Quickly Finding a Truss in a Haystack

Oded Green[1], James Fox[1], Euna Kim[1], Federico Busato[2], Nicola Bombieri[2], Kartik Lakhotia[3], Shijie Zhou[3], Shreyas Singapura[3], Hanqing Zeng[3], Rajgopal Kannan[3], Viktor Prasanna[3], and David Bader[1]

[1]Computational Science and Engineering, Georgia Institute of Technology
[2]Department of Computer Science, University of Verona
[3]Department of Electrical Engineering, University of Southern California

***Abstract*— The k-truss of a graph is a subgraph such that each edge is tightly connected to the remaining elements in the k-truss. The k-truss of a graph can also represent an important community in the graph. Finding the k-truss of a graph can be done in a polynomial amount of time, in contrast finding other subgraphs such as cliques. While there are numerous formulations and algorithms for finding the maximal k-truss of a graph, many of these tend to be computationally expensive and do not scale well. Many algorithms are iterative and use static graph triangle counting in each iteration of the graph. In this work we present a novel algorithm for finding both the k-truss of the graph (for a given k), as well as the maximal k-truss using a dynamic graph formulation. Our algorithm has two main benefits. 1) Unlike many algorithms that rerun the static graph triangle counting after the removal of non-conforming edges, we use a new dynamic graph formulation that only requires updating the edges affected by the removal. As our updates are local, we only do a fraction of the work compared to the other algorithms. 2) Our algorithm is extremely scalable and is able to concurrently detect deleted triangles in contrast to past sequential approaches. While our algorithm is architecture independent, we show a CUDA based implementation for NVIDIA GPUs. In numerous instances, our new algorithm is anywhere from 100X-10000X faster than the Graph Challenge benchmark. Furthermore, our algorithm shows significant speedups, in some cases over 70X, over a recently developed sequential and highly optimized algorithm.**

I. Introduction

The subgraph isomorphism problem tries to answer the following question, given two graphs H and G (where H is the smaller of these graphs): is there a $1-1$ mapping of vertices in H to vertices in G such that each edge in H is also in G?. For example, H might be a clique of size k, in which case the question is, "Is there a clique of size k in G?". The answer to this question is an $NP-Complete$ problem. Yet, there are simplifying assumptions on the structure of H that can help make the problem computationally feasible and tractable - so long as a simpler subgraph H is defined. The need for subgraph isomorphism presents itself in numerous applications, including community detection and social network analysis, were there is a need to find a subgraph with a given set of properties. Another way of looking at the subgraph isomorphism problem is pattern finding, where H represents the pattern. Therefore, it is not overly surprising that in many cases the pattern will be relatively small in comparison with the initial input - this is almost like looking for a *"needle in a haystack"*. For example, a triangle in a graph can be thought of as a pattern and enumerating all the triangles in the graph meets the requirements of the subgraph isomorphism problem. While maximal clique finding is computationally intractable, finding and enumerating all the triangles in a graph can be done in polynomial time. Thus, the problem of subgraph isomorphism can be tractable for specific patterns and for a known H.

The HPEC Graph Challenge [31] seeks to find a high performance solution for a specific subgraph isomorphism problem where the structure of H is a k-truss within G. A k-truss is subgraph where each edge is part of at least $k-2$ triangles. The maximal k-truss in a graph, denoted by $k=k_{max}$ is the largest k-truss in the graph where the set of satisfying edges is not empty. The exact k or structure of the final maximal k-truss is not known apriori and is dependent on the graph. Finding the maximal k-truss can be done in polynomial time [4], [33], [14], [8].

Contribution

In this paper we show a new algorithm for finding a k-truss subgraph as well as the maximal k-truss in a graph. While the algorithm focuses on finding trusses in a static graph, we introduce concepts and principles used in dynamic graph algorithms. First of all we use a dynamic graph data structure designed for sparse networks where the edges can be removed efficiently from the graph without needing to rebuild the graph after each change [10], [3]. Second, we show a highly efficient and scalable dynamic graph triangle counting algorithm for updating the number of triangles in the graphs without needing to recompute all the triangles in every iteration. While our algorithm is architecture independent, we show an implementation of it for the NVIDIA GPU.

Altogether, our new algorithm is significantly faster than the HPEC Graph Challenge [31] benchmarks. While our algorithm always completed in a reasonable amount of time, there are numerous instances in which one or more of the benchmarks did not complete. In most cases we saw that our new algorithm is easily $100X$ faster than the best performing Graph Challenge benchmark and upto $10000X$ faster than the other remaining benchmarks. Our new algorithm also outperformed the recent work of Wang & Cheng [33]. While the algorithm in [33] is highly efficient, it is also inherently sequential as it is unable to update the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

triangle count concurrently when removing multiple edges. Our algorithm is concurrent and extremely scalable.

II. Related Work

A. Problem Definition

Given a graph, $G = (V, E)$, the vertices are denoted as V and the edges are denoted by E. The maximal k-truss of the graph, $H = (\hat{V}, \hat{E})$ meets the following criteria: 1) $\hat{V} \subseteq V$, 2) $\hat{E} \subseteq E$, 3) $H \subseteq G$, and 4) $\forall e \in \hat{E}, tri(e) >= k - 2$. For the maximal k-truss problem, k needs to be the maximal value before $E = \emptyset$ and $V = \emptyset$. In many papers, the term *support of edge* can be used to replace the term $tri(e)$. We use both throughout this paper.

B. K-Truss

The k-truss was first introduced by Cohen [4] as a relaxation of a clique (due to the reduced complexity of the truss) while still ensuring that if two vertices are in a given truss it is quite likely their common neighbors will be in the truss. Several different approaches for finding trusses and the maximal k-truss are also discussed in [4]. Yet, these share common algorithmic properties: 1) the algorithm is executed in an iterative fashion and 2) in each iteration of the algorithm, a subset of edges is removed from the graph. Edges are removed from the graph if their support (i.e. number of triangles they participate in) is not large enough based on the given iteration.

In [5], Cohen discusses the benefits of implementing graph algorithms in the Map-Reduce framework which enables the analysis of large networks. It is worth noting that the work in [5] preceded the creation of the Pregel [25] framework. The introduction of Pregel improved expressibility and simplicity of implementing graph algorithms in a Map-Reduce framework, though performance of these algorithms did not improve as much. While the work in [5] showed the ability to scale to larger networks, the work by Wang and Cheng [33] showed that an optimized algorithm designed for a single shared-memory system can easily outperform the Map-Reduce implementation.

Wang and Cheng [33] show several different optimizations for finding the maximal k-truss , though these algorithms are sequential[1]. Further, Wang and Cheng [33] discuss several iterative approaches for finding trusses, including a bottom-up approach and a top-down approach. The bottom-up approach is closer to the approach taken by [4], whereas the top-down approach works in the reverse direction (starting from the edge with the largest number of triangles and working its way down). The top-down approach is ideal for cases when there is a need to find either the maximal k-truss or for k's close to the maximal k-truss . In practice, the top-down approach has a performance penalty making it more expensive than the bottom-down approach in many instances.

[1]One of the main challenges associated with a parallel implementation of their algorithms is the need for correct triangle counting in parallel and dynamic environment. A solution to the problem was recently given in [24] and is discussed in additional detail in Section II-E.

Sariyuce *et al.* [28] present a new approach for decomposing a graph into a forest of nuclei. These nuclei are dense sub-graphs with clique-like properties. One nuclei subgraph used in the decomposition is the maximal k-truss.

Kabir and Madduri [17], [18] show a parallel algorithm for k-truss decomposition for multi-core systems. Their algorithm also uses an efficient triangle counting phase that avoids unnecessary graph intersections as well as two edge queues for storing the list of active edges in the graph. The active edges are used for updating the support of edges and filtering edges not matching the necessary support.

Gadepally *et al.* present the Graphulo [8] framework which enables implementing graph algorithm using linear algebra operators over the Apache Accumulo NoSQL database. The formulation for finding a k-truss in Graphulo is similar to the formulations of the baseline benchmarks of the HPEC Graph Challenge, which are implemented in Matlab, Julia, and Python using highly optimized libraries. The linear algebra based algorithm in [8] presents one iteration of [4], [33] for a specific k, though this can be extended to find the maximal k-truss . Huang *et al.* [14] show how to maintain the various trusses of a graph in a dynamic environment.

C. Triangle Counting

Triangle counting is a building block for numerous applications. Therefore, it is not overly surprising that numerous algorithms and optimizations have been designed to efficiently compute it. Some libraries and implementations have focused on good system utilization with good load-balancing [12], [34], others have focused on data scalability to support larger graphs GraphX [35], GraphLab [23]. Techniques such as vertex re-ordering have been shown to help reduce the number of cache misses [30], [27]. Other algorithms have used vertex covers to reduce the number of necessary intersections [9].

D. GPU Triangle Counting

Leist *et al.* [21] show the first GPU algorithm for triangle counting. In this approach each GPU thread is responsible for a different intersection. Green *et al.* [13] offer a different parallelization scheme for the GPU that uses numerous GPU threads for each adjacency intersection and extends the Merge-Path formulation [26], [11] to Intersect-Path. Intersect-Path improves the performance over [21] by an order of magnitude. Wang *et al.* [34] analyze the performance of several different approaches for triangle counting on the GPU.

E. Streaming and Dynamic Triangle Counting

Similar to the static graph triangle counting algorithms, numerous algorithms have been designed for streaming graphs [2], [1], [19]. Streaming graphs are graphs were the edges are inserted or removed one at a time (typically at high rates) and the number of vertex and edges memory accesses per update is limited to $O(1)$ operations. In the case of triangle counting, many streaming graph algorithms focus on approximating the number of triangles. Furthermore, many streaming graph algorithms focus on the easier case of edges insertions [19]. Becchetti *et al.* [1] note that there are numerous applications

where these approximations are not good enough - this is also true for the case of finding the exact and largest k-truss in a graph.

In addition to streaming graphs algorithms, dynamic graph triangle counting algorithm can be found in [6], [29], [24]. Ediger *et al.* [6] use the STINGER [7] dynamic graph data structure for updating the graph and analytics in batches. For a single update this is simple, however, then the update consists of multiple edges (combined into a single batch) a situation can arise where numerous edges in a batch can create a triangle - such a triangle can go undetected in a parallel environment. Therefore the approach taken in [6] and GraphIn [29] is to recompute the triangles of a vertex from scratch even if only one of its edges is affected.

Recently, a dynamic graph triangle counting algorithm was presented by Makkar *et al.* [24] that shows a new inclusion-exclusion formulation for detecting triangles within a given batch, thereby reducing the amount of work required to update the number of triangles per vertex. This new algorithm does not require recomputing the number of triangles for a whole vertex as required by previous approaches. This algorithm, with its ability to support a batch of edge deletion, is extremely useful for finding the k-truss . Our new algorithm, discussed in Section III extends this algorithm from [24] to support counting triangles per edge rather than per vertex.

III. KTruss Algorithm Using Dynamic Graphs

In this section we present our new algorithm for finding the maximal k-truss (or a specific k-truss) in a graph. The algorithm in [4] suggests recomputing the triangles in every iteration - this is computationally expensive. The algorithm in [33] avoids recomputing triangles for effect edges, yet is sequential. Our new algorithm is both scalable and avoids unnecessary computations. The new algorithm extends the algorithm from [24] and updates the number of triangles per edge rather than per vertex.

Both the algorithms in [4] and [33] require removing edges from the graph once the edges no longer support the necessary number of triangles. This edge deletion process is exactly where the algorithm in [24] excels by avoiding unnecessary computations. Part of the edge deletion process also includes removing the edge from the graph. For sparse graphs, this has proven to be challenging, yet several recent data structures have been created that take care of the graph update at high rates, these include STINGER [7] and cuSTINGER [10], [3] for the GPUs. We use cuSTINGER in our implementation as it supports sorted updates [24] and its data layout is great for both static graph and dynamic graph triangle counting.

Algorithm 1 presents the pseudo code for our new algorithm. While the various functions in the algorithm do not highlight the parallelism in the algorithm, the function calls are all inherently parallel. For example, finding all the vertices with a support smaller than $k-2$ can be done by accessing all the edges in the graph concurrently. Deleting the edges that lack support can also be done in parallel. Lastly, updating the triangle counting of the edges can also be done in parallel.

Algorithm 1 New algorithm for finding k-truss

```
Input: G = (V, E), K
1: procedure UpdateTriangle(Ĝ, E_rem)
2:     Construct G_rem = (V_rem, E_rem)
3:                                          ▷ Update triangles where 1 edge-pair is deleted
4:     parallel for all ⟨u, v⟩ ∈ E_rem do
5:         Intersect(Ĝ_u, Ĝ_v)
6:
7:                                          ▷ Update triangles where 2 edge-pairs is deleted
8:     parallel for all ⟨u, v⟩ ∈ E_rem do
9:         Intersect(Ĝ_u, G_rem,v)
10:
11: function OneK(G, K)
12:                                         ▷ Par-for on all edges in graph looking for sup(e) < k − 2
13:     E_rem ← FindUnderKm2(G, K)
14:     while (|E_rem| ≠ 0) do
15:         Remove(G, E_rem)
16:         if (G = ∅) then
17:             return
18:         UpdateTriangle(G, E_rem)
19:                                         ▷ Par-for on all edges in graph looking for sup(e) < k − 2
20:         E_rem ← FindUnderKm2(G, K)
21:     return
22: function NewKTruss(G, K)
23:     while True do
24:         OneK(G, K)
25:         if (G = ∅) then
26:             return k − 1;
27:         k ← k + 1
```

A. Triangle Subtraction

Consider a triangle in a graph consisting of three vertices u, v, w. The different and **ordered** triangles consisting of vertices are : $\langle u, v, w\rangle$, $\langle u, w, v\rangle$, $\langle v, w, u\rangle$, $\langle v, u, w\rangle$, $\langle w, u, v\rangle$, and $\langle w, v, u\rangle$. As the graph is undirected, there is a certain amount of symmetry: $sup(u, v) = sup(v, u)$. This also means that if (u, v) is deleted, then (v, u) is also deleted. We denote a set of two edges (u, v) and (v, u) as an **edge-pair**.

Thus, given a triangle in the graph prior to the removal of a subset of edge-pairs, the following scenarios can arise from the removal: 1) a single edge-pair is removed, 2) two edge-pairs are removed, and 3) all three edge pairs are removed. If a single edge-pair is removed, then the remaining two edge-pairs need their support to be modified. If two edge-pairs are removed, then the remaining edge-pair needs to be updated. When all three edge-pairs are deleted, then no modifications are required as all the edges are no longer in the graph. Note, that 1) a single deleted edge-pair can affect multiple triangles and 2) these three scenarios capture all the possible changes caused by a deletion of a given edge-pair.

B. Triangle Detection For Single Edge-Pair Deletions

Assuming that the deleted edge-pair consists of vertices u and v, we are required to find all the affected triangles. This requires intersecting the adjacency arrays of u and v in $\hat{G}$ (where $\hat{G} = (\hat{V}, \hat{E})$) is the graph after the removal of the edges). By intersecting (u, v) and (v, u), the common neighbors are found. For each of these common neighbors a triangle is decremented from its edge count. This is the simpler of the two cases.

C. Triangle Detection For Dual Edge-Pair Deletions

The process for detecting and updating the number of triangles when deleting two edge pairs is a bit more complex (the reader is referred to [24] for additional details) and we provide only a sketch of the process. For simplicity, assume that these edges are $(u, v), (v, u), (u, w), (w, u)$. Thus, we are required to update the edge pair (v, w) and (w, v). Given the set of deleted edges E_{rem}, a graph of the deleted edges is created, we call this graph $G_{rem} = (V_{rem}, E_{rem})$.

TABLE I

GPU AND CPU SYSTEM USED IN EXPERIMENTS.

Architecture	Processor	Micro-architecture	SM	SP (per SM)	Total SPs	DRAM Size	DRAM Type
GPU-CUDA	P100	Pascall	56	64	3854	16GB	HBM2

Architecture	Micro-architecture	Processor	Frequency	Cores	LL-Cache	DRAM Size	DRAM Type
CPU x86-64	Broadwell	2× Intel Xeon E5-2695 v4	2.1 GHz	2× 16	2× 45 MB	1024GB	DDR4-2400

TABLE II

NETWORKS USED IN OUR EXPERIMENTS. $|E|$ REFERS TO DIRECTED EDGES. NETWORKS ARE SORTED BASED ON THE NUMBER OF EDGES. EXECUTION TIME IS FOR CUSTINGER-DELTA.

Name	$\|V\|$	$\|E\|$	k_{max}	$Time(s)$
p2p-Gnutella08	6.3K	21K	5	0.007
ca-HepTh	9.8K	26K	32	0.005
ca-HepPh	12K	119K	239	0.009
email-Enron	37K	184K	22	0.026
soc-Epinions1	76K	406K	33	0.09
cit-HepPh	35K	421K	25	0.24
soc-Slashdot0902	82K	504K	36	0.085
roadNet-PA	1M	1.5M	4	0.078
flickrEdges	106K	2.3M	574	0.26
amazon0601	400K	2.4M	11	0.12
graph500-scale18	262K	4.2M	159	0.74
graph500-scale19	524K	8.4M	213	6.8
graph500-scale20	640K	16M	284	17.3
cit-Patents	3.8M	16.5M	36	45.3
graph500-scale21	2.1M	34M	373	117
graph500-scale22	4.2M	67M	485	291
graph500-scale23	8.4M	134M	625	780
Name (Wang & Chang [33])	$\|V\|$	$\|E\|$	k_{max}	$Time(s)$
wiki-Talk	2.4M	4.7M	53	9.07
as-skitter	1.7M	11M	68	57.1
soc-LiveJournal1	4.8M	43M	362	258

Given G_{rem} and $\hat{G}$, to find the common neighbors (v, w) and (w, v), we do the following intersections between the vertex pairs: $\langle u_{rem}, \hat{v}\rangle$, $\langle v_{rem}, \hat{u}\rangle$, $\langle u_{rem}, \hat{w}\rangle$, $\langle w_{rem}, \hat{u}\rangle$ where the adjacencies arrays of $\hat{v}, \hat{u}, \hat{v}$ are in $\hat{G}$ and the adjacencies arrays of $u_{rem}, v_{rem}, w_{rem}$ are in G_{rem}. Note, all four of these intersections are required due to the asymmetry of the adjacency arrays in the two different graphs.

IV. EXPERIMENTAL SETUP

Our experiments are conducted on an NVIDIA P100 GPU connected to an Intel Xeon E5-2695 with 32 cores (details in Table I. The P100 is a Pascal based GPU with 56 SMs and 64 SPs per SM, for a total of 3584 SPs. The P100 has a total of 16GB of HBM2 memory. The Intel Xeon E5-2695 is a Broadwell based processor running at 2.1 GHz with 45MB L3 cache. The server consists of two such processors with a total of 1TB of memory. While the new algorithm is architecture independent, the final implementation targets the GPU. Thus, while the GPU is connected to a high-end Intel processor, in practice we only utilize a single CPU thread.

A. Dynamic Graphs

The *cuSTINGER* data structure is the first fully dynamic graph data structure for the GPU [10], [3]. *cuSTINGER* uses dynamically growing arrays. This allows for improved locality and increased parallel scalability for the GPU's warp based execution model. Specifically, the use of arrays *cuSTINGER* allows for inserting and deleting edges from the graph while ensuring that the edge lists are sorted after the update with relatively low computational effort [24].

B. Benchmarks

We compare the performance of our algorithms with several different implementations, including the baseline benchmarks defined by the HPEC Graph Challenge [31]. The baseline benchmarks are formulated in a linear algebra based formulation in several different programming languages: Matlab/Octave, Julia, and Python. While these programming languages are high-level, they utilize several optimized libraries for the sparse matrix representation as well as for the SpMV operations. We also compare our new algorithm to Graphulo [8] and Wang and Chen [33]. Of these, only [33] and our algorithm use a non linear algebra formulation.

We evaluate the benchmarks on two related but distinct challenges: 1) finding k-truss for a specific value of k and 2) finding all k-trusses up to and including the maximal. We treat these as distinct because there are algorithmic optimizations that are available to the former that aren't to the latter, and vice versa. The results of Wang and Cheng [33] are only for the latter problem. On the other hand, the Graph Challenge benchmarks find trusses of a single k by default. We extended these implementations to iteratively find the maximal k-truss, as suggested in [8].

Our implementations - in our performance analysis we compare two different implementations: 1) *cuSTINGER-Iterative* - a näive algorithm that enumerates the triangles for all the edges in the graph for each iteration of the algorithm using a static graph formulation and 2) *cuSTINGER-Delta* - an implementation of the new algorithm discussed in III. While both these algorithms utilized the cuSTINGER data structure for deleting edges not meeting the support requirements of the k-truss, only cuSTINGER-Delta utilizes the smart update process.

Python - we found the Python implementation to typically be the most stable of the Graph Challenge benchmarks. Whereas the other benchmarks did not always complete, the Python benchmark always did. The Python implementation utilizes SciPy [16] library for its sparse operations. This benchmark is sequential.

Matlab/Octave - this sequential benchmark supports both Matlab and Octave syntax. We used the Octave framework for the execution. We ran into memory-related errors (exceeding memory, seg faults) on some inputs.

Julia - we found that the sequential Julia implementation had several problems, including memory leakage and bad parsing of the input files. Further, there were several cases that the execution was so slow that the benchmarks were stopped.

Wang and Cheng [33] - this benchmark is highly optimized, yet sequential algorithm for finding the k-truss. The code for this algorithm is not open-source, as such we compare the performance of our algorithm directly to the numbers reported in their paper. Details of the system used for these experiments can be found in [33].

TABLE III

EXECUTION TIME COMPARISON FOR FINDING THE MAXIMAL k-TRUSS WITH THOSE FOUND IN [33] AND $k = 3$ WITH THOSE FOUND IN [15].

	P2P	HEP	Amazon	Wiki	Skitter	LJ
Time (Wang & Cheng [33])	< 1	< 1	31	121	281	664
Time (cuSTINGER-Delta)	0.014	0.038	0.43	9.07	57.1	258
Speedup	< 70	< 26	72	13	5	2.57

	S10	S11	S12	S13	S14	S15	S16
Time (Graphulo [15])	1.63	3.93	12.1	37.2	110	3290	8770
Time (cuSTINGER-Delta)	0.003	0.007	0.016	0.042	0.106	0.352	1.18
Speedup	518	595	741	883	1041	9330	7847

Graphulo [8] - while the Graphulo framework has an open-source k-truss implementation, we were unable to collect execution times due to errors. As such, we use the execution times reported in [15], and ran the same inputs on cuSTINGER for comparison. This benchmark is parallel - details of the system used for these experiments can be found in [15].

C. Dataset

The HPEC Graph Challenge [31] has a pre-determined set of networks that are to be used for evaluating the performance of the new algorithm. This consists of graphs from the SNAP dataset [22] and synthetic graphs which are also used for the Graph500 benchmark. Details of the Graph Challenge Graphs can be found in the upper part of Table II. For the sake of brevity we do not present all the graphs in the Graph Challenge list, rather we highlight only a subset of them. We used adjacency files as provided by the Graph Challenge dataset for cuSTINGER, and convert them into their incidence forms for linear-algebra-based benchmarks. The graphs used in [33] also consist of SNAP graphs and can be found in the bottom part Table II, though they are not in the original Graph Challenge List. As such, we preprocessed these graph from their original SNAP [22] format to the one required by the benchmarks. For comparison with the results of Graphulo [15], we generated scale-free graphs (scales 10-16) using their generator script and seed. We then converted these graphs to be run on cuSTINGER.

V. PERFORMANCE ANALYSIS

Fig. 1 (a) and (c) depict the execution of the various algorithms for finding the maximal k-truss and for finding the k-truss for $k = 4$, respectively. The abscissa denotes the number of edges in the input graph and the ordinate depicts the execution time. Note, both the abscissa and the ordinate are log based. Missing data points imply either the benchmark did not finish in a reasonable amount of time (our upper-bound on execution time was approximately 8 hours) or the benchmark did not complete for some reason (exceed memory, crashed in the graph loading phase). The motivation for separately getting execution times for the maximal k-truss versus finding trusses of a predefined k stems from the way that edges are removed from the graph. Consider the case where k is selected to be k_{max}. Even though the output of both these test cases will be the same, the execution time for $k = k_{max}$ will be faster than the execution time when needing to look for k_{max} due to more aggressive edge filtering.

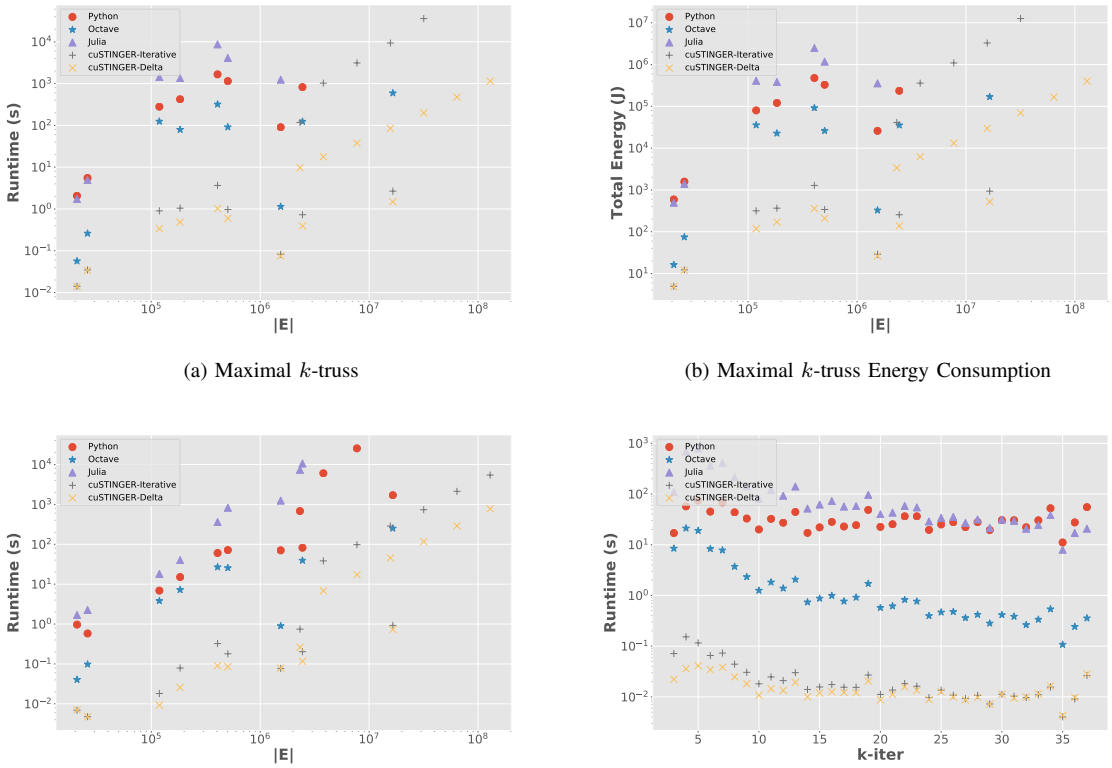
For both these approaches, it can be distinguished from Fig. 1 that both the new implementations, cuSTINGER-Iterative and cuSTINGER-Delta, significantly outperform the remaining benchmarks by orders of magnitude (from $100X$ and upto $10000X$). While it is not fair to compare a GPU implementation with a sequential implementation, we note that the speedup of our algorithms is not just from the use of a NVIDIA GPU, but is due to several additional factors: 1) problem formulation, 2) algorithmic optimizations, and 3) data structure support. It is also worth noting the work of Lee *et al.* [20] compares the performance of CPUs to GPUs and narrowed down the relative speedup of a GPU over a CPU to a significantly smaller value than our achieved speedups. **Problem formulation** - based on the findings of Schank & Wagner [32], it has been established that the time complexity for the linear algebra formulation is higher than the vertex-centric formulation (which we use). **Algorithmic optimizations** - our new algorithm has several important algorithmic optimizations that reduce the total amount of work required for re-enumerating the number of triangles per edge. **Data structure support** - given our usage of cuSTINGER [10], [3] and its support of dynamic graphs, we don't need to recreate the sparse graph after each iteration of the graph. This saves a lot of time on memory allocations.

While we are unable to measure the magnitude of each of the aforementioned optimizations and their contribution to the overall speedup, we do know that our new algorithm, cuSTINGER-Delta, is several times faster than our cuSTINGER-Iterative (we saw an up to 30X difference between the two algorithms). This is directly attributed to our algorithmic optimizations. Also, it is worth noting that due to these optimizations, our implementations scale to significantly larger graphs (well over 100M edges) compared to the remaining benchmarks.

Fig. 1 (b) depicts the projected energy (J) consumption for maximal k-truss run-times and power measurements while running benchmarks on the CPU and GPU respectively. We used the *ipmitool* tool for measuring power. The power measured for the CPU is ($286\frac{J}{s}$). The power measured for the GPU also includes the CPU power and is only slightly higher at ($350\frac{J}{s}$). Based on our measurement, the GPU is not using its peak power. This probably has to do with the fact that the GPU is executing relatively small kernels and for a short period of time. However, the power-performance plot of Fig. 1 (b) does not look significantly different than Fig. 1 (a) - this is due to our new algorithm being extremely power efficient due to its short execution time.

Fig. 1 (d). depicts the runtime of the various implementations as a function of the k, in the process of searching for the maximal k-truss . The *soc-Slashdot0902* graph was selected as it successfully completed on all benchmarks. As expected, generally the time per iteration decays over time as the number of active edges (edges under consideration) tends to monotonically decrease with the increase of k [2]. The

[2]Note, the execution time for a given k is dependent on the number of sub-iterations for that given k - this can also explain the increase in execution time from $k = 3$ to $k = 4$



(a) Maximal k-truss

(b) Maximal k-truss Energy Consumption

(c) $k = 4$

(d) Maximal k-truss , per iteration

Fig. 1. (a),(b), (c): execution times and energy consumption for finding the trusses as a function of the graph size (number of edges in the input graph): (a) maximal k-truss of the graph, (b) the energy consumption for find the maximal k-truss , and (c) for trusses of $k = 4$. These are log-log plots. (d) Execution time of the various algorithms for finding the maximal k-truss ,for the *soc-Slashdot*, as a function of the iteration (k). Note, that while $k_{max} = 36$, the algorithms terminate for $k = 37$ when the graph becomes empty. The Y-axis is log-scale.

only exception is the Python benchmark, where the time per iteration remains in a constant range for this input. Our new algorithms are orders of magnitude faster than the Graph Challenge Benchmark [31].

Comparison with Wang and Cheng [33] Table III (upper) compares our new algorithms with the algorithm by Wang & Cheng [33]. We picked six inputs of varying sizes from their list of tested graphs (all from the SNAP graph repository). While [33] had numerous implementations, we compare against their bottom-up approach as it is the most similar to our algorithm. We show speedups of our algorithm over theirs. In all cases the cuSTINGER-Delta implementation outperforms [33].

Comparison with Graphulo [15] Lastly, we compare our new algorithm, Table III (lower), with the Graphulo framework which has a linear algebra based implementation for finding a k-truss of a given size, using inputs from their paper. The Graphulo framework is intended to process larger graphs. Similar to the other linear algebra based formulations, our new algorithm is orders of magnitude faster.

VI. Conclusions

In this paper we showed a new algorithm for finding k-truss subgraphs. Our new algorithm uses a dynamic graph formulation and exploits two important features: 1) it utilizes a dynamic graph data structure that can insert and remove edges without creating a new data structure after each update and 2) it avoids recomputing the number of triangles per edge in each iteration of the algorithm after the edge removals. The latter of these properties means that our new algorithm does a fraction of the work that static graph algorithms do - this leads to significant speedups. In addition to this, our new algorithm is also extremely scalable and can concurrently detect when a deleted edge is part of multiple triangles and it can update all the affected edges (in parallel).

While our algorithm is architecture independent, our CUDA based implementation showed massive speedups over the Graph Challenge benchmarks. There were numerous instances where the Graph Challenge benchmarks did not complete in reasonable amount of time (8 hours) whereas our algorithm finished in a few minutes. Our new algorithm was often over a hundred times faster than the best performing Graph Challenge Benchmarks and thousands of times faster than the remaining benchmarks. Our algorithm also scaled to much larger graphs than in the benchmarks. While part of our speedup can be attributed to the usage of an NVIDIA GPU, the bigger part of the speedup is due to the new algorithmic optimizations we showed. Further, our new algorithm is in some case over $70X$ faster than a recently developed and optimized algorithm that is inherently sequential.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

ACKNOWLEDGMENTS

Funding for the researchers at Georgia Institute of Technology and University of Southern California 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.

REFERENCES

[1] 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.

[2] 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.

[3] F. Busato, O. Green, N. Bombieri, and D. Bader, "A Sparse Dynamic Graph and Matrix Data Structure," Georgia Tech Technical Report.

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

[5] J. Cohen, "Graph Twiddling in a Map-Reduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[6] 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.

[7] 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.

[8] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear Algebra Graph Kernels for NoSQL Databases," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 822–830.

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

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

[11] 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.

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

[13] 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.

[14] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying K-Truss Community in Large and Dynamic Graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1311–1322.

[15] D. Hutchison, J. Kepner, V. Gadepally, and B. Howe, "From nosql accumulo to newsql graphulo: Design and utility of graph algorithms inside a bigtable database," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–9.

[16] E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open Source Scientific Tools for Python," 2014.

[17] H. Kabir and K. Madduri, "Parallel k-truss decomposition on multicore systems," in *IEEE Proc. High Performance Embedded Computing Workshop (HPEC)*, 2017.

[18] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," *arXiv preprint arXiv:1707.02000*, 2017.

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

[20] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 451–460, 2010.

[21] 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.

[22] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[23] 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.

[24] D. Makkar, D. Bader, and O. Green, "Exact and Parallel Triangle Counting in Streaming Graphs," Georgia Tech Technical Report.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *2010 ACM SIGMOD Int'l Conf. on Management of data*, 2010, pp. 135–146.

[26] 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.

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

[28] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015, pp. 927–937.

[29] 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.

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

[31] S. Siddharth, 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 Embedded Computing Workshop (HPEC)*, 2017.

[32] 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.

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

[34] 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.

[35] 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.