Truss Analytics Algorithms and Integration in Arkouda

ZHIHUI DU, JOSEPH PATCHETT, OLIVER ALVARADO RODRIGUEZ, and DAVID A. BADER, New Jersey Institute of Technology, USA

The K-Truss of a graph is a cohesive subgraph that has been widely used for community detection in applications such as social networks and security analysis. In this paper, we first propose one optimized triangle search kernel with a few operations that can be used in both triangle counting and triangle search to replace the existing list intersection method. Based on the optimized kernel, three truss analytics algorithms, an optimized K-Truss parallel algorithm, a maximal K-Truss parallel algorithm, and a Truss decomposition parallel algorithm, are developed to enable different kinds of graph analysis efficiently. Moreover, all proposed parallel algorithms have been implemented in the highly-productive parallel language Chapel and integrated into the open-source framework Arkouda. Experimental results compared with the existing list intersection-based method show that for both synthetic and real-world graphs, the proposed method can significantly improve the performance of truss analysis on large graphs. The implemented method is publicly available from GitHub (https://github.com/Bears-R-Us/arkouda-njit).

Additional Key Words and Phrases: K-Truss, Triangle Counting, Graph Analytics

ACM Reference Format:
Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, and David A. Bader. 2022. Truss Analytics Algorithms and Integration in Arkouda. In . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

K-Trusses [7] have been widely used to discover close relationships in a graph and are more rigorous than *k-cores* (where all the nodes have a degree at least k in a subgraph) but less stringent than *k-cliques* (where all the nodes are connected pairwise in a subgraph). The clique decision problem is NP-complete, but *K-Trusses* can be computed in polynomial time, so *K-Trusses* can be used in large graph analysis. Despite this, the increasing size of real-world graphs has become a great challenge for *K-Truss* analysis.

At the same time, exploratory data analysis (EDA) [2, 14, 19] has become a critical method to discover the value of data quickly. Unfortunately, most EDA tools, which often run on laptops or common personal computers, cannot handle large data efficiently, let alone produce highly productive analysis results. Developing efficient *K-Truss* algorithms to enable most EDA users to conduct their analysis on large graphs productively is the primary goal of this research.

Arkouda [22, 24] is an EDA framework under early development that brings together the productivity of Python at the front-end with the high-performance computing capability of Chapel [6] at the back-end. In this work, we integrate the proposed *K-Truss* parallel algorithms into Arkouda so that data scientists can take advantage of Python using their laptops to conduct interactive real-world graph analysis on very large compute platforms (including clusters) productively.

The major contributions of this paper are as follows.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CHIUW'2022, June 10, 2022, Virtual

© 2022 Association for Computing Machinery.

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

(1) A fast triangle search kernel that can take advantage of the properties of real-world graphs is proposed. Multiple parallel and performance optimization methods have been employed in our *K-Truss* algorithms.
(2) The proposed *K-Truss* algorithms have been implemented into the open-source framework Arkouda to support high-level Python users to analyze large graphs using their laptops with high productivity.
(3) Experimental results on synthetic and real-world graphs show that the proposed performance optimization methods achieve significant speedup compared with the widely used list intersection method.

2 ALGORITHM DESIGN

2.1 Notation, Analysis, and Data Structure

2.1.1 Notation. The graph, $G = (V, E)$, comprises the vertex set V and the edge set E. We use $\triangle(e, G)$ to express the set of all triangles including edge $e = \langle u, v \rangle$ in the graph G. The support of e, which means the number of triangles including edge $e = \langle u, v \rangle$ in G, is expressed as $sup(e, G) = |\triangle(e, G)|$.

Given an integer $K \geq 2$, the *K-Truss* of G is defined as the maximal subgraph $SubG = \langle SubV, SubE \rangle$ of G such that $\forall e \in SubE \subseteq E$, we will have $sup(e, SubG) \geq K - 2$. The *Max K-Truss* is the *K-Truss* that has the maximum value of K among all the non-empty *K-Trusses* of G. For all $e \in E$, the *truss value* or *trussness* of e is defined as the K of the maximal *K-Truss* that includes e. It is expressed as $truss(e, G)$. Based on the definition, we have $truss(e, G) \leq sup(e, G)$. The *truss decomposition* of a graph G is assigning each edge with its truss value.

2.1.2 Bound Analysis. Based on the definition, the minimum value of K is 2. We use $MaxK$ to denote the maximum K value for a graph G and the corresponding subgraph $MaxKG = \langle MaxV, MaxE \rangle$. Then, $\forall e \in MaxE$, we will have $sup(e, MaxKG) \geq MaxK - 2$. This means that the total number of vertices in $MaxV$ should meet $|MaxV| \geq MaxK$ (if $|MaxV| < MaxK$, $\forall e \in MaxE$, we cannot have another $MaxK - 2$ different vertices to form triangles with e). We define the degree of an edge $ld(e) = min(degree(u), degree(v))$. If an edge $e = \langle u, v \rangle \in MaxE$, we must have $ld(e) \geq MaxK - 1$. So for a maximal *K-Truss* of a given graph, the total number of edges in the subgraph cannot be less than $MaxK$ and the degree of each vertex should not be less than $MaxK - 1$. So we can sort the vertices in decreasing order based on their degrees and add them into a set $VSet$ step by step. Let $Dmin = min\{degree(v)|v \in VSet\}$ and $k_{up} = max\{x|x = min\{Dmin + 1, |VSet|\}\}$ for all possible $VSet$, we will have $MaxK \leq k_{up}$. In this way, we may use k_{up} to set the upper bound of $MaxK$.

2.1.3 Double Index Data Structure. This paper focuses on sparse graphs that can model a wide range of real-world applications such as social networks, bioinformatics, and cybersecurity. A compact and efficient Double-Index (*DI*) sparse graph data structure (edge index arrays and vertex index arrays) that was developed in our previous work [13] is employed in this research to support our *K-Truss* analysis. The DI data structure can support both edge-based search and vertex-based search quickly. At the same time, the edge index arrays can be used to partition a graph's edges equally to achieve load balance for edge search based graph algorithms. All these features can support a quick triangle search.

2.2 Novel Triangle Searching Kernels

Given edges $e = \langle u, v \rangle \in E$, if the adjacency lists of u and v are Adj_u and Adj_v, then the number of triangles including e should be $|Adj_u \cap Adj_v|$. This is the formula of the widely used list intersection method [9] in *K-Truss* analysis. If Adj_u and Adj_v are sorted, then the execution time of sequential list intersection to find all triangles including edge $e = \langle u, v \rangle$ can be $|Adj_u| + |Adj_v|$ [16]. If we use a small number of parallel threads as possible to find all triangles, it will take $log_2|Adj_v|$ (we assume $|Adj_u| \leq |Adj_v|$ and $|Adj_u|$ threads will run a binary search in Adj_v in parallel). However, this method does not take advantage of the property of vertex $w \in Adj_u \cap Adj_v$ to improve the parallel performance. So, we propose a novel minimum search method to significantly improve the performance of parallel triangle counting and searching for real-world graphs.

Let h (l) be u or v which has more (less) adjacent vertices. Adj_h (Adj_l) be Adj_u or Adj_v that has more (less) elements. $\forall w \in Adj_l$, let Adj_w be the adjacency list of w. The proposed minimum search method directly checks if there is a third edge $\langle w, h \rangle$ that can close the wedge $\langle l, h \rangle$ and $\langle l, w \rangle$ to form a triangle. Furthermore, the check method will be based on the degrees of both vertex w and h. If Adj_w and Adj_h are sorted, the parallel minimum search method will need $log_2(min(|Adj_w|, |Adj_h|))$ instead of $log_2(|Adj_h|)$ time. So, $log_2(|Adj_h|) - log_2(min(|Adj_w|, |Adj_h|))$ operations are saved for checking the third edge $\langle w, h \rangle$. The larger difference in $|Adj_w|$ and $|Adj_h|$, the more operations can be saved. The total time to get all the triangles including given edge $\langle u, v \rangle$ in parallel can be calculated as in Eq.1.

$$\max_{w \in Adj_l} log_2(min(|Adj_w|, |Adj_h|)) \tag{1}$$

List intersection does not care about the degree of the third vertex that may form a triangle with the given two vertices. However, the proposed minimum search is a fine-grained method. It will consider the degrees of the third vertex to reduce the search operations as much as possible. For any vertex $w \in Adj_l$, if $|Adj_w| \geq |Adj_h|$, the number of operations to decide if u, v, w can form a triangle will be $log_2|Adj_h|$ that is the same as in the list intersection. If $|Adj_w| < |Adj_h|$, then our method will have fewer operations.

The standard list intersection method can only work on two given lists. However, our method can take advantage of the adjacency list of the third vertex to further exploit the optimization space to reduce the total number of operations. If $|Adj_l| = 4$, $\forall w \in adj_l, |Adj_w| \leq 8$ and $|Adj_h| = 1024$, it will need 4 parallel threads and each thread will execute $\lceil log_2 1024 \rceil = 10$ operations to search the triangles containing given edge $\langle u, v \rangle$. The proposed novel method will also need 4 parallel threads, and each thread will take $\lceil log_2 8 \rceil = 3$ search operations. It is less than half of the standard list intersection's parallel execution time. For real-world graphs, their edges are highly skewed, and only a tiny amount of vertices have huge adjacency lists. So, our method can avoid the large adjacency list searches and work on smaller adjacency lists to improve the parallel performance.

2.3 Naive K-Truss Parallel Algorithm

In this section, we will first introduce the naive method to show the basic idea of *K-Truss* analysis.

Algorithm 1: Naive K-Truss Parallel Algorithm

```
NaiveKTruss(G, k)
/* G = ⟨E, V⟩ is the input graph with edge set E and vertex set V. k is the given K-Truss value. */
EdgeDel[] = −1 // initialize all edges as not deleted
while there is any edge can be deleted do
    sup[] = 0 // initialize the triangle counting array
    forall (undeleted edge e = ⟨u, v⟩ ∈ E) && (e is local) do
        calculate sup(e, G) using list intersection or minimum search method
        sup[e] = sup(e, G)
    end
    forall (e = ⟨u, v⟩ ∈ E) && (e is local) do
        if (EdgeDel[e] == −1) && (sup[e] < k − 2) then
            EdgeDel[e] = k − 1
        end
    end
end
return EdgeDel
```

Based on the *DI* sparse graph data structure, which can locate both vertex and edge in constant time [13], we first develop a naive but distributed parallel framework for *K-Truss* algorithm that can be easily implemented in Chapel.

Peeling [7] is a simple but very efficient *K-Truss* subgraph generation method. It removes the edges whose number of triangles is less than $K - 2$ step by step, like peeling an onion. We propose a naive version of this method in Alg.1.

Our naive algorithm can run on distributed memory clusters to take advantage of multiple computing resources to handle huge graphs. At the same time, in each shared-memory multicore/SMP node, the triangle counting and the checking for different edges on the current locale can also be executed in parallel. The Chapel $forall$ parallel construct can implement implicit synchronization among all the parallel threads so we do not need explicit synchronization operation between the first $forall$ construct from lines 5 to 8 and the second $forall$ construct from lines 9 to 13.

The naive *K-Truss* algorithm shows how we can exploit parallelism and employ our novel triangle search kernel in *K-Truss* analysis using Chapel.

2.4 Optimized K-Truss Parallel Algorithm

The naive *K-Truss* algorithm is simple and easy to implement. Under most scenarios, it cannot achieve high performance even though it has an excellent parallel framework. The reason for low performance is that it will recalculate the number of triangles in each iteration. The more iterations it has, the more unnecessary triangle counting operations will be executed.

If an edge is deleted, all other edges that can form a triangle with such an edge will be affected. The affected edges can be found from the deleted edges. The basic idea of the optimized method is that we just update the number of triangles of affected edges instead of recalculating the number of triangles of all edges.

Algorithm 2: Optimized K-Truss Parallel Algorithm

```
OptKTruss(G,k)
/* G = ⟨E,V⟩ is the input graph with edges set E and vertices set V. k is the given K-Truss value.  */
EdgeDel[] = −1 // initialize all edges as undeleted
sup[] = 0 // initialize the support array of each edge
SetDel = φ;SetAff = φ
forall ( edge e ∈ E) && (e is local) do
    sup[e] = sup(e,G) using minimum search method
end
forall (e ∈ E) && (e is local) do
    if (EdgeDel[e] == −1) && (sup[e] < k − 2) then
        EdgeDel[e] = 1 − k
        Add e into SetDel
    end
end
while (SetDel is not empty) do
    forall (e1 ∈ SetDel) && (e1 is local) do
        using minimum search method to find e2 and e3 that can form a triangle with e1
        reduce the support of e2 and e3 if they are undeleted edges
        add the affected edges into SetAff if their supports are less than k − 2
    end
    forall (e ∈ SetDel) && (e is local) do
        if (EdgeDel[e] == 1 − k) then
            EdgeDel[e] = k − 1
        end
    end
    SetDel.clear()
    SetDel <=> SetAff // switch the values of the two sets.
end
return EdgeDel
```

The major optimization method of algorithm Alg. 2 is parallel searching affected edges to avoid repeat triangle counting[1, 4, 10, 15, 21]. If edge e_1 was deleted and e_1, e_2, and e_3 can form a triangle, then e_2 and e_3 are the affected edges of e_1. We will reduce the number of triangles of unremoved edges that will be affected by the removed edges. Two removed edges may affect the same unremoved edge in the same triangle. So, our algorithm should avoid updating the same undeleted edge in the same triangle twice. At the same time, one unremoved edge may be affected by two removed edges in two different triangles. So, we use an atomic subtraction operation provided by Chapel to reduce the support of the unremoved edge to avoid the write race. Chapel's atomic array is very convenient to support such operations.

After all affected edges have been updated, the unremoved edges whose support values are less than $k-2$ will also be removed. All the newly removed edges will be used to parallel search new affected edges until no affected edges can be found. This optimization can avoid repeat triangle counting from scratch, so it can significantly reduce the total number of operations.

Alg. 2 includes two main procedures. The first procedure is the minimum search kernel based triangle counting part, just like the naive method. The second part is the affected edges search based support updating method. Two additional data structures are introduced in the optimized algorithms. $SetDel$ is the set of edges that were just removed. $SetAff$ is the set of edges that may be deleted because we delete the edges in $SetDel$ will affect and reduce their support values. Chapel's Set module can support set operations well.

The proposed minimum search kernel can be adopted in the optimized algorithm to search and update the affected edges in a much smaller set, and no unnecessary operations will be executed. At the same time, each deleted edge will be assigned a thread to search the affected edges, and all the threads are executed in parallel.

2.5 Max K-Truss Parallel Algorithm

Based on the proposed optimized parallel *K-Truss* algorithm, we can design the algorithm to find the maximum truss value of the given graph. We develop a *DownwardSearch* method to locate the maximum truss value quickly.

Based on the discussion in section 2.1, we can first get the upper bound k_{up} of the maximum *K-Truss* value. So we only need to check the maximum k value in range $[3..k_{up}]$ that will not delete all the edges in a graph. Then k will be the maximum *K-Truss* value of the given graph. In Alg.3 we give the description of our *Max K-Truss* parallel algorithm.

In line 2 we initialize the range of maximum *K-Truss* search value k_{low} and k_{up}. Based on the feature of *K-Truss* search, we have the inequality $k_{low} - 1 \leq MaxK \leq k_{up}$. We call the *DownwardSearch* procedure at line 3 to return the maximum k value and the edge array *EdgeDel* that describes the remaining subgraph of a given graph.

In lines from 4 to 30, we implement the *DownwardSearch* search function. After checking the lower and upper bounds, we update the search bounds in lines from 15 to 28. If we find that the k_{mid} value is too large, we will continuously reduce the value of k_{up} and k_{mid} until we find a k_{mid} value that will not delete all the edges. This is the downward search procedure. The particular downward search procedure is different from the general binary search method.

Based on our optimized *K-Truss* parallel algorithm, the *Truss Decomposition* procedure is straightforward. We just need to increase the value of k step-by-step until all edges have been removed. So we ignore the detailed description here.

3 INTEGRATION WITH ARKOUDA

Arkouda is an open-source framework that allows data scientists to take the next step in data analytics from their own laptops by transferring the burden of high-performance computing to a back-end server. Arkouda contains three major components: an interactive Python front-end,

Algorithm 3: Max K-Truss Parallel Algorithm

```
MaxKTruss(G)
/* G = ⟨E,V⟩ is the input graph with edges set E and vertices set V.    */
Let k_low = 3 and set k_up based on the proposed analysis method
return DownWardSearch(G, k_low, k_up)
function DownWardSearch(G, k_low, k_up)
EdgeDel = kTruss(G, k_low)
if (All edges have been deleted) then
    return (k_low − 1, EdgeDel)
end
else
    EdgeDel = kTruss(G, k_up)
    if (there are undeleted edges in EdgeDel) then
        return (k_up, EdgeDel)
    end
    else
        k_mid = (k_low + k_up)/2
        EdgeDel = kTruss(G, k_mid)
        while (All edges have been deleted in EdgeDel) do
            k_up = k_mid − 1
            k_mid = (k_low + k_up)/2
            EdgeDel = kTruss(G, k_mid)
        end
        if (k_mid == k_up − 1) then
            return (k_mid, EdgeDel)
        end
        else
            k_low = k_mid + 1
            return DownwardSearch(G, k_low, k_up)
        end
    end
end
```

a ZeroMQ middleware, and a Chapel back-end. The front-end python mimics the workflow of a Jupyter notebook and abstracts away the computations done on the back-end.

After implementing the kernel Chapel data structure and algorithm, we need to follow Arkouda's integration rule to make the new functionality work well to create an end-to-end response from Chapel to Python.

We developed our calling method in Python as *KTruss(graph, k)* where to be called, the user needs to pass a graph to the function as well as some integer k. This k can be either -1 (for *Max K-Truss*) , -2 (for *Truss Decomposition*), or ≥ 3. This method is added into Arkouda's front-end file *graph.py*.

The developed Chapel functions are located in the *TrussMsg.chpl* file. This procedure accepts the command's name, a payload message, and a symbol table name where our data will be housed from the Chapel back-end. The payload is parsed to extract the name of the Chapel graph class that houses our graph data, and then using the name, we extract the data from the symbol table and then work with it to run our algorithm. These are the integration steps for Arkouda.

4 EXPERIMENTS

4.1 Experimental Setup

Our datasets were chosen from a selection of publicly available synthetic and real-world datasets. Real-world graphs have degree distributions that follow a power-law distribution, while sparse synthetic graphs follow a normal distribution. The real-world graphs are downloaded from SNAP [1]. The synthetic graphs were Delaunay from the DIMACS10.

[1] Stanford Large Network Dataset Collection, https://snap.stanford.edu/data/

Experiments were performed on a 32-node high-performance server connected through Infiniband FDR 56 Gbit/s loaded with 2 x Intel Xeon E5-2650 v3 @ 2.30GHz CPUs with ten cores per CPU. It also has 512GB of DDR4 RAM per node. The utilized Chapel and Arkouda version used during testing were 1.25.0 and 2022.3.15 respectively.

4.2 Performance Results

This part will provide three kinds of truss analysis algorithms' results based on our minimum search based triangle search kernel. We implemented three different versions to provide the comparison results for the k-truss algorithm (we let k=4 in the experiments). Table 1 shows the experimental results. Column "LI Naive K-Truss" is the execution time of the list intersection method based on the naive k-truss algorithm framework. "MS Naive K-Truss" is the execution time of the minimum search method based on the naive k-truss algorithm framework. "MS Opt K-Truss" results from a minimum search method based on the optimized k-truss algorithm framework. It will search and update the affected edges without recalculating the number of triangles from scratch. Based on the results of the three experiments, we can see the advantage of the minimum search method compared with the list intersection method. At the same time, we can further show the optimized search based method compared with the naive method. "MS Max K-Truss" is the execution time of the minimum search based max k-truss method. "MS Truss Decomposition" is the execution time of the minimum search based truss decomposition method. We let the "LI Naive K-Truss" as the baseline, "Speedup 1" is the performance improvement of our "MS Naive K-Truss" algorithm compared with the baseline. "Speedup 2" is the performance improvement of our "MS Opt K-Truss" algorithm compared with the baseline.

The experimental results in Table 1 show that the proposed minimum search based triangle search method is better than the widely used list intersection method. The results from "Speedup 1" show that most graphs can achieve more than two times speedup. "Speedup 2" shows that most graphs can achieve more than ten times speedup. Some can achieve more than one hundred times speedup. Furthermore, based on our minimum search based kernel, the optimized affected edges search method can also significantly improve the performance. All our k-truss algorithms are based on the novel minimum search kernel, and the experimental results show that this kernel can help to improve the performance compared with the widely used list intersection method.

Table 1. Execution time (seconds) of different k-truss algorithms and speedup compared with list intersection method.

Graph	LI Naive K-Truss	MS Naive K-Truss	MS Opt K-Truss	MS Max K Truss	MS Truss Decomposition	Speedup 1	Speedup 2
amazon0601	1008.58	509.29	60.61	93.22	66.22	2.0	16.6
as-caida20071105	16.70	2.98	1.00	1.73	0.88	5.6	16.7
ca-AstroPh	113.28	56.11	9.64	11.16	5.17	2.0	11.7
ca-CondMat	23.52	11.58	2.11	2.58	2.21	2.0	11.2
ca-GrQc	2.49	1.24	0.29	0.35	0.36	2.0	8.6
ca-HepPh	29.33	14.69	3.07	3.22	3.45	2.0	9.6
ca-HepTh	3.88	1.93	0.50	0.61	0.61	2.0	7.7
com-Youtube	4885.27	302.37	55.72	71.89	61.94	16.2	87.7
delaunay_n10	2.04	1.05	0.08	0.09	0.08	1.9	25.5
delaunay_n11	5.50	2.81	0.16	0.18	0.16	2.0	34.2
delaunay_n12	14.00	7.15	0.32	0.36	0.31	2.0	44.3
delaunay_n13	36.69	18.74	0.62	0.70	0.61	2.0	58.9
delaunay_n14	98.61	50.46	1.23	1.46	1.22	2.0	79.9
delaunay_n15	266.96	136.52	2.49	2.93	2.45	2.0	107.3
delaunay_n16	735.75	378.16	4.91	5.83	4.87	1.9	149.8

5 RELATED WORK

GraphChallenge [2] is a vital effort combined by academics and industry to develop new solutions for analyzing graphs and sparse data. *K-Truss* is one of the graph challenging algorithms. The seminal paper about truss decomposition is that by Cohen[7], who introduced the concept of *K-Truss*, motivating it as an effective community indicator.

In this paper, we borrow many fine-grained optimization methods on GPUs to develop our algorithm, such as parallel triangles search for given edge. Green et al.[15] uses a new dynamic graph formulation to achieve scalable performance on GPUs for both *K-Truss* and *Max K-Truss* analysis. Almasri et al.[1] can use multiple GPUs to improve the binary search *Max K-Truss* performance on large graphs. Blanco et al. [5] presents a linear-algebraic formulation of the *K-Truss* graph algorithm and demonstrates the efficiency of their fine-grained parallel approach on both CPU and GPU. Diab et al.[12] explores the design space of different optimizations on GPUs including edge-centric[1, 10, 21, 26] and vertex-centric parallelization[3], directing edges by degree[4, 17], tiling[17, 27], parallelizing intersections[4, 17], removing deleted edges intermediately[4, 5, 10], and recomputing support values to achieve better performance for specific input graphs[1, 4, 10, 15, 21]. Date et al. [10, 21] takes advantage of the heterogeneous platform (CPU+GPU) to improve the performance.

Besides on GPUs, there are a lot researches [20] [25][26] [18][11][8][23] [9] that try to optimize the performance of *K-Truss* from different aspects. We develop a fast triangle search kernel to optimize the performance by significantly reducing the total number of triangle search operations. At the same time, our parallel method is implemented using high-level parallel language Chapel and integrated into Arkouda to enable productive *K-Truss* analysis.

6 CONCLUSION

Productive *K-Truss* analysis is critical to exploit the value of large networks. *K-Truss* is a widely employed community detection method for different applications. This paper develops a very fast triangle search kernel to replace the existing list intersection method. Based on our fast triangle search kernel, we develop highly optimized *K-Truss* analysis algorithms for different truss analyses. Furthermore, our algorithms have been implemented in a productive high-level parallel language Chapel. Our implementation method can employ parallel platforms to achieve high performance and code development efficiency. Our code has been integrated with an open-source EDA framework Arkouda. So the increasing number of developers familiar with Python in the EDA community can easily use Python on their laptops to conduct large graph analysis productively. This work can support more users to solve their real-world problems with high productivity without knowing the low-level implementations.

ACKNOWLEDGMENTS

We appreciate the help from the Arkouda co-creators Michael Merrill and William Reus, as well as Brad Chamberlain, Elliot Joseph Ronaghan, Engin Kayraklioglu, David Longnecker and the Chapel community when we integrated the algorithms into Arkouda. This research was funded in part by NSF grant number CCF-2109988.

REFERENCES

[1] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on k-truss decomposition on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[2] John T Behrens. 1997. Principles and procedures of exploratory data analysis. *Psychological Methods* 2, 2 (1997), 131.

[2] https://graphchallenge.mit.edu/challenges

[3] Mauro Bisson and Massimiliano Fatica. 2017. Static graph challenge on GPU. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[4] Mauro Bisson and Massimiliano Fatica. 2018. Update on static graph challenge on GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–8.

[5] Mark Blanco, Tze Meng Low, and Kyungjoo Kim. 2019. Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[6] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[7] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008).

[8] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Discovering k-trusses in large-scale networks. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.

[9] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2020. Truly Scalable K-Truss and Max-Truss Algorithms for Community Detection in Graphs. *IEEE Access* 8 (2020), 139096–139109.

[10] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. 2017. Collaborative (CPU+GPU) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[11] Timothy A Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.

[12] Safaa Diab, Mhd Ghaith Olabi, and Izzat El Hajj. 2020. KTrussExplorer: Exploring the design space of k-truss decomposition optimizations on GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[13] Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A Bader. 2021. Interactive Graph Stream Analytics in Arkouda. *Algorithms* 14, 8 (2021), 221.

[14] Irving J Good. 1983. The philosophy of exploratory data analysis. *Philosophy of science* 50, 2 (1983), 283–295.

[15] Oded Green, James Fox, Euna Kim, Federico Busato, Nicola Bombieri, Kartik Lakhotia, Shijie Zhou, Shreyas Singapura, Hanqing Zeng, Rajgopal Kannan, et al. 2017. Quickly finding a truss in a haystack. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[16] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[17] Yang Hu, Pradeep Kumar, Guy Swope, and H Howie Huang. 2017. Trix: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[18] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. 2018. Triangle counting and truss decomposition using fpga. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[19] Andrew T Jebb, Scott Parrigon, and Sang Eun Woo. 2017. Exploratory data analysis as a foundation of inductive research. *Human Resource Management Review* 27, 2 (2017), 265–276.

[20] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[21] Vikram S Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2018. Collaborative (CPU+ GPU) algorithms for triangle counting and truss decomposition. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[22] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*. 28–28.

[23] Roger Pearce and Geoffrey Sanders. 2018. K-truss decomposition for scale-free graphs at scale in distributed memory. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.

[24] William Reus. 2020. CHIUW 2020 Keynote: Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science. In *Chapel Implementers and Users Workshop, 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 650–650.

[25] Shaden Smith, Xing Liu, Nesreen K Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. 2017. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[26] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. 2017. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[27] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan Berry, Michael Wolf, Jeffrey S Young, and Ümit V ÇatalyÜrek. 2019. Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments:(Update on static graph challenge). In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*.

IEEE, 1–4.