TriX: Triangle Counting at Extreme Scale

Yang Hu*, Pradeep Kumar*, Guy Swope†, and H. Howie Huang*
*The George Washington University
†Raytheon Company

***Abstract*—Triangle counting is widely used in many applications including spam detection, link recommendation, and social network analysis. The DARPA Graph Challenge seeks a scalable solution for triangle counting on big graphs. In this paper we present TriX, a scalable triangle counting framework, which is comprised of a 2-D graph partition strategy and a binary search based intersection algorithm designed for GPUs. The 2-D partition provides balanced work division among multiple GPUs. On the other hand, binary search based intersection achieves fine-grained parallelism on GPUs via intra-warp scheduling and coalesced memory access. TriX is able to scale to a large number of GPUs, and count triangles on billion-node graph (2 billion node, 64 billion edges) within 35 minutes, achieving over 16 million traverse edges per second (TEPS).**

I. Introduction

Graph structure plays an important role in a variety of applications such as social media, web, and e-commerce, where the relationship between different entities carries as much, if not more, importance as the entities themselves. Such relationship can be utilized in identifying different community structures, for which the triangle counting algorithm is developed to enumerate the common neighbors between two vertices of every edge in a graph. Given its importance, the DARPA Graph Challenge seeks high-performance, scalable solutions for triangle counting [18].

Traditionally, triangle counting can be executed only on shared-memory systems [13]. However, the exponential growth of the graphs necessitates external memory based triangle counting [11], [5], [7] to process graphs larger than memory size by taking additional disk I/Os. To this end, several prior projects [5], [11], [2], [16], [12] use external memory model [1] and require to load data from storage such as hard disk. Unfortunately, these external memory algorithms take significantly longer time to run than in-memory algorithms.

We envision that to provide a fast, scalable solution for triangle counting, a host of issues must be addressed. At a high level, there is a need for an efficient graph partitioning which should be able to understand the limited memory available in the system, and an intelligent work scheduling strategy that can not only efficiently utilize the variety of compute cores in an optimum way but also understand that the data is not always in the memory but may reside in the disks. Meanwhile, at the low level, the compute cores must run a highly efficient version of the counting algorithm on its partitioned data.

In this paper, we present *TriX*, a scalable triangle counting framework that combines a 2-D graph partition strategy [23] and a binary search based intersection algorithm designed for graphics processing units (GPUs) [24]. TriX is able to count triangles on the large graph (e.g., 2 billion vertices and 64 billion edges) within 35 minutes on a small cluster of 32 machines equipped with a single GPU each, while fetching data from the external memory.

This result has been made possible due to our 2-D partitioning that allows us to efficiently co-ordinate the triangle counting work on individual partitions, IOs from the disk, and workload distribution. Furthermore, we utilize massive parallelism available on GPUs [15] for performing concurrent intersection tasks. GPUs offer two key features that are beneficial for triangle counting. First, there are thousands of small cores which can be utilized to run the algorithm in a massively parallel form. Second, the high memory bandwidth, e.g., 288GB/s in K40, can be used for fast data I/O. In this work, we have observed that traditional merge-based intersection is only suited for CPUs, and suffers excessively on GPUs due to strided memory access and branch diversion. To address this problem, we have developed a binary search based intersection and show that it achieves very high throughput on GPUs.

The remaining of the paper is organized as follows. Section II presents background related to graphs and triangle counting. Section III provides a high-level overview of TriX, while Section IV and V present technical designs. We also give an overview of our K-truss algorithm in Section VI. Evaluations are presented in Section VII, related work in Section VIII, and we conclude in Section IX.

II. Background

A graph $G = (V, E)$ is constituted by its vertex set V and edge set E that each edge is a link between two vertices. Figure 1 presents an example graph, and the storage formats used in this paper. An *edge list* format stores all edge tuples in form of an edge array. Alternatively, *compressed sparse row (CSR)* contains two arrays, the adjacency array $\{A\}$ and begin position array $\{B\}$. The adjacency array A stores the neighbor vertices of each vertex sorted by their ID. And the begin position array indicates the offsets of neighbors of any vertex in the adjancency array A, that is, neighbors of vertex v starts from $A(B(v))$ and ends at $A(B(v+1)) - 1$. CSR is most widely used format in many algorithms including triangle counting [13]. In this paper, we utilize both formats for different purposes which will be presented later.

Intersection Operation. Triangle counting can be described as many intersection operations, where each intersection is performed between neighbor lists of the two vertices of the

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

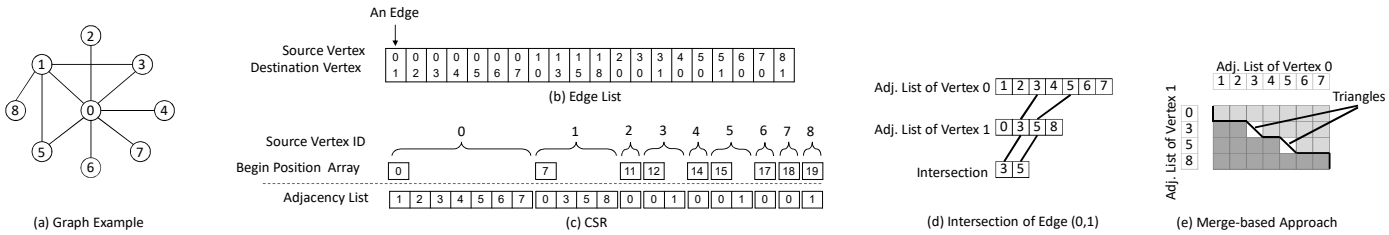

Fig. 1: Undirected graph, various representation format, and intersection of base edge (0,1).

Algorithm 1 Triangle counting algorithm using intersection

```
1: G = (E, V)
2: foreach v ∈ V in parallel do
3:     foreach w ∈ N(v) do
4:         count += Intersection(v, w);
5:     end for
6: end for
```

edge. We refer this edge as *base edge*. The algorithm is listed in the Algorithm 1, and one example is also shown in Figure 1(d). Figure 1(e) shows the merge based intersection approach. Prior works [20], [7] have used this approach on CSR format of the graph for triangle counting.

Rank-by-degree Orientation. Conventional graph data structure stores each edge of an undirected graph twice in both direction. With the intersection algorithm described above, each triangle will be counted for six times. A pre-processing technique, rank-by-degree orientation, is used in triangle counting works [20] to store only one direction of each undirected edge, it is, edge (u, v) is stored only if $d(u) < d(v)$ or $d(u) = d(v), u < v$. As this step simplifies the counting task, we also adopt it in this work.

III. Architecture

In this work we aim to use multiple GPUs on a distributed system to provide fast triangle counting on big graphs. To achieve the goal, there are several challenges that we need to address. First, GPU memory size is limited (up to about ten gigabytes) and hard to hold the whole dataset for big graphs. Second, current GPU-based triangle counting algorithm uses the merge-based intersection (which is used on CPUs) and fails to achieve very good performance.

While several external memory triangle counting algorithms [11], [7], [2] are proposed before, they run with a fixed memory bound, which cannot support distributed GPU-based triangle counting. For example, GPU-based triangle counting requires edge-centric parallelization to mitigate intra-warp workload imbalance which could lead to heavy divergence penalty [24]. Further, state-of-the-art external memory works use a streaming approach which support only a small proportion of work at any time, thus cannot saturate thousands of cores of a GPU. In addition, the scalability of existing external algorithms on distributed system suffers from workload imbalance, and since each task is tied to one partition, a distributed system implementation cannot directly enjoy dynamic scheduling for workload balancing.

TriX is designed to be a highly scalable system that can work in many system configuration from a shared memory system to distributed memory system utilizing the available GPUs and disks. It has two main components, 2-D partitioning for external memory algorithm, and GPU-based intersection. First, the 2-D partitioning algorithm divides the triangle counting task into multiple subtasks. Each of the subtasks is an in-memory triangle counting instance that can be executed on one GPU, and requires two CSR partitions and one edge list partition to be loaded to the GPU memory. Second, for each sub-task instance on GPU, we design a GPU specific binary search based intersection to optimize memory access pattern and intra-warp parallelization.

Note that our proposed external memory algorithm can support edge-centric parallelization through edge list streaming buffer and each sub-task can saturate GPU computation power. On the coarse grainularity, the number of subtasks for the proposed external memory algorithm is larger than the task number of state-of-the-art external memory algorithm, which can enjoy almost perfect scalability through dynamic load balancing. It also achieves better I/O complexity. Figure 2 shows the high level design of TriX.

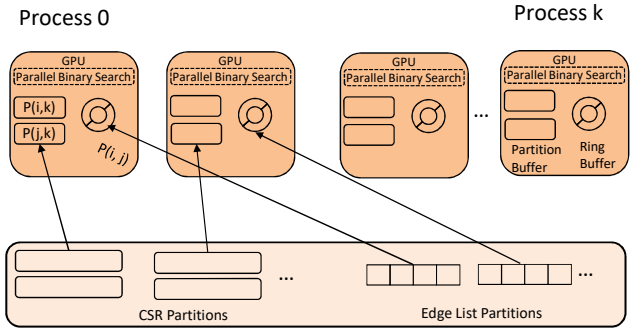

Fig. 2: TriX Architecture

IV. Graph Partitioning

A. 2-D Managed Partition

TriX proposes an advanced partitioning method based on the idea of balancing the 2-D partitions as much as possible so that all the partitions can be roughly of equal size, and at the same time it should be easy to track the edge membership, that is, which partition contains a particular edge.

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

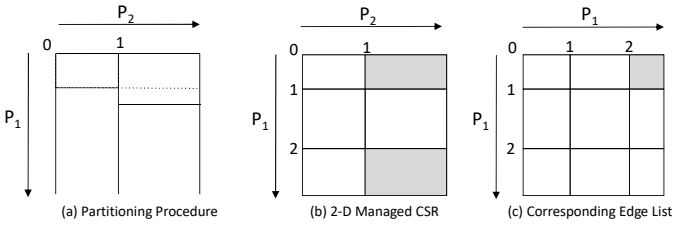

Fig. 3: 2-D Managed Partitioning. TriX uses it to get a simple as well as balanced partitions. The gray boxes shows a sub-task comprising of one edge-list partition and corresponding two csr partitions.

Figure 3(a) shows the procedure to create 2-D managed partition. To achieve this, we first do a 1-D vertical partitioning of partition count P_2, so that each vertical partition is of roughly equal size. This is achieved using different vertex range for each vertical partitioning. Then, we do a horizontal partitioning of each vertical partition. We again follow the same strategy and hence, each horizontal partition will have different vertex range. For example, the solid line shows the first horizontal partition of each vertical partition such that each 2-D partition is no more greater than size M. However, it would be very complex to track the vertex range of each horizontal partition and perform triangle counting. Instead, we simplify the horizontal partitioning. We choose the horizontal vertex range from each horizontal partition and select the one with minimum range. And, then we do a straight horizontal cut there, resulting in P_1 partitions . Figure 3(b) shows the result of final partitioning of size $P_1 \times P_2$.

Thanks to 2-D managed partitioning method, triangle counting in TriX is achieved by dividing the task to many smaller triangle counting sub-task based on few partitions. We assume the number of sub-tasks $P_1 \times P_1 \times P_2$ is larger than number of GPUs, and thus, dynamic scheduler can perfectly balance the workload of all GPUs.

B. Support for Fine-Grained Parallelism

Our 2-D partitioning method clearly allows a well-balanced workload distribution among many GPUs by dividing the whole task into many sub-tasks. A sub-task is identified using three suffix: i, j and k, which represents a partition in our 2-D managed partitioning method. Partition (i, j) is used as a collection of edges, called base partition, to decide which two vertices' neighbors should perform intersection. Then, the two neighbor lists are chosen from the partitions (i, k) and (j, k) respectively for the corresponding vertices, and the intersection is performed on them. Thus, each sub-task can independently be scheduled on any GPUs.

However, using partition (i, j) in the CSR format as a base partition does not provide fine-grained parallelism within a GPU. It can only provide a coarse-grained parallelism based on vertex-centric approach, where each warp has to handle all the edges of one source vertex, which would have resulted in workload imbalance inside the GPUs warp.

To this end, TriX provides an edge-centric based work division, and hence, a 2-D partitioned edge list, as shown in Figure 3(c), is used as group of base edges. The 2-D edge list contains the graph data in edge-list format, thus it is easy to divide the work within a GPU, where each warp handles one base edge at a time, i.e. performs intersection of the neighbor lists of the two vertices of one edge.

The 2-D partitioned edge-list does not have to be a balanced partition like its corresponding CSR partition. Indeed, we use edge-list partitions as base edges, and hence it is aligned to horizontal partition of the corresponding CSR version, i.e. it is of size $P_1 \times P_1$ partitions. This simplifies the sub-task formation as two sub-tasks never overlaps with each other. Moreover, the IO cost is completely amortized using a streaming buffer, where only few edges from the edge-list partition of the sub-task are streamed to GPUs for the one intersection task. Thus, a sub-task in one GPU consists of many streaming steps.

V. Binary Search Based Intersection

In this section, we discuss that the merge based intersection algorithm is not well-suited for GPU architecture, and present a binary search based intersection algorithm. Overall, it achieves much better memory bandwidth, experiences much fewer memory traffic compared against merge based intersection method and thus highly improves the computation efficiency.

A. Shortcomings in Merge based Intersection in GPUs

GPU architecture is very different than CPU architecture. In CPUs, number of cores are limited, each core operate independently, and each core has its own hardware cache. But in GPUs, a number of cores make a group called warp, and execute together in SIMD fashion, and share a small last level cache. The cores or threads of a warp are not independent like CPU cores, instead they all work together. And hence, conventional merge based triangle counting algorithm is not suited for GPUs as explained in Figure 5 for the example of Figure 1(e).

Within a warp in GPUs, merge-based intersection need to have two steps: divide the work among the threads of a warp as shown in Figure 5(a); and execute the work in individual threads [9]. Both of these steps are inefficient. First, one needs to scan the whole workload to figure out the division of work, which takes non-trivial operations. And during execution phase, the threads will have strided memory access within a warp. For example in Figure 5(b), thread 1 is comparing the content of the first section of the two arrays $\{1, 2, 3, 4\}$ and $\{0, 3\}$. Meanwhile, thread 2 is comparing the other of those two arrays $\{5, 6, 7\}$ and $\{5, 8\}$. This strided memory access has shown to be very inefficient in GPUs as they under-utilize the available memory bandwidth [15], [14].

B. Parallel Binary Search on GPU

A binary search based intersection algorithm takes two arrays, the neighbor lists of the two vertices of an edge as the inputs. The algorithm selects one neighbor array as *lookup array*, while the other as *binary search array*. Then, each element of the former is searched in the latter array to find

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

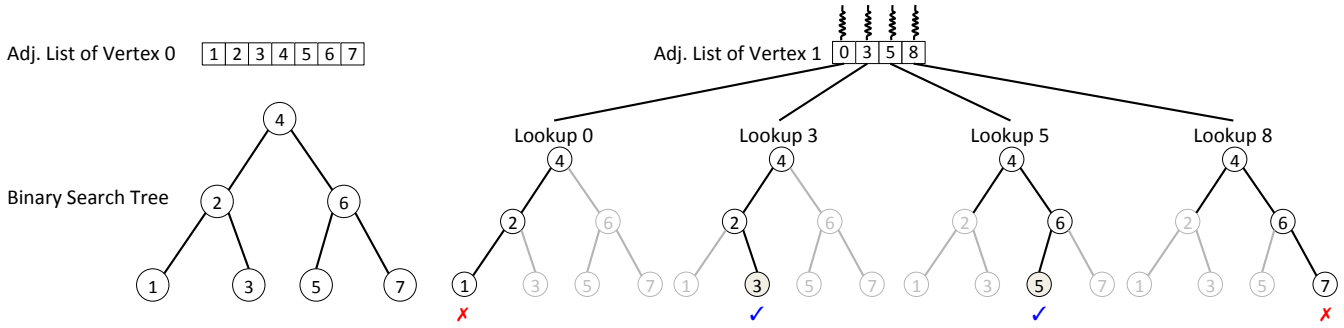

Fig. 4: Binary search based intersection: binary search tree, and parallel lookup process

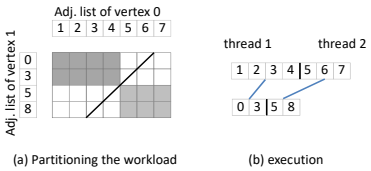

Fig. 5: Workload division for merge based intersection.

the intersection result. In each step of the binary search, the algorithm compares the lookup element to the middle one of the current search range in binary search array, and halves the search space.

The basic idea is illustrated in Figure 4. However, instead of using a real binary search tree structure, we use the neighbor list directly, which is already sorted, similar to what is used in merge based intersection. Each lookup key takes $O(\log n)$ time cost on an array with n elements. Before we always compare the lengths of both lists to use the longer neighbor list as a binary search tree to reduce the cost.

Binary search based approach may not be well-suited for CPU based architecture, but its unique benefits are clear on GPUs, achieving an efficient intra-warp parallelization and coalesced memory access.

Intra-warp Parallelism. In the binary search intersection, each thread is assigned to a consecutive elements of the lookup list, thus there is no separate step for partitioning and distributing workload within the warp. Comparing to merge based intersection, there is no need to figure out the range of keys for each thread to enable multiple threads to take equal workload of the intersection.

For example, the two neighbor lists as in Figure 4, we assign different threads to work on four lookup keys. Since, the depth of each binary search tree is log-scale to the length of the corresponding neighbor list, which is the largest number of operations to search the tree, thereby imbalance across threads might be limited to few operations for any intersection. Further, because majority of the lookups fail to find a match, and the binary search tree is a balanced tree that each leaf has almost same depth, thereby the workloads across threads are balanced in the worst case.

Coalesced Memory Access. Binary search-based intersection also shows better memory access patterns than that of merge-based. Accessing lookup key arrays is definitely consecutive and sequential, as four consecutive threads of the warp load four consecutive lookup keys as shown in Figure 4. In this case these entries can be loaded in coalesced access provided by GPU hardware which uses only one load transaction for many access. Performance counter shows that optimized binary search intersection reduces 80% load transactions comparing to merge-based one. For binary search tree, since each lookup needs to checks the same binary search tree, accessing binary search tree array also experiences high cache hit ratio.

VI. K-Truss

K-truss is defined as a subgraph, derived from the original graph where each edge has at least $k-2$ triangles. For example, a k-clique is a k-truss. K-truss is strongly related to triangle counting algorithm, and its implementation is always based on triangle counting algorithms.

The conventional k-truss algorithm iteratively performs two steps: (1) count the triangles for each edge, and (2) remove all edges from the graph whose triangle count is less than $k-2$, until it converges [6]. One can see that the major task of a k-truss algorithm consists of counting triangles for edges iteratively.

A reverse approach is also possible, that is, instead of counting the triangles for whole updated graph in each iteration, decrement the triangle count of each removed edge in each iteration. Considering that the number of newly removed edges are typically smaller than the number of remaining edges in the graph, the latter approach may bring overall considerable speedup in k-truss counting.

However, there are two main bottlenecks in the latter approach. First, undirected input graph is used which leads to low efficiency on counting the triangles on the original graph. As we mentioned earlier, using oriented graph to count triangles bring extra-ordinary speedup, which current k-truss algorithm does not use and leads to at least one order of magnitude slow down. Second, k-truss takes heavy space cost. It consumes large amount of memory size ($|E|$) to store support number of all edges. Also, a key-value store indexing is often used to provide quick lookup of edge id by the two vertices.

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

TABLE I: Graph Specification and performance. The performance numbers for smaller graph are reported for 1 GPU only. Others are reported for 32 GPUs.

Name	Description	$\|V\|$	$\|E\|$	Triangles	Time(s)	MTEPS
email-EuAll	Email network from a EU research institution	265,215	364,481	267,313	0.003009322	121.12
soc-Epinions1	Who-trusts-whom network of Epinions.com	75,880	405,740	1,624,481	0.004831075	84.08
flickrEdges	Image relationships on Flickr (edges only)	105,939	2,316,948	107,987,357	0.004825446	32.59
RoadNet	Road network of California	1,965,207	2,766,607	120,676	0.0196982	140.45
graph500-scale18-ef16	Synthetic graph500 network of scale 18	174,148	3,800,348	82,287,285	0.126783	29.98
cit-Patents	Citation network among US Patents	3,774,769	16,518,947	7,515,023	0.1280756	128.98
Orkut	Orkut online social network	8,730,857	327,036,486	223,127,577	10.65	30.70
Kron-25-16	Kronecker (scale 25, degree 16)	33,554,432	523,611,003	22,535,831,016	72	7.27
Friendster	Friendster online social network	68,349,466	1,811,849,342	4,176,922,719	17.12	70.25
Twitter MPI	Twitter follower connection	52,579,682	1,614,106,187	55,428,217,664	43.51	41.64
Gsh-2015 [10]	WebGraph Dataset	988,490,691	33,274,090,228	1,784,146,861,411	2,029.03	16.40
Kron-30-16	Kronecker (scale 30, degree 16)	1,073,741,824	17,022,115,838	2,306,560,594,152	736.43	23.11
Kron-31-16	Kronecker (scale 31, degree 16)	2,147,483,648	34,101,759,806	1,074,908,326,232	2,079.33	16.40

In this work, we develop a preliminary version of k-truss targeting only the CPU architecture. We use rank-by-degree oriented graph, which is discussed in triangle counting algorithm, as our input data format for the k-truss. To store the support number (number of triangles) for each edges, an array with size $|E|$ is needed to present in memory. Instead of removing the edges in each iteration, we use a bitwise status array with size $|E|$ to track whether any edge is removed or not from the input graph.

Note that no other data structure, such as a key-value store, is needed to figure out the offset of an edge in both support and status arrays. Indeed, since in triangle counting algorithm, all edges are visited through the CSR format, the offset of any edge in adjacency list is exactly the offset of this particular edge in support array and status array. In summary, this design provides much faster k-truss and has smaller memory footprint which enables larger graph to be processed for a given memory size.

VII. Experiments

We implemented TriX in 2000 lines of in C/C++ and CUDA code. We used CUDA toolkit version 7.5, nvcc and nvprof. G++ version 4.9.2 (GCC) with OpenMP version 1.8, with compilation flag as $O3$ is used for host code. All the graph that we use for evaluations are listed in Table I. We use real-world graph as well as synthetic graphs. We divide the graphs in several categories grouped by different edge sizes. The largest real-world graph is named as kron31-16, which has around 2 billion vertices and 64 billion edges. We pre-process them using rank by degree, which makes the size of the graph half. For vertex ID representation, we used unsigned integer of 32bit size while for offset contains unsigned integer of 64bit size. The results exclude the timing spent on performing the partitioning.

We run all the experiments in two setups. 1) using a local cluster at The George Washington University. Each node in this cluster has Intel Xeon E-2620 2.0GHz 6-core processors with NVIDIA K20 GPUs (with 5 GB GPU memory) and 128 GB of RAM. We use up to 32 GPUs in experiments. And 2) a single machine with 512GB memory system with 1 NVIDIA K40 GPUs, and dual Intel Xeon E5-2683 CPU with each having 14 hyper-threaded cores. We use this setup to test 1 GPU configuration, and also to compare result against the serial code of graph challenge. K-truss experiments are also run on this system.

A. Performance of TriX For Ranking

For smaller graphs having edge count less than 1 billion edges, we use a single GPU setup to report the numbers, as they can easily fit in the GPU memory. In this case, IO time and data copy time is not reported, similar to Graph500 benchmarking rule [8].

We use a local cluster to run triangle counting on bigger graphs having edges more than 1 billion. In this case, we used 32 GPUs, each GPU is hosted in 1 machine. We list the total run time, as well as traverse edges per second (TEPS) result in Table I. The total time includes the IO time, data copy to GPUs and computation time. TEPS are calculated by dividing the total edges to the total run time.

B. Power Consumption Measurement

We run power consumption measurement while triangle counting is running. We performed power measurement in the single machine setup, where only one GPU is used for triangle counting. We report the power consumption for triangle counting algorithm only, avoiding the cost associated with IO and data copy phase. Table II shown the power consumption, and TEPS per watt of few graphs.

TABLE II: Power measurement for triangle counting

Graph Name	Power (Watt)	KTEPS/Watt
orkut	122.3	251.02
kron25-16	130.8	55.60
twitter-mpi	118.7	94.12
friendster	127.8	140.58

C. Comparison Against Serial Code of Graph Challenge

We choose few smaller graphs to compare the result of TriX against the serial code (c++) provided by the graph challenge github repository. All the experiments were performed in the single machine, with 1-GPU configuration. TriX is run on the GPU, while serial code is run using a single core of the CPU.

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

Table III compares the TEPS of the two systems, and also present the speedup achieved by TriX. In all, TriX is able to achieve tens of millions of TEPS, outperforming by several orders of magnitude.

TABLE III: Speedup of TriX compared to serial code provided by graph challenge. All experiments were run in 1 GPU. K=1,000 and M=1,000,000.

Graph Name	Serial TEPS	TriX TEPS	Speedup
cit-Patents	50.67K	128.98M	2,545
email-EuAll	1.99K	121.12M	61,011
flickrEdges	1.30K	32.59M	25,154
graph500-scale18-ef16	29.98K	29.98M	103,764
soc-Epinions1	4.07K	84.08M	20,654
RoadNet	421.64K	140.45M	333

D. Scalability Test

We use kron30-16, kron31-16 and Gsh-2015 graphs to show the scalability of TriX. Figure 6 shows the scalability result when the number of GPUs are increased from 1 to 32. TriX has an ideal scaling curve, and we notice that kron30-16 and kron31-16 scales very well, almost identical to idea scalability. Gsh-2015 shows slightly lower scalability. The scalability results are possible due to independent and balanced sub-tasks design offered by the managed 2-D partitioning strategy.

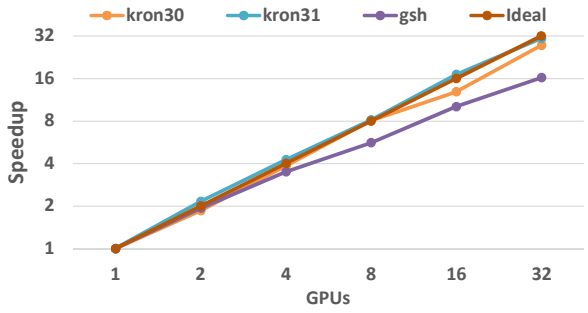

Fig. 6: Scalability of TriX

TABLE IV: Ranking results of k-truss in KTEPS. K = 1,000.

Graphs	twitter-mpi	friendster	orkut	kron25-16
k=3	5,540.86	12,409.93	29,730.59	5,903.17
k=4	533.24	1,432.30	4,523.33	5,305.08
k=5	139.94	942.20	2,137.49	5,540.86
k=6	437.66	556.12	2,319.41	1,637.31
k=7	305.87	1,242.69	3,865.68	1,590.73
k=8	288.03	1,241.84	11,199.88	1,546.02

TABLE V: Speedup of TriX k-truss (for k = 3) compared to python serial code of k-truss [18]. K = 1,000, M = 1,000,000.

Graph Name	Serial TEPS	TriX TEPS	Speedup
cit-Patents	66.10K	195.49M	2,957
email-EuAll	7.11K	202.49M	28,488
flickrEdges	7.23K	15.52M	2,145
graph500-scale18-ef16	1.27K	13.54M	10,692
soc-Epinions1	15.89K	42.89M	2,700
RoadNet	137.89K	477.00M	3,459

E. K-Truss

K-Truss is currently implemented in C++ for multi-threaded CPU version. Hence, we report the k-truss number on our single node machine, without using the GPU. Further development is under way. Table IV shows the k-truss performance in TEPS metric for various k values.

Table V compares our k-truss performance to serial python version of k-truss provided by graph challenge repository. The experiment is done for the value k=3 only. We are able to achieve several order of magnitude speedup across a number of different graphs.

VIII. Related Work

Triangle counting and listing algorithms were investigated early by Schank [19] and Latapy [13]. They show that intersection based algorithms with orientation, compact-forward in Latapy's paper, is the most time efficient algorithm. Algorithms other than intersection such as Hash-based algorithm along with corresponding hash table data structure is also considered in previous works. However inserting edges into hash table to construct the input data is costly, and also hash-based implementation performs much slower than merge-based intersection because of the cache behavior [4].

CPU based triangle counting systems [20], [13] are limited to a single machine and cannot take advantage of new compute devices such as GPUs. Also, the GPU based system [9] is based on the merge-based intersection inspired from CPU based system. Instead we present a parallel binary search based intersection that improves the performance in GPU.

Several triangle counting work [21], [17], [22] are based on MapReduce framework. Suri and Vassilvitskii [21], proposes to partition the graph in overlapped subgraphs to make all triangle appear in one of the partitions. It used a static scheduling that each machine covers equal number of vertices. A follow-up work [17] carefully classifies the type of triangles during partitioning so that each triangle is counted only once.

MPI based triangle counting works [2], [3] also use a static workload partitioning which relies on a expensive preprocessing to estimate the workload for each vertex. The latter one deploys a dynamic re-assignment scheme to guarantee the effect of load balancing.

IX. Conclusion

We presented TriX, a highly scalable, distributed and external triangle counting algorithm. It employs a novel 2-D managed partition and a parallel binary search based intersection for GPUs. TriX is able to scale to a large number of GPUs, and count triangles on billion-node graph (2 billion node, 64 billion edges) within 35 minutes, achieving 16.40 million traverse edges per second (TEPS).

Acknowledgment

The authors would like to thank the anonymous reviewers for their suggestions. This work was supported in part by National Science Foundation and Raytheon.

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

REFERENCES

[1] A. Aggarwal and S. V. Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A Parallel Algorithm for Counting Triangles in Massive Networks. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, 2013.

[3] S. Arifuzzaman, M. Khan, and M. Marathe. A Fast Parallel Algorithm for Counting Triangles in Graphs using Dynamic Load Balancing. In *Big Data*. IEEE, 2015.

[4] A. Buluç and K. Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011.

[5] S. Chu and J. Cheng. Triangle Listing in Massive Networks and Its Applications. In *SIGKDD*. ACM, 2011.

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

[7] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. PDTL: Parallel and Distributed Triangle Listing for Massive Graphs. In *ICPP*. IEEE, 2015.

[8] Graph500. http://www.graph500.org/.

[9] O. Green, P. Yalamanchili, and L. Munguía. Fast Triangle Counting on the GPU. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, 2014.

[10] Gsh dataset from WebGraph. http://law.di.unimi.it/webdata/gsh-2015/, 2015.

[11] X. Hu, Y. Tao, and C. Chung. Massive Graph Triangulation. In *SIGMOD*. ACM, 2013.

[12] P. Kumar and H. H. Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[13] M. Latapy. Main-memory Triangle Computations for Very Large (Sparse (Power-Law)) Graphs. *Theoretical Computer Science*, 2008.

[14] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[15] C. Nvidia. Programming guide, 2008.

[16] R. Pagh and F. Silvestri. The Input/Output Complexity of Triangle Enumeration. In *PODS*. ACM, 2014.

[17] H. Park and C. Chung. An efficient MapReduce Algorithm for Counting Triangles in a Very Large Graph. In *International conference on Conference on information & knowledge management*, 2013.

[18] 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 HPEC*, 2017.

[19] T. Schank and D. Wagner. Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study. In *Experimental and Efficient Algorithms*. 2005.

[20] J. Shun and K. Tangwongsan. Multicore Triangle Computations Without Tuning. In *Proceedings of the IEEE ICDE*, 2015.

[21] S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. In *International conference on World wide web*, 2011.

[22] W. Wang, Y. Gu, Z. Wang, and G. Yu. Parallel Triangle Counting over Large Graphs. In *Database Systems for Advanced Applications*, 2013.

[23] H. Yang and H. H. Huang. TriP: 2D Graph Partitioning for External-Memory Triangle Counting. Technical report, Department of Electrical and Computer Engineering, The George Washington University, 2018.

[24] H. Yang, H. H. Huang, and H. Liu. TriCore: Parallel Triangle Counting on GPUs. Technical report, Department of Electrical and Computer Engineering, The George Washington University, 2018.

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