Fast Triangle Counting on GPU

Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin
National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Laboratory
Cluster and Grid Computing Laboratory
Huazhong University of Science and Technology
Wuhan, China
{chygui, longzh, pcyao, xfliao, hjin}@hust.edu.cn

***Abstract*—Triangle counting is one of the most basic graph applications to solve many real-world problems in a wide variety of domains. Exploring the massive parallelism of the *Graphics Processing Unit* (GPU) to accelerate the triangle counting is prevail. We identify that the stat-of-the-art GPU-based studies that focus on improving the load balancing still exhibit inherently a large number of random accesses in degrading the performance. In this paper, we design a prefetching scheme that buffers the neighbor list of the processed vertex in advance in the fast shared memory to avoid high latency of random global memory access. Also, we adopt the degree-based graph reordering technique and design a simple heuristic to evenly distribute the workload. Compared to the state-of-the-art HEPC Graph Challenge Champion in the last year, we advance to improve the performance of triangle counting by up to $5.9\times$ speedup with $> 10^9$ TEPS on a single GPU for many large real graphs from graph challenge datasets.**

***Index Terms*—triangle counting, GPU**

I. Introduction

Graph data is ubiquitous in a broad range of fields, from social networks to semantic webs [1]–[3]. As part of graph analytics, subgraph isomorphism mining plays an important role in many real-world applications, e.g., discovering 3D motifs in protein structures and spam detection in web data. Highlighted as a representative of subgraph isomorphism problems by HPEC Static Graph Challenge [4], triangle counting, which is defined to find the number of triangles in a given graph, is becoming increasingly important in recent years.

With an increasing scale of graph data, it is highly demanded that triangle counting implementations provide desirable efficiency over large graph datasets. Moreover, the irregular data structure of the graph increases the difficulty to accelerate triangle counting. In response, numerous graph frameworks have been designed to enhance the productivity, including specifically-designed algorithms [5] and generally-purposed software systems [6]–[8].

However, these studies still fall short because of the limited parallelism in general-purposed CPUs. Therefore, recent researches propose to leverage GPU to achieve desirable performance for triangle counting [11]–[17]. Most of these studies focus on improving the load-imbalance caused by irregular graph structures to efficiently compute triangles. Fox et al. [13] extended logarithmic radix binning load-balancing scheme to the GPU. Hu et al. [12] adopted a 2D-parition method to balance the workloads among different GPUs. Bisson et al. [11] reordered the graph according to the degree of vertices to balance workloads.

Despite these research efforts, triangle counting still suffer from considerable performance impacts caused by heavy random memory accesses. Typically, triangle counting algorithms need the data of both vertices of an edge to detect triangles. Since the graph is always stored in the global memory of GPU for its large data size, these vertex accesses would lead to frequent strided memory accesses to global memory, leading to significant memory overheads. Fortunately, we find that the shared memory is friendly to this kind of memory accesses. To copy with this problem, we argue that the memory performance of triangle counting can be improved by managing both shared and global memory.

In this work, we design a prefetching scheme that alleviates the global random access penalty for accessing the neighbor list of a vertex in triangle counting. For load balance, the degree-based graph reordering is adopted that reduces the maximum neighbor size of the vertices to a large extent. In order to improve the utilization of computing resources, we design a heuristic to assign the vertices to kernels with different size. Finally, the experimental results show that for a number of benchmark graphs our implementation can achieve the rate at $> 10^9$ TEPS. The comparison with the Graph Challenge Champion in the last year [12] presents an improved performance by average $2.4\times$ and up to $5.9\times$.

The remaining of the paper is organized as follows. Section II introduces the preliminary knowledge and related research about triangle counting. Section III presents the details of the implementation and optimizations of our work. The evaluation results are discussed in Section IV. Finally, we conclude the paper and discusses some possible techniques that may improve the scalability and performance in Section V.

II. Related Work

We discuss different computational approaches and optimizations for triangle counting in this section. The related work on GPU is also discussed.

A. Triangle Counting Approaches

Given the standard notation $G(V, E)$ to represent a graph, the edge is denoted as (u, v). The notation $adj(u)$ and $adj(v)$

are used to represent the neighbor lists of two ends respectively. There are typically two kinds of approaches for counting the number of triangles of a given graph: the approach using set intersection and the linear algebra approach. In the former approach, the number of triangles associated with an edge (u, v) is counted by $|adj(u) \cap adj(v)|$. This process can be done in several methods, e.g., the sorted list intersection [18]–[20], the map-based method [21], and the binary search method [14]. A comparative study of these methods are presented in [9]. Leveraging the linear algebra approaches is also a promising way to accelerate the triangle counting. Prior work has designed many variations of matrix multiplication methods to compute the triangles on CPUs [22]–[24]. Existing linear algebra approaches on GPUs lack the efficiency for TC [15] and it would be potential to seek for specific optimizations.

In state-of-the-art merge-based sorted list intersection method [18], it requires dedicated partition strategies for efficient execution on GPUs in parallel and there exists frequent strided memory access [14]. The map-based method uses the hash-map to maintain the information of neighbors of the processed vertices and has lower search time complexity. However, there exist unavoidable random access to the hash-map among threads and extra overhead on hash-map transformation. For sparse lists, there exists redundant space for zero values. The binary search method searches the neighbor list directly for each element and has been witnessed to have the advantages of sequential memory access and high parallelism [14]. This work focuses on improving upon the binary search method for triangle counting for better efficiency on the GPU architecture.

B. Optimization Techniques

Prior work has proposed many efficient optimizations for triangle counting. For processing undirected graphs, triangles may be computed multiple times. Transforming the undirected graph into a directed graph can significantly reduce the workload [25]. This can be done by orienting an edge only from the vertex with smaller degree to the one with larger degree [26]. In order to further reduce the time complexity and the memory consumption, those one-degree vertices can be removed because they cannot contribute to any triangle [27]. In case that the memory capacity is tense, the bitmaps can be used to extend to larger graphs [28]. To achieve better performance and scalability, these methods are usually necessary to be applied to the implementations on the GPU.

C. GPU Algorithms

Exploring the massive parallelism of GPU to accelerate triangle counting has attracted a lot of research [10], [15], [16]. The Static Graph Challenge promotes a large amount of excellent work that achieves high performance and scalability [11], [12]. Most of the prior champions of the Graph Challenge adopt the GPUs for triangle counting acceleration. Bisson *et al.* [21] use a map-based approach that finds the common neighbors for each vertex and its direct neighbors. In this scenario, the bitmap is used to maintain large graphs in the GPU memory and a heuristic is designed to select appropriate kernels. Hu *et al.* [14] utilize the binary search based intersection for triangle counting that presents better intra-warp parallelism and coalesced memory access. The main focus of the work is to support efficient processing among multiple GPUs. The 2-D managed partition approach is used for load balancing. These projects have shown great opportunities for high performance graph processing on GPUs. Prior optimizations in these work also motivate our design.

III. Our Approach

In this Section the details of our implementation are presented. The triangle counting algorithm adopted is introduced first. We then illustrate the prefetching method for better memory performance. In order to reduce the prefetching buffer size, the degree based graph reordering is introduced. For higher resource utilization, we design a heuristic to improve the load balance based on the reordered graph.

A. The Binary Search based Triangle Counting

Several approaches for triangle counting on GPUs are presented in a comparative study in [15]. The results point out that the set intersection based implementation fits the best on GPUs. However, different kinds of strategies for set intersection can impact the performance. Green *et al.* [16] propose a merge-based method for triangle counting. For each pair of lists, the workload needs to be delicately divided to achieve high parallelism among threads. However, there exits strided memory access in a GPU warp. The binary search based intersection has the advantage of better memory bandwidth and computation efficiency compared to the merge-based approach [12]. Basically, each thread can process a consecutive target in the lookup list and the workload among the threads in a warp is similar. We also adopt the binary search based approach for its benefits. Note that there is no such algorithm that can perform the best for all kinds of workload [9], [17]. The strategies for choosing appropriate algorithms may impact the performance to a large extent which is beyond the scope of the paper.

Algorithm 1 depicts the implementation of the basic binary search based triangle counting. For each vertex u, the direct neighbor list $adj(u)$ of the vertex u are fetched first. After that, for each direct neighbor v in $adj(u)$ the algorithm also finds corresponding neighbors v' in $adj(v)$. The following step is to binary search $adj(u)$ (or $adj(v)$) for each element in $adj(v)$ (or $adj(u)$) according to the length of two lists. Each thread maintains a local count that records the number of triangles which is added to a global count in the end. Although the binary search method has the benefits of coalesced memory access and workload balance in a warp, the warp divergence may cause performance degradation because of many conditional operations [29]. Algorithm 2 presents an optimized binary search approach that avoids the branches as in [29].

B. Index-assisted Prefetching

Before the counting process, the graph is compressed and stored as the *Compressed Sparse Row* (CSR) format in GPU

Algorithm 1 Basic binary search based triangle counting.

```
1: nt = 0
2: function TriangleCount(V, E, N)
3:     for each u ∈ V do
4:         for each v ∈ ADJ(u) do
5:             if |ADJ(v)| <= |ADJ(u)| then
6:                 for each v' ∈ ADJ(v) do
7:                     ret = BINSEARCH(v', ADJ(u))
8:                     nt+ = VALID(ret)
9:             else
10:                for each u' ∈ ADJ(u) do
11:                    ret = BINSEARCH(u', ADJ(v))
12:                    nt+ = VALID(ret)
13:    return
14:
15: function BinSearch(x, array[0:ln-1])
16:    l = 0
17:    r = ln
18:    while l < r do
19:        temp = (l + r)/2
20:        if x < array[temp] then
21:            r = temp
22:        else if x > array[temp] then
23:            l = temp + 1
24:        else
25:            return temp
26:    return r
```

Algorithm 2 Optimized binary search approach.

```
1: function BinSearchOpt(x, array[0:ln-1])
2:     ret = 0
3:     temp = ln
4:     while temp > 1 do
5:         halfsize = temp/2
6:         candidate = array[ret + halfsize]
7:         ret+ = (candidate < x)?halfsize : 0
8:         temp− = halfsize
9:     ret+ = (array[ret] < x)
10:    return ret
```

global memory. Typically, there is an index array with the length of $|V| + 1$ and an edge array with the length of $|E|$. The index array contains the starting position information of the first out-edge of each vertex for accesssing the edge array. The edge array stores the sorted out-neighbors of each vertex. Note that the undirected input graph is preprocessed into a directed graph for workload reduction which will be detailed later.

Due to the irregular connection structure, the neighbors related to the two ends of a given edge are stored in non-consecutive positions. When a group of threads are assigned for a given vertex, each warp or each thread iterates over the out edges in parallel. Before the binary search operation, the edge offsets of the given vertex and its neighbors are needed for accessing edge information. However, random access occurs when fetching the offset information because the indexes of these vertices are usually not consecutive. During the binary search process, one of the two neighbor lists is randomly searched for the target elements in another list. These random access can influence the performance of the kernel when they occur in the GPU global memory where memory latency is relatively high. Considering the fast shared memory on GPU, it is important to leverage the shared memory for these random access to global memory.

In order to alleviate the impact of random access to global memory, we use an index-assisted prefethingmethod. Basically, certain thread block is used for processing a fixed scope of vertices. The edge offset information for the next vertex to process can be known in advance using the vertex indexes. The edge offset information are stored using the texture memory for faster access. Each thread prefetches the offset information before next round according to the block it belongs to. For each block, the neighbor list of the processed vertex will be accessed by the threads frequently for multiple times. After the prefetching of the offset information, the neighbor list information of the next vertex can be prefetched as well. Specifically, we use one warp for prefetching in each block while the other warps are doing the computations. In this way, the latency of accessing from the global memory is alleviated to an extent. Fig. 1 illustrates the prefetching scheme.

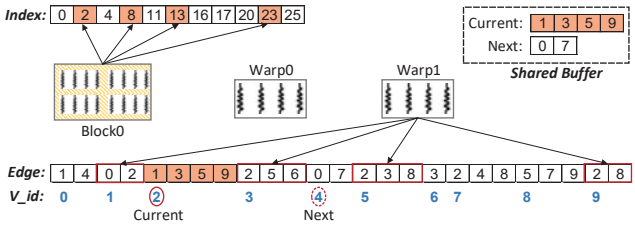

Fig. 1. Example of the graph storage format and the kernel implementation. Each thread block processes certain vertices. There exists random access to the index array and the edge array which is highlighted in orange background or red rectangle. Warp0 is used to prefetch the next neighbor list.

However, the prefetching method adopted for now is limited to the possible shared memory capacity on the GPU. Currently, the required space of the neighbor list buffer on the shared memory is equal to the maximum out degree of the vertices. Twice the size of the buffer is required to support prefetching. In order to support for processing larger graphs, two kinds of optimizations can be applied. The first is to use the bitmap to store neighbor list which can save a large memory space. Dividing the original neighbor list into small sections can also help for larger graphs.

C. Degree based Graph Reordering

The original input graphs are undirected in the Static Graph Challange. In order to reduce the workload, the undirected graph is transformed into a directed one as mentioned before [25], [27]. The undirected edges are directed from the vertex

with lower degree to the one with higher degree. By this way half of the edges are removed. The redundant computations for each vertex are also avoided. The most significant advantage of the degree based directing method is that the out edges of original vertices with high degree are reduced to a large extent. This not only makes the workload of vertices more equally distributed but also reduce the maximum degree which can help to prefetch for larger graphs.

However, the out degrees may still vary sharply in the order of original vertex indexes. An augmented degree based graph directing is adopted in [11]. By reordering the vertices in the order of descending degrees before the directing transformation, the vertices with close degrees are heuristically ordered together. This can help to achieve better load balance among different threads. Fig. 2 shows the distribution of neighbor size after reordering. In this paper, we adopt the transformation method in [11] to improve threads utilization.

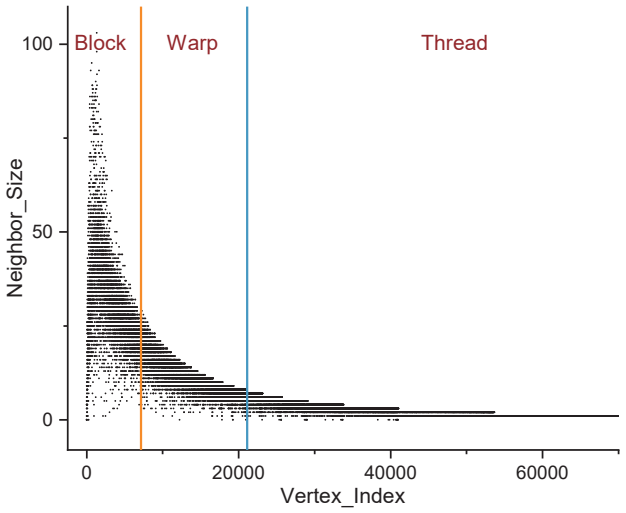

Fig. 2. The neighbor size distribution with vertex indexes of the graph soc-Slashdot0902. The yellow line and the blue line divide the vertices into three scopes according to the average degrees. Kernels with different size are assigned for each scope.

D. Support for Fine-Grained Parallelism

Previous optimizations help to achieve better memory access efficiency and load balance. In order to achieve higher performance, the utilization of GPU threads is another important issue. In the former configuration, each thread block iterates on a fixed number of vertices. After the degree based graph reordering, the vertices with relatively lower indexes usually have degrees higher than the vertices with higher indexes. Using a block for those vertices with low degrees may result in threads underutilization in the block. For this issue, we designed multiple size of kernels for vertices with different degrees. Specifically, the kernels are designed in the granularity of a thread, a warp, and a block. Before launching the kernels, we use a simple heuristic to distribute the vertices to different kernels. Because the vertices with high degrees stay close at the front of the vertex list, we can divide the vertex list into three scopes and compute the averaged degrees of each vertex scope during preprocessing. The three scopes are then assigned to three kinds of kernels respectively to achieve higher resource utilization. The schematic diagram is shown in Fig. 2. The kernels are executed concurrently during the processing in an asynchronous way.

IV. Experimental Evaluation

A. Experimental Setup

The experiments are operated on a machine equipped with a NVIDIA Tesla P100 GPU and two Intel Xeon E5-2680 v4 14-core CPU running at 2.40GHz with 256GB memory. The P100 GPU is with 56 SMs and 16GB HBM2 memory. We use CUDA 10.0 toolkit for GPU development and compilation, and G++ version 6.5.0 as the host compiler.

The examined graphs are retrieved from the Graph Challenge website, including both the real-world graphs and the synthetic graphs. These graphs are originally stored in the undirected structure. Note that the edges are reduced into half using the degree-based orientation before the counting process.

B. Compared to The 2018 Graph Challenge Champion [12]

We make an apple-to-apple comparison with the 2018 Graph Challenge Champion [12] that is also based on the binary search implementation. We both use the same type of P100 GPU as depicted above. The statistics are retrieved from the original paper and the speedup is presented in Fig. 3. For each graph we report the speedup of the kernel performance and the performance including the data transfer time. The results show that our implementation outperforms [12] for all the benchmarks on all the tested graphs. On the whole, we achieve on average the $2.4\times$ speedup and up to $5.9\times$.

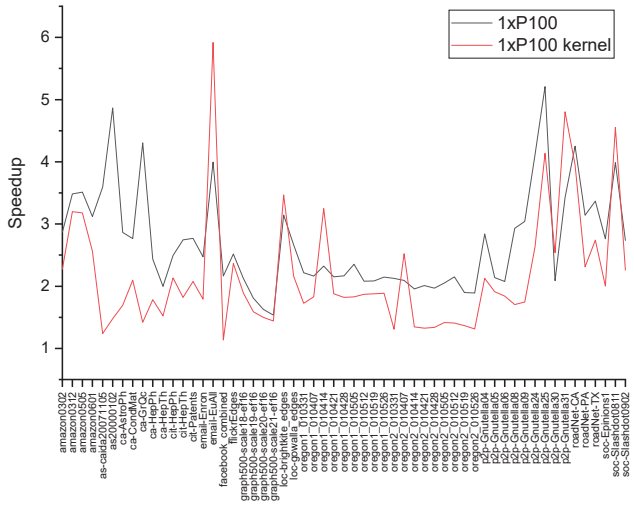

Fig. 3. The speedup of our implementation over the prior Graph Challenge champion [12] on single P100 GPU

In most cases, the speedup labeled as $1\times$P100 in Fig. 3 is higher than that of the kernel processing. The reason is that in [12] the counting process is executed in the parallelism

of edges. For better load balance, the edges are dynamically scheduled among the thread groups. In order to achieve this goal, there is extra overhead on transferring the edges from the host into the GPU memory. We instead utilize the parallelism of the vertices and statically assign the vertices to different thread groups. In this way, the transfer of the edges is avoided and the locality of the source vertices can be exploited. Although the load balance issue is more severe in the vertex based scheme, using a simple heuristic can alleviate this issue to a large extent as discussed in Section III.

C. Benefit Details

In this subsection we report and analyze the results in detail. The benchmark graphs are presented in Table I. For each graph, the counting process runs for 64 turns to get the avarage runtime and rate. The runtime labeled as 1×P100 is recorded when the data is ready in the host memory. We also report the kernel processing time labeled as *kernel* that excludes the time transfering the data from the host to the GPU memory. The rate is computed as the total edge number divided by the runtime. Exact counting number of triangles of each graph is presented for correctness evaluation.

Fig. 4 shows the rate of kernel processing and the rate considering the data transfer time. For kernel performance, the computing rates of a number of graphs are $> 10^9$ TEPS considering only the kernel time. Most of the graphs achieve $> 10^8$ TEPS rate. The highest rate is obtained with the roadNet-CA graph which is beyond 3×10^9 TEPS. After preprocessing the road graph is with a maximum degree by 4. The computing complexity is reduced to a large extent which contributes to the performance gains. Taking the data transfer time into consideration, Fig. 4 shows that the overhead on I/O is a bottleneck for small graphs. The counting time dominates the overhead for large scale-free graphs.

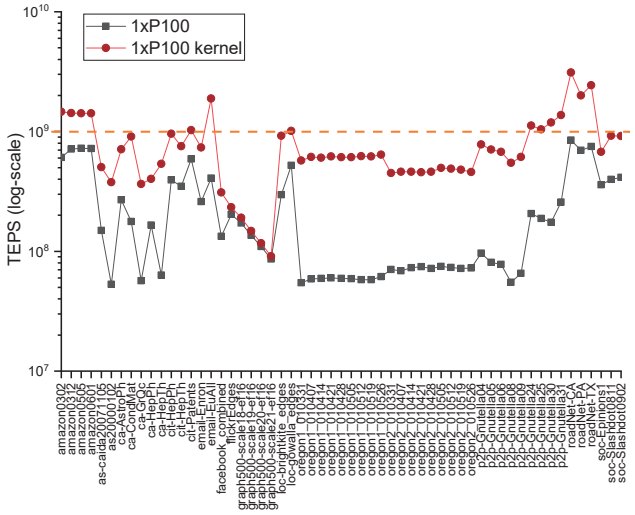

Fig. 4. The rate of each graph represented as TEPS in log-scale. The dash line indicates the results $> 10^9$ TEPS.

D. Sensitivity Study with Varying Edges

The report of the performance varying with different graph size is presented in Fig. 5. The graph size is represented as the edges number, because it is usually far larger than that of the vertices. The results show a sub-linear increase of the runtime with the graph size. In present implementation, for processing vertices with high degree we divide the neighbor list into small blocks to utilize the scarce fast shared memory. This can slightly introduce more computations for a vertex. Utilizing the compressed bitmap array [20] may help to reduce the memory consumption and improve the performance of larger graphs. It would also be interesting to combine the state-of-the-art partitioning schemes and scheduling strategies to scale our design to multiple GPUs in a distributed environment [12], [14], [30].

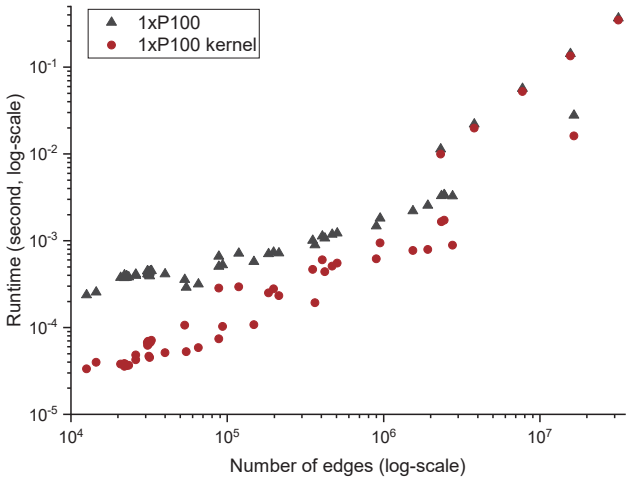

Fig. 5. The runtime for processing graphs of different size in the log-scale. As the graph size grows larger, the runtime is increasing in a sub-linear way.

V. Conclusion

Triangle counting is well-studied in the perspective of load-balancing on GPUs. The intrinsic random access can still limit the performance. In this work we design a prefetching scheme for better efficiency of the memory hierarchy of the GPU for triangle counting by buffering the neighbor list of the vertex in advance. In order to reduce the pressure on scarce memory space we adopt the degree-based graph reordering method that minimize the maximum degree to a large extent. We also use a simple heuristic based on the reordered graph to assign the workload for better resource utilization. The experimental evaluation shows that our work outperforms the prior Graph Challenge champion at the average speedup of $2.4\times$ and up to $5.9\times$ on single GPU. In the future work, we will scale our design to multiple GPUs for larger graphs and utilize compression techniques to save the memory consumption.

Acknowledgement

This paper is supported by the National Key Research and Development Program of China under Grant No.

TABLE I
PERFORMANCE DETAILS OF DIFFERENT BENCHMARK GRAPHS ON SINGLE P100 GPU

Dataset	$\|V\|$	$\|E\|$	$\#Triangles$	Runtime (second)		Rate (TEPS)	
				1×P100	1×P100 kernel	1×P100	1×P100 kernel
amazon0302	262,111	899,792	71,719	1.48×10^{-3}	6.17×10^{-4}	$6.08\times10^{+8}$	$\mathbf{1.46}\times10^{+9}$
amazon0312	400,727	2,349,869	3,686,467	3.29×10^{-3}	1.65×10^{-3}	$7.14\times10^{+8}$	$\mathbf{1.42}\times10^{+9}$
amazon0505	410,236	2,439,437	3,951,063	3.37×10^{-3}	1.72×10^{-3}	$7.24\times10^{+8}$	$\mathbf{1.42}\times10^{+9}$
amazon0601	403,394	2,443,408	3,986,507	3.39×10^{-3}	1.72×10^{-3}	$7.21\times10^{+8}$	$\mathbf{1.42}\times10^{+9}$
as-caida20071105	26,475	53,381	36,365	3.57×10^{-4}	1.06×10^{-4}	$1.50\times10^{+8}$	$5.04\times10^{+8}$
as20000102	6,474	12,572	6,584	2.37×10^{-4}	3.34×10^{-5}	$5.30\times10^{+7}$	$3.76\times10^{+8}$
ca-AstroPh	18,772	198,050	1,351,441	7.38×10^{-4}	2.79×10^{-4}	$2.68\times10^{+8}$	$7.10\times10^{+8}$
ca-CondMat	23,133	93,439	173,361	5.27×10^{-4}	1.03×10^{-4}	$1.77\times10^{+8}$	$9.07\times10^{+8}$
ca-GrQc	5,242	14,484	48,260	2.55×10^{-4}	3.97×10^{-5}	$5.68\times10^{+7}$	$3.65\times10^{+8}$
ca-HepPh	12,008	118,489	3,358,499	7.16×10^{-4}	2.95×10^{-4}	$1.65\times10^{+8}$	$4.02\times10^{+8}$
ca-HepTh	9,877	25,973	28,339	4.11×10^{-4}	4.83×10^{-5}	$6.32\times10^{+7}$	$5.38\times10^{+8}$
cit-HepPh	34,546	420,877	1,276,868	1.07×10^{-3}	4.40×10^{-4}	$3.94\times10^{+8}$	$9.57\times10^{+8}$
cit-HepTh	27,770	352,285	1,478,735	1.01×10^{-3}	4.67×10^{-4}	$3.49\times10^{+8}$	$7.54\times10^{+8}$
cit-Patents	3,774,768	16,518,947	7,515,023	2.80×10^{-2}	1.61×10^{-2}	$5.90\times10^{+8}$	$\mathbf{1.03}\times10^{+9}$
email-Enron	36,692	183,831	727,044	7.08×10^{-4}	2.50×10^{-4}	$2.60\times10^{+8}$	$7.35\times10^{+8}$
email-EuAll	265,214	364,481	267,313	8.94×10^{-4}	1.93×10^{-4}	$4.08\times10^{+8}$	$\mathbf{1.89}\times10^{+9}$
facebook_combined	4,039	88,234	1,612,010	6.63×10^{-4}	2.85×10^{-4}	$1.33\times10^{+8}$	$3.10\times10^{+8}$
flickrEdges	105,938	2,316,948	107,987,357	1.14×10^{-2}	9.93×10^{-3}	$2.03\times10^{+8}$	$2.33\times10^{+8}$
graph500-scale18-ef16	174,147	3,800,348	82,287,285	2.21×10^{-2}	1.99×10^{-2}	$1.72\times10^{+8}$	$1.91\times10^{+8}$
graph500-scale19-ef16	335,318	7,729,675	186,288,972	5.68×10^{-2}	5.22×10^{-2}	$1.36\times10^{+8}$	$1.48\times10^{+8}$
graph500-scale20-ef16	645,820	15,680,861	419,349,784	1.43×10^{-1}	1.34×10^{-1}	$1.10\times10^{+8}$	$1.17\times10^{+8}$
graph500-scale21-ef16	1,243,072	31,731,650	935,100,883	3.68×10^{-1}	3.49×10^{-1}	$8.62\times10^{+7}$	$9.09\times10^{+7}$
loc-brightkite_edges	58,228	214,078	494,728	7.22×10^{-4}	2.32×10^{-4}	$2.97\times10^{+8}$	$9.23\times10^{+8}$
loc-gowalla_edges	196,591	950,327	2,273,138	1.82×10^{-3}	9.41×10^{-4}	$5.22\times10^{+8}$	$\mathbf{1.01}\times10^{+9}$
oregon1_010331	10,670	22,002	17,144	4.02×10^{-4}	3.84×10^{-5}	$5.47\times10^{+7}$	$5.73\times10^{+8}$
oregon1_010407	10,729	21,999	15,834	3.75×10^{-4}	3.58×10^{-5}	$5.87\times10^{+7}$	$6.14\times10^{+8}$
oregon1_010414	10,790	22,469	18,237	3.78×10^{-4}	3.72×10^{-5}	$5.94\times10^{+7}$	$6.04\times10^{+8}$
oregon1_010421	10,859	22,747	19,108	3.79×10^{-4}	3.67×10^{-5}	$6.00\times10^{+7}$	$6.20\times10^{+8}$
oregon1_010428	10,886	22,493	17,645	3.79×10^{-4}	3.69×10^{-5}	$5.93\times10^{+7}$	$6.10\times10^{+8}$
oregon1_010505	10,943	22,607	17,597	3.83×10^{-4}	3.71×10^{-5}	$5.90\times10^{+7}$	$6.09\times10^{+8}$
oregon1_010512	11,011	22,677	17,598	3.92×10^{-4}	3.64×10^{-5}	$5.78\times10^{+7}$	$6.23\times10^{+8}$
oregon1_010519	11,051	22,724	17,677	3.92×10^{-4}	3.67×10^{-5}	$5.80\times10^{+7}$	$6.19\times10^{+8}$
oregon1_010526	11,174	23,409	19,894	3.81×10^{-4}	3.66×10^{-5}	$6.14\times10^{+7}$	$6.40\times10^{+8}$
oregon2_010331	10,900	31,180	82,856	4.42×10^{-4}	6.94×10^{-5}	$7.05\times10^{+7}$	$4.49\times10^{+8}$
oregon2_010407	10,981	30,855	78,138	4.48×10^{-4}	6.68×10^{-5}	$6.89\times10^{+7}$	$4.62\times10^{+8}$
oregon2_010414	11,019	31,761	88,905	4.35×10^{-4}	6.87×10^{-5}	$7.30\times10^{+7}$	$4.62\times10^{+8}$
oregon2_010421	11,080	31,538	82,129	4.23×10^{-4}	6.89×10^{-5}	$7.46\times10^{+7}$	$4.58\times10^{+8}$
oregon2_010428	11,113	31,434	78,000	4.37×10^{-4}	6.82×10^{-5}	$7.19\times10^{+7}$	$4.61\times10^{+8}$
oregon2_010505	11,157	30,943	72,182	4.14×10^{-4}	6.22×10^{-5}	$7.47\times10^{+7}$	$4.97\times10^{+8}$
oregon2_010512	11,260	31,303	72,866	4.28×10^{-4}	6.41×10^{-5}	$7.31\times10^{+7}$	$4.88\times10^{+8}$
oregon2_010519	11,375	32,287	83,709	4.49×10^{-4}	6.74×10^{-5}	$7.19\times10^{+7}$	$4.79\times10^{+8}$
oregon2_010526	11,461	32,730	89,541	4.52×10^{-4}	7.12×10^{-5}	$7.24\times10^{+7}$	$4.60\times10^{+8}$
p2p-Gnutella04	10,876	39,994	934	4.14×10^{-4}	5.13×10^{-5}	$9.66\times10^{+7}$	$7.80\times10^{+8}$
p2p-Gnutella05	8,846	31,839	1,112	3.93×10^{-4}	4.51×10^{-5}	$8.10\times10^{+7}$	$7.06\times10^{+8}$
p2p-Gnutella06	8,717	31,525	1,142	4.06×10^{-4}	4.67×10^{-5}	$7.76\times10^{+7}$	$6.75\times10^{+8}$
p2p-Gnutella08	6,301	20,777	2,383	3.77×10^{-4}	3.80×10^{-5}	$5.51\times10^{+7}$	$5.47\times10^{+8}$
p2p-Gnutella09	8,114	26,013	2,354	3.98×10^{-4}	4.24×10^{-5}	$6.54\times10^{+7}$	$6.14\times10^{+8}$
p2p-Gnutella24	26,518	65,369	986	3.17×10^{-4}	5.84×10^{-5}	$2.06\times10^{+8}$	$\mathbf{1.12}\times10^{+9}$
p2p-Gnutella25	22,687	54,705	806	2.91×10^{-4}	5.25×10^{-5}	$1.88\times10^{+8}$	$\mathbf{1.04}\times10^{+9}$
p2p-Gnutella30	36,682	88,328	1,590	5.06×10^{-4}	7.42×10^{-5}	$1.75\times10^{+8}$	$\mathbf{1.19}\times10^{+9}$
p2p-Gnutella31	62,586	147,892	2,024	5.74×10^{-4}	1.08×10^{-4}	$2.58\times10^{+8}$	$\mathbf{1.37}\times10^{+9}$
roadNet-CA	1,965,206	2,766,607	120,676	3.27×10^{-3}	8.90×10^{-4}	$8.46\times10^{+8}$	$\mathbf{3.11}\times10^{+9}$
roadNet-PA	1,088,092	1,541,898	67,150	2.21×10^{-3}	7.70×10^{-4}	$6.98\times10^{+8}$	$\mathbf{2.00}\times10^{+9}$
roadNet-TX	1,379,917	1,921,660	82,869	2.56×10^{-3}	7.90×10^{-4}	$7.51\times10^{+8}$	$\mathbf{2.43}\times10^{+9}$
soc-Epinions1	75,879	405,740	1,624,481	1.13×10^{-3}	6.00×10^{-4}	$3.59\times10^{+8}$	$6.76\times10^{+8}$
soc-Slashdot0811	77,360	469,180	551,724	1.18×10^{-3}	5.10×10^{-4}	$3.98\times10^{+8}$	$9.20\times10^{+8}$
soc-Slashdot0902	82,168	504,230	602,592	1.22×10^{-3}	5.50×10^{-4}	$4.13\times10^{+8}$	$9.17\times10^{+8}$

2018YFB1003502, National Natural Science Foundation of China under grant No. 61825202, 61832006, and 61702201.

REFERENCES

[1] 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," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716-727, 2012.

[2] L. Zheng, X. Liao, H. Jin, J. Zhao, and Q. Wang, "Scalable concurrency debugging with distributed graph processing," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 188-199.

[3] L. Zheng, X. Liao, and H. Jin, "Efficient and scalable graph parallel processing with symbolic execution," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 1, Article 3, 2018.

[4] 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 *Proceedings of the 2017 IEEE High Performance Extreme Computing (HPEC)*. IEEE, 2017, pp. 1-6.

[5] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proceedings of the International Conference for High Performance Computing*. IEEE, 2015, pp. 1-12.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX, 2012, pp. 17-30.

[7] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2013, PP. 135-146.

[8] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 456-471.

[9] J. Zhang, D. G. Spampinato, S. McMillan, and F. Franchetti, "Preliminary exploration of large-scale triangle counting on shared-memory multicore system," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-6.

[10] A. Polak, "Counting triangles in large graphs on GPU," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 740-746.

[11] M. Bisson and M. Fatica, "Update on Static Graph Challenge on GPU," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-8.

[12] Y. Hu, H. Liu, and H. H. Huang, "High-Performance Triangle Counting on GPUs," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-5.

[13] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader, "Fast and adaptive list intersections on the GPU," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-7.

[14] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "TriX: Triangle counting at extreme scale," in *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1-7.

[15] 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.

[16] O. Green, P. Yalamanchili, and L.M. Munguía, "Fast triangle counting on the GPU," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 2014, pp. 1-8.

[17] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader, "Logarithmic radix binning and vectorized triangle counting," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-7.

[18] O. Green, R. McColl, and D. A. Bader, "GPU merge path: A GPU merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*. ACM, 2012, pp. 331-340.

[19] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge path - parallel merging made simple," in *Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2012, pp. 1611-1618.

[20] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using SIMD instructions," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1587-1602.

[21] M. Bisson and M. Fatica, "Static Graph Challenge on GPU," in *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1-8.

[22] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804-811.

[23] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with KokkosKernels," in *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1-7.

[24] A. Yaşar, S. Rajamanickam, M. Wolf, J. Berry, and Ü. V. Çatalyürek, "Fast triangle counting using Cilk," in *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1-7.

[25] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 149-160.

[26] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1-7.

[27] R. Pearce, "Triangle counting for scale-free graphs at scale in distributed memory," in *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1-4.

[28] M. Bisson and M. Fatica, "High performance exact triangle counting on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3501-3510, 2017.

[29] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching," *Journal of Experimental Algorithmics*, vol. 22, no. 1, Article 3, 2017.

[30] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 752-768.