work was also partially sponsored by NSF Grant ACI-1339745 (XScala).

References

[1] "Hornet Data Structure repository," https://github.com/hornet-gt/hornet, 2017.

[2] D. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Georgia Institute of Technology, Tech. Rep., 2009.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop*, ser. Contemporary Mathematics, no. 588, 2013.

[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs," in *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2008, pp. 16–24.

[5] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting Triangles in Data Streams," in *25th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, 2006, pp. 253–262.

[6] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the 17th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2011, pp. 672–680.

[7] J. Cohen, "Trusses: Cohesive Subgraphs for Social Network Analysis," *National Security Agency Technical Report*, p. 16, 2008.

[8] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[9] D. Ediger and D. Bader, "Investigating Graph Algorithms in the BSP Model on the Cray XMT," May 2012.

[10] D. Ediger., K. Jiang, J. Riedy, and D. Bader, "Massive Streaming Data Analytics: A Case Study with Clustering Coefficients," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[11] D. Ediger, R. McColl, J. Riedy, and D. Bader, "STINGER: High Performance Data Structure for Streaming Graphs," in *IEEE High Performance Embedded Computing Workshop (HPEC 2012)*, Waltham, MA, 2012, pp. 1–5.

[12] D. Ediger and J. Fairbanks, "Deriving Streaming Graph Algorithms for Static Definitions," in *IEEE Int'l Conf. Parallel and Distributed Processing Symposium Workshop (IPDPSW),*, 2017.

[13] G. Feng, X. Meng, and K. Ammar, "DISTINGER: A distributed graph data structure for massive dynamic graph processing," in *IEEE Int'l Conf. on Big Data (Big Data)*, 2015, pp. 1814–1822.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, vol. 12, 2012.

[15] O. Green and D. Bader, "Faster Clustering Coefficients Using Vertex Covers," in *5th ASE/IEEE International Conference on Social Computing*, ser. SocialCom, 2013.

[16] O. Green and D. Bader, "cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2016.

[17] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, "Quickly Finding a Truss in a Haystack," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.

[18] O. Green, R. McColl, and D. Bader., "GPU Merge Path: A GPU Merging Algorithm," in *26th ACM International Conference on Supercomputing*, 2012, pp. 331–340.

[19] O. Green, L. Munguia, and D. Bader, "Load Balanced Clustering Coefficients," in *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, Feb. 2014.

[20] O. Green, P. Yalamanchili, and L. Munguía, "Fast Triangle Counting on the GPU," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[21] M. Han and K. Daudjee, "Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[22] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[23] K. Kutzkov and R. Pagh, "Triangle counting in dynamic graph streams," in *Scandinavian Workshop on Algorithm Theory*. Springer, 2014, pp. 306–318.

[24] A. Leist, K. Hawick, D. Playne, and N. S. Albany, "GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs," in *Int'l Conf. on Scientific Computing (CSC'11)*, 2011.

[25] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[26] J. Leskovec, K. J. Lang, and M. Mahoney, "Empirical comparison of algorithms for network community detection," in *Proceedings of the 19th Int'l Conf. on World Wide Web*. ACM, 2010, pp. 631–640.

[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proc. of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727.

[28] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. Bader, "A Performance Evaluation of Open Source Graph Databases," in *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, 2014, pp. 11–18.

[29] M. E. Newman, D. J. Watts, and S. H. Strogatz, "Random Graph Models of Social Networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl 1, pp. 2566–2572, 2002.

[30] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge path - parallel merging made simple," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, may 2012.

[31] A. Polak, "Counting triangles in large graphs on GPU," *arXiv preprint arXiv:1503.00576*, 2015.

[32] A. Prat-Pérez, D. Dominguez-Sal, J. M. Brunat, and J.-L. Larriba-Pey, "Shaping Communities out of Triangles," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM '12, 2012, pp. 1677–1681.

[33] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static Graph Challenge: Subgraph Isomorphism," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2017.

[34] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, "Graphin: An Online High Performance Incremental Graph Processing Framework," in *European Conference on Parallel Processing*. Springer, 2016, pp. 319–333.

[35] J. Shun and K. Tangwongsan, "Multicore Triangle Computations Without Tuning," in *IEEE Int'l Conf. on Data Engineering (ICDE)*, 2015.

[36] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[37] T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," in *Experimental & Efficient Algorithms*. Springer, 2005, pp. 606–609.

[38] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 837–846.

[39] J. Wang and J. Cheng, "Truss Decomposition in Massive Networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[40] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the gpu," in *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 2016, pp. 1–8.

[41] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: GPU Graph Analytics," *arXiv preprint arXiv:1701.01170*, 2017.

[42] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[43] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, Independent Management of Dynamic Graphs on GPUs," in *International Supercomputing Conference*. Springer, 2017, pp. 97–119.

[44] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[45] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 745–754.